MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

The ANSI C (Internal) Spline Version 3.0 Application Program Interface

R. Waters, D. Anderson, A. Greysukh, W. Lambert, H. Kozuka, B. Perlman, V. Phan, D. Schwenke, S. Shipman, E. Suits, W. Yerazunis

TR97-11 December 1997

Abstract

This document describes the (Internal) Spline Version 3.0 Application Program Interface (API) as exported to ANSI C. It documents each publicly available object class and function. In addition, it describes the facilities available to those that wish to modify the system core. This is a machine generated document created from a database of information. It exists in both paper and HTML form. Other documents describe the API exported to other programming language environments.

MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY
http://www.merl.com

The ANSI C (Internal) Spline Version 3.0 Application Program Interface

R. Waters, D. Anderson,
A. Greysukh, W. Lambert, H. Kozuka, B. Perlman, V. Phan,
D. Schwenke, S. Shipman, E. Suits, and W. Yerazunis

TR-97-11 December, 1997

Abstract

This document describes the (Internal) Spline Version 3.0 Application Program Interface (API) as exported to ANSI C. It documents each publicly available object class and function. In addition, it describes the facilities available to those that wish to modify the system core.

This is a machine generated document created from a database of information. It exists in both paper and HTML form. Other documents describe the API exported to other programming language environments.

Contents

1 Introduction

This document describes the (Internal) Spline Version 3.0 Application Program Interface (API) as exported to ANSI C. It documents each publicly available object class and function. In addition, it describes the facilities available to those that wish to modify the system core.

This is a machine generated document created from a database of information. It exists in both paper and HTML form. Other documents describe the API exported to other programming language environments.

The following lists the classes described in this document. If a class is part of the external API, then its name is printed in bold. If a class is fundamental in the sense that if could not have been written by an application programmer, then its name is underlined. Note that many of the classes are not fundamental. These classes are included for convenience, but do not require special support from the system core. It is expected that application writers will write many more classes like them.

The API includes the following generic and particular applications:

spApp - A standard application (Section 2).
  spVisual - Visual renderer (Section 3).
  spAudio - Audio renderer (Section 4).

The API includes the following ordinary data structures:

spWM - A world model (Section 5).
spFn - Operation that can be mapped (Section 6).
spMask - World model view mask (Section 7).
spTransform - Position, axis, angle, scale specification (Section 8).
spVector - 3-element vector (Section 9).
spRotation - Rotation specified using axis and angle (Section 10).
spQuaternion - A quaternion (Section 11).
spMatrix - 4x4 transformation matrix (Section 12).
spPath - Stored motion path (Section 13).
spFormat - Sound data format (Section 14).

The API includes the following abstract classes that are multiply inherited:

sp - Specifies minimal sharable object (Section 15).
spPositioning - Specifies position and orientation (Section 16).
spDisplaying - Specifies visual appearance and extent (Section 17).
spLinking - Specifies link to large slowly-changing data (Section 18).
  spMultilinking - Link to index (Section 19).
spBeaconing - Specifies content-addressable object (Section 20).
spObserving - Specifies point of view (Section 21).
spVisualParameters - Specifies visual rendering parameters (Section 22).
spAudioParameters - Specifies audio rendering parameters (Section 23).

The API includes the following shared object classes:

spVisualDefinition - Link to graphic model (Section 24).
spSound - Link to stored sound (Section 25).
spLocale - Link to separate coordinate system (Section 26).
spBoundary - Link to bounding box (Section 27).
  spTerrain - Link to efficient 3D boundary description (Section 28).
spClass - Link to description of a shared class (Section 29).
spThing - Thing in the virtual world (Section 30).
  spRoot - Root of recognized whole (Section 31).
    spAvatar - Whole representing user or agent (Section 32).
spAudioSource - Source of sound. (Section 33).
spBeacon - Minimal Content-addressable object (Section 34).
spPositionedBeacon - Beacon specifying a position (Section 35).
  spSpeaking - Connects to user's speech (Section 36).
  spHearing - Connects to user's ears (Section 37).
  spSeeing - Connects to user's eyes (Section 38).
spSimulationObserver - Observer of basic data (Section 39).
spVisualObserver - Observer of visual data (Section 40).
spAudioObserver - Observer of sound data (Section 41).
spIntervalCallback - Interval callback (Section 42).
  spAlerter - Event detector (Section 43).
    spBeaconMonitor - Inspect beacons (Section 44).
      spBeaconGoto - Puts spThing beside beacon (Section 45).
spAction - Specifies program triggered by system core (Section 46).
  spOwnershipRequest - Requests getting ownership (Section 47).
  spDoSoundPlay - Plays sound data (Section 48).
  spMover - Supports smooth motion (Section 49).

1.1 Reading This Document

This document is a reference manual rather than a tutorial. In general, each topic is only discussed once and therefore any order of reading the sections in this document will not be quite right, because every section can be best understood only after having read many other sections.

The document is organized around the various classes enumerated above. At the top level, there is a section for each class. Within these sections, there are subsections corresponding to each instance variable and function.

There are two ways to look things up in this document. By using the table of contents, you can look instance variables and functions up by the name of the class the are in. The functions can be looked up alphabetically by using the quick reference index at the end of this document (Appendix B).

There are several quite different kinds of classes listed above: application templates, passive data structures, abstract classes that embody the key capabilities of the system and are multiply inherited, and shared objects that are the basis of communication between processes.

The application templates (e.g., spApp) show how simple applications can be written.

The passive data structures (e.g., spTransform) are pieces of data that are stored in shared objects or used in intermediate computations. In the C interface, these items are not objects, but rather just simple vectors and structures.

The basic abstract classes encapsulate fundamental features that are used by shared classes. A given shared class often makes use of several sets of features.

The shared objects consist of the remaining classes. These classes receive specialized treatment by the system. In particular, instances of the shared classes are communicated between processes. Applications can define new shared classes using a Java interface supported by a special Java preprocessor (Section 1.6).

Most of this document consists of descriptions of classes. The first line of s class section shows the first line of the declaration used in the Java interface to define the class. This is followed by a brief discussion of the class and a listing of the shared instance variables and functions in the class. Subsections of the section for a class describe each of its instance variables and functions in detail.

The first line of a subsection discussing an instance variable shows the declaration used in the Java interface to define the variable. This line includes both the type of the variable in Java (in the middle) and the type in C (in a comment at the end of the line). From the perspective of this document, only the C type is relevant. The variable declaration line is followed by a general discussion of the instance variable and the various access functions for manipulating it (Section 1.5.1).

The first line of a subsection discussing a function is the ANSI C signature of the function. This is followed by a summary of what the function does.

The names of C functions in the API are derived using the following scheme. If a class spK contains a method M, then the corresponding C function has the name spKM. Since every shared class name begins with the letters "sp", every C function in the API begins with the letters "sp".

At various places in this document, topics of interest are discussed at length in subsections. One reason for gathering information into a separate subsection is so that it can be referred to from many places in the document.

This copy of the documentation describes a number of object classes, instance variables, and methods that are used by the system core, but are not intended to be used by application programs. The titles of the sections that describe these features end in "(Internal)". From the perspective of an application writer, reading these sections could be more confusing than helpful, since the features described are not intended to be used by application programmers. Someone who is interested merely in writing applications is probably better off reading the external version of the documentation, which omits these sections.

1.2 System Overview

Above all else, the system provides a convenient architecture for creating Distributed Virtual Environments (DVEs) [see R.C. Waters and J.W. Barrus, "The Rise of Shared Virtual Environments," *IEEE Spectrum*, 34(3):20-25, March 1997]. This architecture is centered on a *world model* that mediates all interaction. Figure 1 illustrates the application programming model. It shows five applications interacting through the world model.

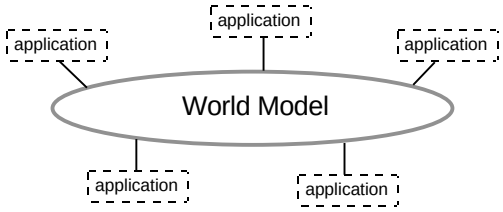

Figure 1: The programming model.

Applications do not communicate directly with each other, but rather only with the world model. This allows applications to be written without thinking about how communication is achieved. An application does exactly the same things when it is interacting with an application running in shared memory on the same machine as it does when interacting with an application connected via the Internet.

The world model is not a scene graph, but rather an object-oriented database that does not consider one kind of content to be any more important than another. In particular, we believe that audio information and autonomous behavior are at least as important as visual information and should not be limited by constraints inherited from visual rendering.

The world model specifies what objects exist in the virtual world, where they are, what they look like, and what sounds they are making. The world model does not contain historical information, but rather is just a snapshot of what the virtual world is like at the current moment. As the virtual world changes millisecond by millisecond, the world model changes.

The emphasis in the design of the world model is on the term *database*, not *object oriented*. The objects have methods associated with them, but by far the dominant operations consist of reading and writing data stored in the instance variables of world model objects.

Applications observe the virtual world by retrieving data from the world model. Applications affect the virtual world by adding, removing, and modifying objects in the world model. To avoid readers/writers conflicts, each object in the world model has one process as its owner and only the owning process can modify it. However, the ownership of an object can be transferred from one process to another (Section 47.1).

By itself, the system does not cause objects to persist over time. An object exists only so long as the process that owns it runs. To have persistent objects, an application must provide persistent processes that accept ownership of these objects. These processes could make use of a persistent file format for objects to provide efficient long term support for infrequently visited parts of a virtual world.

1.2.1 Scalability

Application programmers are encouraged to think in terms of Figure 1. However, it would not work well to use a centralized architecture when actually implementing the system. Rather, the system operates as shown in Figure 2. To provide low latency interaction with the world model, the world model is replicated so that a copy resides in each application process. Messages sent over a computer network linking the processes are used to propagate changes from one world model copy to another.

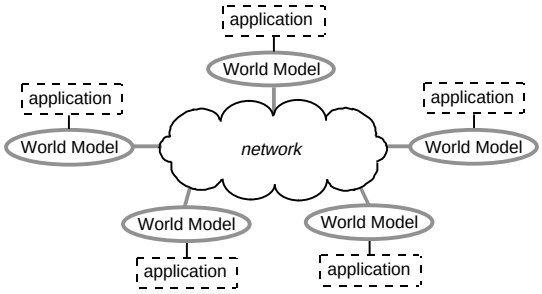

Figure 2: The communication model.

A central feature of the system is that it is designed to be scalable to a large number of users (e.g., thousands) interacting in real time. Two key features support this: providing only approximate equality of local world model copies and dividing the world model into chunks each of which is communicated only to the small group of users that are actually interested in it, rather than to all the users of the world model.

Distributed databases typically require that all local copies of the database always agree exactly on the information in the database. However, this requires object locking and handshaking that is incompatible with real-time interaction if there are more than a very small number of users. In contrast, the focus here is on real-time interaction at all costs, providing only approximate equality between world model copies.

The primary way in which world model copies are only approximately equal is that different users observe things as occurring at slightly different times. We call this a *relativity model* of communication and is actually not unlike the real world. When you hear sounds from distant sources, you do not hear the sound that is being made now, but rather the sound that was made seconds ago. As a result, people in different locations do not hear the same things at the same time. How great the differences are depends on how far apart the sound sources and people are.

Similarly, when an application process finds out about a world model change, it is not finding out about a change that is happening now, but rather about one that happened some time ago. How long ago depends on the network distance between the two processes. In general, this distance is not more than a couple hundred milliseconds and does not lead to world model differences that are unduly large.

Having only approximate equality of world model copies allows real-time interaction, but does not of itself prevent the computation required to maintain each local world model copy from growing in proportion to the total number of simultaneous users of a virtual world. To prevent this, the world model is broken up into many small chunks called *locales* (Section 26) and information about a given locale is communicated only to the small number of users that are near enough to that locale to be interested in it. Each locale is associated with a separate communication channel so that processes that are not interested in a locale do not have to expend any processing ignoring it. This allows the system to scale based solely on the maximum number of users that are gathered in any one locale, rather than on the total number of users in the virtual world.

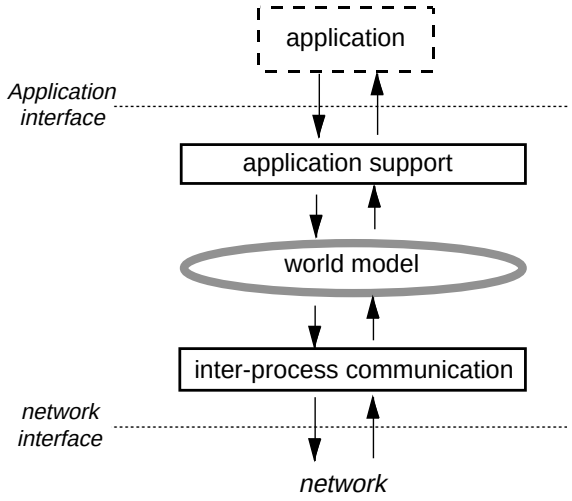

Figure 3: An application process.

Because the world model copy in an application process only contains information about the objects in the locales the process is attending to, a mechanism is needed for locating far away objects in the virtual world. This is done via specialized objects called *beacons* (Section 34). Specifically, the system provides a name service that makes it possible to locate beacons based on associated tags no matter where they are in the virtual world.

1.2.2 Application Processes

The structure of a single application process is shown in Figure 3. The dashed box at the top of the figure shows how the application itself fits into the picture.

The foundation of the system is the inter-process communication module shown at the bottom of Figure 3. It provides all the processing necessary to maintain approximate consistency between the world model copies associated with a group of communicating processes, sending messages describing changes in the world model caused by the local application and receiving messages from other processes about changes made remotely. The network interface specifies the format of these messages. Any process that obeys this interface can interoperate.

The messages sent are of three kinds, corresponding to three kinds of data in the world model: small rapidly changing objects, large slowly changing objects, and continuous streams of data. An important feature of the system is that it includes an efficient scheme for synchronizing these different kinds of data.

The most prevalent kind of object in the world model is small things that can change rapidly. For example, an object representing something in the virtual world (e.g., a chair) requires only a small description—i.e., to specify its position and orientation, whether it is contained in some other object, and which appearance should be used when displaying it. The features of small objects can be changed very rapidly.

Messages describing changes in small objects are sent using User Datagram Protocol (UDP) messages. This allows them to be communicated very rapidly. The objects must be small enough so that a message describing one will fit in one UDP message.

Graphic models, recorded sounds, and behaviors are represented using large objects. These objects are identified by Universal Resource Locators (URLs) and communicated using standard World Wide Web protocols. Standard formats are used (e.g., VRML for graphic models, WAVE for sounds, and Java for behaviors) so that standard tools can be used. There is no limitation on the size of the large objects, but several seconds can be required to communicate one. Fortunately, since these objects change infrequently, this latency can generally be masked by preloading the objects before they need to be used.

The final kind of object in the world model corresponds to continuous streams of data such as sound captured by a microphone. These streams are communicated in small chunks using UDP messages. At the moment, video streams are not supported. When they are, they will be communicated in a similar fashion, but of necessity using larger messages.

A central feature of the system is that using the various messaging approaches above, every kind of data in the world model can be communicated between processes. Therefore, applications can modify and extend every aspect of a virtual world. Furthermore, while it is often advantageous to prestore data for an application to use (i.e., by delivering it on a CD-ROM) this is not necessary. Once started, an application can fetch everything it needs that has not been prestored.

1.2.3 The API

The Application Program Interface (API), which described in this document, consists primarily of operations for creating/deleting objects in the world model and reading/writing instance variables in these objects. The application support module (see Figure 3) contains various facilities that make application writing easier. For the convenience of application writers, multiple APIs are provided—the principle APIs being in ANSI C and Java.

An interesting application support tool consists of special support for the smooth motion of objects (Section 16.13). The simplest way for an application to move an object along a trajectory from one position and orientation to another is to repetitively set the object's position and orientation to one spot after another along the trajectory. However, to get smooth appearing motion by this method, the position and orientation must be specified many times per second (i.e., 30 times or more) which leads to high computation and communication costs. Interpolation-based facilities are provided that allow smooth motion to be achieved while communicating at most a few positions and orientations per second.

Another common problem experienced in virtual worlds is the need to move an avatar or other object along the surface of the ground, while avoiding fixed obstacles. To support this, a general terrain following facility (Section 28) is provided. Given any 3D point, this facility can determine in only a few microseconds the height of the ground below a query point and whether a collision with a fixed obstacle is occurring. This is done by sorting 2D projections of the polygons describing the terrain in a virtual world into a grid for rapid access.

The API described in this document is fundamentally low level in nature. The focus in developing the system has been on providing a wide range of facilities that make many things possible, rather than on the ease of use of any particular part of the API. In the long run, a much higher level, easier to use API should be provided along with authoring tools that facilitate the creation of DVE content.

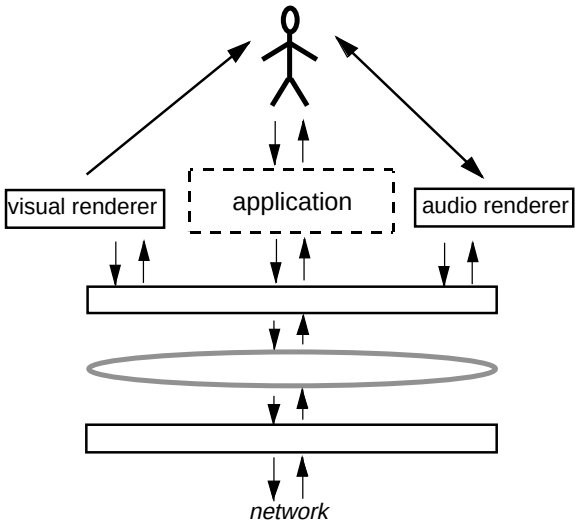

Figure 4: Typical configuration supporting a user.

1.2.4 Rendering

Figure 4 shows the system being used to support an application that interacts with a human user. The primary feature of the figure is that three applications are used in this situation. The main application (in the dashed box) presents an interface to the user.

Visual and audio rendering modules are provided; however rather than being combined into the system, they are separate applications interfaced to the system. They use the same API as other applications and do not have to be tightly coupled with the main application.

One advantage of the loose coupling of renderers is that the renderers interacting with a person can run in separate processes from the main application. This allows them to operate on separate machines in situations where maximum performance is required. However, the primary mode of operation is for them to run in the same process with the main application, sharing a single copy of the world model as shown in Figure 4.

The greatest advantage of the loose coupling between rendering and the system core is that the system is not tied to any one renderer. Rather, the system is designed to be easily interfaced to almost any renderer. Default renderers are supplied with the system, but it is expected that demanding applications will switch to renderers that are tuned to the task at hand.

Consider visual rendering as an example. In the world model, objects can have positions, orientations, and appearances. The visual renderer creates a scene graph by combining the appearances associated with the objects that are near enough to be seen and renders the scene graph from the vantage point specified by the application. The system itself does nothing with the graphic models that describe the appearances of objects. The only thing that matters is that the visual renderer being used can load them.

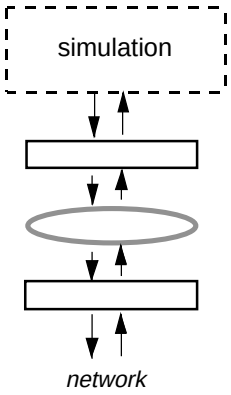

Figure 5: Typical configuration supporting a simulation.

Audio rendering is supported in a similar fashion. The world model contains objects representing sources of sounds. These objects specify the places where the sounds are located. They can either be point sources or diffuse ambient sources. Sound to be played through these objects can be captured live from a microphone or prestored in recorded sound objects. The audio renderer creates an *audio scene graph* by combining the sounds associated with sound source objects that are near enough to be heard and renderers this from the vantage point specified by the application. The system itself does nothing with sound encodings. The only thing that matters is that the audio renderer being used can decode them.

1.2.5 Supporting Simulations

From the earliest days of work on the system, we paid close attention to supporting interaction with computer simulations as well as people. The way this is done is shown in Figure 5.

A simulation operates just like any other application using the same API to interact with the world model. However, no visual or audio rendering is needed, because there is no person to see or hear anything. Complex simulations, intelligent agents, and large persistent databases can all be directly connected to a virtual world. Large powerful computers without support for graphics or sound can be used to manage shared content.

It is important to note that it is a great deal easier to say that a simulation interacts with the world model just like any other application than it is to make it possible for a simulation to effectively interact with the world model. The reason for this is that applications supporting a user have a human being in the loop and simulations do not.

For example, it is typically easy for a person to look at an avatar and determine which way the avatar is facing, based on a rendered image. However, it verges on impossible for a program to tell where the face of an avatar is by looking at a list of polygons. To deal with this kind of problem, the system has been designed to make information such as which way an avatar is facing easily accessible to programs. Specifically, the world model contains an object class spAvatar that is used to identify avatars as opposed to other kinds of objects and by convention, the center of the coordinate system for an object is at the center of the object, the Y axis is up, and the object faces down the negative Z axis

1.2.6 ISTP

The network communication interface in Figure 3 is called the Interactive Sharing Transfer protocol (ISTP) [Waters R.C., Anderson D.B., and Schwenke D.L., "Design of the Interactive Sharing Transfer Protocol", *Postproc. WET ICE '97 – IEEE Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, (MIT, June 1997), IEEE Computer Society Press, Los Alamitos CA, 1997] . ISTP is a hybrid protocol that uses three different kinds of communication for three different purposes.

The WWW Layer of ISTP uses HTTP to communicate large chunks of data such as graphical models, prerecorded sounds, and object class descriptors. This data corresponds to various kinds of spLinking objects (Section 18).

The peer-to-peer layer of ISTP uses multicasting to communicate world model changes and real time data streams as rapidly as possible directly from one process to another.

The control layer of ISTP uses TCP messages between ISTP processes to control the peer-to-peer layer and obtain various services that are provided by ISTP processes.

An essential feature of Figure 2 is that it does not contain any central server processes. To minimize latency, prevent bottlenecks, and maximize scalability, ISTP communication is peer-to-peer rather than passing through centralized processes.

Like the world wide web, the ISTP world is a single implicit entity spanning the globe. To enter this world, all one has to do is run an ISTP process. In particular, just as all the web servers in existence are implicitly combined into a single world wide web, all the ISTP processes that are running at any given moment are capable of interacting with each other. Running a web browser and knowing a Uniform Resource Locator (URL) (or running a web server and advertising a URL) are the only things that are necessary to participate in the world wide web. Similarly, running an ISTP process and knowing (or advertising) a beacon tag are the only things that are necessary to enter the ISTP world.

While there are no centralized servers in ISTP, there are nevertheless two key services that are provided in a distributed way. Which processes are providing these services at any given moment is determined dynamically. In particular, the number of processes providing a service can be dynamically increased in order to scale to larger numbers of users.

An ISTP process providing Locale-Based Communication service maintains a record of everything in a given locale. When the locus of attention of a user process enters a new locale, the appropriate locale server is queried to obtain initial information about the state of objects in the locale. After this initial download, the user process obtains further incremental information by peer-to-peer communication. Responsibility for locales is parceled out among many ISTP processes, so that no one process is responsible for a larger piece of the virtual world than it can easily handle.

An ISTP process providing Content-Based Communication service supports part of a distributed name service that allows ISTP processes to locate beacons based on their tags. When a user process wants to locate a particular beacon object, it consults the appropriate Content-Based Communication server to find the beacon. As with locales, responsibility for beacons is parceled out among a many ISTP processes, so that no one process is responsible for more beacons than it can easily handle.

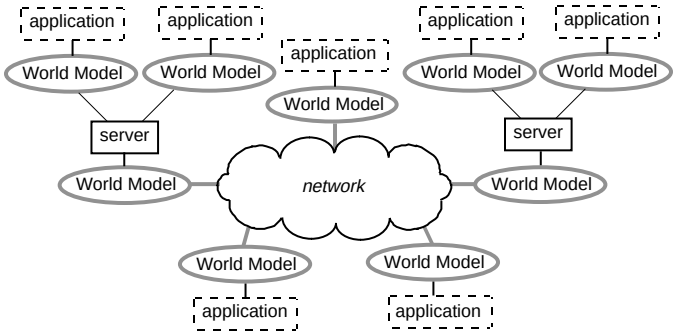

Figure 6: User servers.

1.2.7 User Servers

ISTP processes require relatively high bandwidth network connections. To accommodate processes with weaker network connections, ISTP utilizes *user servers*. as illustrated in Figure 6.

The purpose of a user server is to support users with slow network connections (e.g., modems) that do not allow them to operate as first class ISTP peers. A user server intercepts all communication to and from a given user. The message traffic to the user is then compressed to take maximum advantage of the bandwidth available. As part of this, audio streams are combined and localized before sending them to the user. Servers are replicated as needed so that no one server has to support more users than it can handle. Two user servers are shown in Figure 6.

1.3 Atomic Data

Much of the data used in the API is simple atomic data of 64 bits or less in length such as boolean values, integers, and floating point numbers. This data is copied whenever it is passed to or returned from a function.

The following paragraphs briefly describe each kind of atomic data used in the API. Some of these types of data are described at greater length in other sections.

Boolean values represented using the type spBoolean. These are used to indicate True and False values.

16-bit integers represented using the type short. These are used when low range integers are adequate.

32-bit integers represented using the type long. These are used for representing most integer values.

32-bit floating point numbers represented using the type float. In keeping with standard graphics practice, these are used to represent most non-integer values. They have limited precision, but are compact.

32-bit unique names represented using the type spName. These are a compressed form of longer GUIDs that are used for uniquely identifying objects when they are communicated between machines. The same name space is also used to identify the the processes that own objects. A compressed form is used to simplify operation on individual machines.

IP address/port pairs represented using the type spAddress. These are used to represent network addresses. They contain two pieces of information, a 32-bit IP host address and a 16-bit port number. In C an spAddress is a struct containing the host address, followed by the port number.

32-bit time durations in milliseconds represented using the type spDuration. Whenever an API function takes a time duration as an argument, this argument is in terms of milliseconds. For compactness, 32-bit integers are used. This allows a range of approximately plus or minus two weeks. (This does not impose any limitation on the lifetime of a session in the virtual world, but rather only limits the maximum difference between two times that can be represented.)

32-bit time stamps in milliseconds represented using the type spTimeStamp. Internally, the system uses absolute timestamps. At heart, these timestamps represent the time in milliseconds modulo one week (604,800,000 milliseconds). However, as discussed in [Waters R.C., *Time Synchronization In Spline*, MERL TR 96-09, April 1996] timestamps are manipulated in a non-standard way that allows very efficient comparison between timestamps

World model view masks represented using the type spMask (Section 7). These bit masks are used to control the visibility of shared objects in the world model.

Audio formats represented using the type spFormat (Section 14). These values are used to represent audio encodings and sample rates.

1.4 Pointer Data

In addition to atomic data, the API makes use of several different kinds of pointer data. As discussed at length in the next section, the most important kind of pointer data is pointers to shared objects. However, several other kinds of pointer data are used as well.

A key feature of the API is that pointer data is always passed into and returned from functions by reference rather than copying. This promotes efficiency, but means you must pay close attention to the following.

1- The system never alters data that is passed to it via a pointer unless this document explicitly states otherwise. This means you can depend on the value remaining the same unless you change it yourself.

2- The system never retains a pointer passed to it beyond the time the function that was given the pointer returns. (If the system needs to keep data, it copies it.)

3- When a pointer is passed to you, you must never alter the data pointed to unless it is explicitly stated that you should, because the system depends on the data not being altered. In particular, you must never free something referred to by a pointer unless you yourself allocated the storage.

4- It is allowed that you retain pointers returned by an API function, but only until the next time spWMUpdate is called. The reason for this is that the system assumes that during calls on spWMUpdate, it can free any storage it has allocated. The only exception to this is certain shared objects (Section 1.5.3). If you want to save data across a call on spWMUpdate, you must copy it.

The following paragraphs briefly describe each kind of pointer data used by the system. Many of these types of data are described at greater length in other sections.

A UTF8 ASCII string represented using the type char *. This is a null-terminated string. If the characters are all 7-bit ASCII characters, then this is an entirely ordinary string. If extended characters are used, then special escape sequences are present.

An immutable spFixedAscii string containing no more than 500 characters represented using the type spFixedAscii. This is the same as the above, but is limited in length so that it can fit into a single UDP message. It is used for most shared character data in shared objects. This data must be specified when the object is initially created and cannot be altered afterward.

An ASCII string containing no more than 31 characters represented using the type spAscii32. This is the same as the above, but is limited in length so that it can fit in 32 bytes.

A 17-element position, orientation, and scaling vector represented using the type spTransform (Section 8). This structure is used as the primary representation of the position and orientation of objects.

A 3-element vector represented using the type spVector (Section 9). Vectors containing three values are used for several different purposes.

A 4-element quaternion represented using the type spQuaternion (Section 11). Quaternions are one way of representing rotations.

A 4x4 transformation matrix represented using the type spMatrix (Section 12). Following standard graphics practice, the position, orientation, and scaling of objects can be represented using a 4x4 matrix.

A sequence of spTransforms and times represented using the class spPath (Section 13). These are used to represent recorded motions.

A locally defined operation represented using the class spFn (Section 6). The API makes heavy use of callback functions and the mapping of functions over objects. For convenience, a single type is used to represent functional arguments in all these situations.

An interaction window represented using the type spWindow. Exactly what type this is depends on the execution environment.

An opaque pointer represented using the type void *. In a number of places, the system records pointers that are not part of the external interface and are not intended to be manipulated by applications.

1.5 Shared Objects

The central kind of data in the API consists of shared objects. These objects are stored in the world model and shared between the participants in a session. (Other data is shared only if it is stored in a shared variable of a shared object.) The key feature of shared objects is instance variables. They can have methods associated with them as well; however, the system makes relatively little use of these methods. Rather, shared objects are essentially passive, principally just representing a database of information.

1.5.1 Accessors

All interaction with instance variables in shared objects is via access functions, rather than via direct access to instance variables or structure fields. In the external API, the number of access functions available depends on the visibility of the instance variable in the API. However, in the internal API, a full range of accessors is available for each variable. The following discusses the accessors in detail for an instance variable V in a shared class spK containing data of type T.

```
T spKGetV(sp Object)
```

Object - The object from which the value is to be obtained.
Return value - The value of the instance variable V.

The above accessor obtains the value of the instance variable V for an object. It is part of the API if and only if V is part of the external API. It reports an error (Section 5.18) if the object from which the data is being obtained is not an instance of (a subclass of) the class spK.

```
void spKSetV(sp Object, T Value)
```

Object - The object whose V value is to be set.
Value - New value for V.
Return value - There is no return value.

The above accessor sets the value of V. It is part of the API only if the external API allows V to be set. It reports an error if the object to be modified is not an instance of (a subclass of) the class spK, if the object has been removed from the world model, or if Value is a shared object that has been removed. If V is an instance variable that is shared between the processes in a session, then spKSetV reports an error if the owner of the object is not equal to the current value of spWMGetMe. In addition, spKSetV sets the MessageNeeded bit (Section 15.20) and the Change bit (Section 15.21). However, if spKSetV is executed in the Function of an spAction, then unless the object being modified is the action object itself, the MessageNeeded bit is not set. This is needed for the proper functioning of actions that run in both locale and remote processes.

```
T spKGetOldV(sp Object)
```

Object - The object from which the old value is obtained.
Return value - The old value of the variable V.

The above accessor obtains the value that V had at the end of the last call on spWMUpdate. It is part of the API if and only if V is part of the external API, V is a shared variable and the external API allows the value of V to change (in particular, V must not have the C type spFixedAscii). spKGetOldV reports an error if the object from which the data is being obtained is not an instance of (a subclass of) the class spK. Obtaining an old value can take substantially more time than obtaining the corresponding current value. (The primary intended use of spKGetOldV accessors is in alerter tests (Section 6.1).)

As noted, a given instance variable may not have all the accessors described above. If (and only if) a given accessor spKGetV (or spKSetV or spKGetOldV) is not included in the API, then an internal accessor with the name spKiGetV (or spKiSetV or spKiGetOldV, respectively) is typically included in the internal API. These internal accessors operate in exactly the same way as the external accessors described above. They are included in the internal API so that they can be used in the system core and so that they are available to those that wish to go beyond what can be done using the external API.

```
T spKiGetV(sp Object)
```

Object - The object from which the value is to be obtained.
Return value - The value of the instance variable V.

The above accessor obtains the value of the instance variable V for an object. It is part of the API if and only if V is not part of the external API. It reports an error (Section 5.18) if the object from which the data is being obtained is not an instance of (a subclass of) the class spK.

```
void spKiSetV(sp Object, T Value)
```

Object - The object whose V value is to be set.
Value - New value for V.
Return value - There is no return value.

The above accessor sets the value of V. It is part of the API only if the external API does not allow V to be set. It reports an error if the object to be modified is not an instance of (a subclass of) the class spK, if the object has been removed from the world model, or if the value being stored is a shared object that has been removed. If V is an instance variable that is shared between the processes in a session, then spKiSetV reports an error if the owner of the object is not equal to the current value of spWMGetMe. In addition, spKiSetV sets the MessageNeeded bit (Section 15.20) and the Change bit (Section 15.21). However, if spKiSetV is executed in the Function of an spAction, then unless the object being modified is the action object itself, the MessageNeeded bit is not set. This is needed for the proper functioning of actions that run in both locale and remote processes.

```
T spKiGetOldV(sp Object)
```

Object - The object from which the old value is obtained.
Return value - The old value of the variable V.

The above accessor obtains the value that V had at the end of the last call on spWMUpdate. It is part of the API if and only if V is a shared variable and does not have the C type spFixedAscii. spKiGetOldV reports an error if the object from which the data is being obtained is not an instance of (a subclass of) the class spK. Obtaining an old value can take substantially more time than obtaining the corresponding current value.

The internal C API has a further set of accessors for a given instance variable V. These accessors are much faster than the external API accessors but do no error checking and perform fewer collateral operations. They exist purely for implementing the core of the system itself and should never be used in application programs. They follow the same basic pattern as the accessors above, but are distinguished from them by having names that start with the letters "sq" instead of "sp".

```
T sqKGetV(sp Object)
```

Object - The object from which the value is to be obtained.
Return value - The value of the variable V.

The above accessor obtains the value of the instance variable V for an object. No validity checks are performed.

```
void sqKSetV(sp object, T value)
```

Object - The object whose V value is to be set.
Value - New value for V.
Return value - There is no return value.

The above accessor sets the value of V for an object. No validity checks are performed. The message needed bit (Section 15.20) and change bit (Section 15.21) are not set.

```
T sqKGetOldV(sp Object)
```

Object - The object from which the old value is obtained.
Return value - The old value of the variable V.

The above accessor obtains the value that V had at the end of the last call on spWMUpdate. It is part of the API if and only if V is a shared variable and does not have the C type spFixedAscii. No validity checks are performed. Just as with the external accessors, obtaining an old value often takes substantially more time than obtaining the corresponding current value.

In the interest of saving space in this document, the discussions of individual instance variables in shared objects merely list which accessors are available with reference to this section for greater details.

It should be noted that for a given class spK, the accessors above are available not just for the variables directly defined in the class, but also for the variables that are inherited. For example, if spK inherits a variable V from a superclass spJ and the accessor spJSetV is available, then the accessor spKSetV is also available. In the interest of brevity, these inherited accessors are not explicitly listed in this documentation.

1.5.2 Referring To

Instance variables that point to shared objects are handled specially in a number of ways. This is necessary due to the partial and asynchronous way that the world model is copied between processes.

A very important feature of the API is that the local world model copy in a given process does not contain everything in the entire virtual world, but rather only a subset of the objects that are of local interest. As discussed elsewhere, the determination of subsets is based primarily on locales (Section 26).

Because only some of the objects in the world model are in the local world model copy at a given moment, it is entirely possible that the local world model might contain an object A that refers to an object B (e.g., A's Parent might be B) and yet the local world model might not contain the object B.

Communication is arranged so that if two objects refer to each other, then in general they are both communicated in the same locale. As a result, the kind of inconsistency described above seldom exists for long. However, it is very common for this kind of inconsistency to exist briefly for a variety of reasons.

(1) When the focus of interest moves into a new locale, a process inevitably hears about some objects before others. (Because there are places where circularity of references is required, nothing can avoid the fact that this can cause temporary dangling references.)

(2) When new objects are created, a process inevitably hears about some objects before others.

(3) When objects are removed, a process may hear that an object is gone before hearing that other objects have stopped referring to it.

The above situations are handled by essentially putting the burden of a sanitized version of the problem on the application. When an instance variable contains a shared object, the access function that obtains the value (e.g., spGetParent) returns the object referred to only if the object is in the local world model copy. Otherwise it returns Null.

Suppose again that there is an object A in the local world model copy. If spGetParent returns non-Null when applied to A, then the return value is the Parent of A. However, if spGetParent returns Null, then this could mean either that A has no Parent or merely that the Parent of A does not yet exist in the local world model copy. (There is no way to tell the difference between these two things without inspecting details of A that are below the level of even the internal API.)

Suppose that A does indeed have a Parent B and suppose further that A appears in the local world model before B. When B later appears, this registers in all respects just the same as if the Parent of A changed from Null to B. In particular, if there is an alerter (Section 43) watching for such a change, it will be triggered.

The above approach works well because it does not matter to the typical application whether the Parent of A has changed or has merely just become known. An application has to be able to deal with arbitrary changes made by other processes, and doing this properly typically leads to a solution that operates properly in the presence of evolving partial knowledge as well. However, it is important to keep clearly in mind that a process's knowledge of the world model is always partial and therefore it is never justified to draw more than merely provisional conclusions from the absence of something in the world model, since anything can appear during a call on spWMUpdate.

1.5.3 Removal

A key issue is asynchronous changes in the world model. In particular, objects owned by other processes can appear, change, and disappear from the world model any time spWMUpdate is called. In addition, if there are multiple threads in the local process, then from the perspective of a given thread, other threads can asynchronously create, modify, and remove locally owned objects.

A particularly good example of the problems related to asynchronous changes is what must be done to properly handle the asynchronous removal of objects. To start with, it is important to understand what happens when an object is removed.

First, the IsRemoved bit is set on in the object R being removed and all connections between R and other objects are broken. Specifically, any instance variable in any other object that refers to R is set to Null. In addition, any instance variable of R that refers to any other shared object is set to Null. (One result of this is that having an object's Parent removed is treated very much the same as if the object's Parent was simply changed to Null.)

After the first stage of removal above, one can still consult all the instance variables of R, with the proviso that the variables that point to other objects have all become Null. (For variables that are shared with other processes, you can consult the old values of the instance variables to find out what they used to point to.)

Second, some time later (exactly how much time later is discussed in detail below) the storage corresponding to object R is freed. After that time, one can no longer access any of the instance variables of R, and it can be disastrous to try. A key purpose of the first step of removal is to ensure that no other shared object can retain a pointer to a freed object. Applications must be sure that they don't either. There is a separation in time between the first and second stages of removal so that applications have time to notice when objects have been removed.

To deal with asynchronous object removal, an application must check on a regular basis whether any objects it is interested in have been removed and respond appropriately before the objects are freed. The system has carefully designed rules about when objects can be removed and freed in order to reduce the number of times an application has to check for an object's continued existence.

Between calls on spWMUpdate, objects are never removed unless they are explicitly removed by the local process. The basic result of this is that it is sufficient for a process to check for the continued existence of an object it is keeping track of once each time after spWMUpdate returns. This checking is done using the accessor spGetIsRemoved. Once an object has been removed, the local process should drop all pointers to it immediately, because they will very soon be invalid. (A particularly good way to do the checking above is to use an spJustRemoved (Section 6.1) alerter on the object in question.)

Objects are never freed except during calls on spWMUpdate. In addition, if an object is removed after the beginning of a call on spWMUpdate it is not freed until the very end of the next call on spWMUpdate. This means that for every object, there will be at least one period between spWMUpdate calls where the object has been removed and not yet freed. This gives the application an opportunity to detect this fact and do something about it. (It also guarantees that the object stays around long enough for any spJustRemoved alerters to get triggered.)

As a result of the above, C applications need not worry much about objects being asynchronously freed as long as they check that the objects are still in the world model once after each call to spWMUpdate (e.g., with alerters).

In the Java interface, garbage collection and finalize methods ensure that shared objects are not freed as long as any pointers to them remain. In the C interface the function spWMRegister (Section 5.19) is used to obtain a somewhat similar level of protection. This can be used to make sure that the object pointed to from a given variable will never be freed and therefore it will always be valid to check whether the object has been removed.

An important special case is worthy of note. If the local process owns an object, then it can rely on the fact that the object will not get removed unless the local process removes it. This obviates the need for a significant amount of spGetIsRemoved checking in typical applications.

If the local process has multiple threads, then several problems arise. First, the other threads in the process can remove anything and so no object access can be considered entirely safe. Second, it is likely that some of the threads will not be synchronized with calls on spWMUpdate. It is difficult for such threads to know when they should check that objects still exist.

The most robust way to avoid problems with threads is to avoid having asynchronous threads manipulate shared objects directly, but rather have them communicate in some other way with the main thread in a process. Barring this, spJustRemoved alerters are the best way to ensure safety.

1.5.4 Communication Patterns

Shared objects are typically communicated to all (and only) the processes that are paying attention to the locales the objects are in. However, certain classes of objects are communicated in special ways.

spLocale objects themselves are communicated directly to Content- and Locale-Based Communication servers, and only via them to other processes. spClass objects, spBeaconing objects, and their subclasses are communicated via these special channels and using ordinary communication as well. spIntervalCallback objects (and subclasses) are not communicated to other processes. However, spBeaconMonitor objects (and subclasses) are communicated to Content-Based Communication servers as requests.

1.5.5 Descriptions

Information about the shared variables in an object are communicated between processes using messages containing object descriptions. There are two basic kinds of descriptions: full and differential. Differential descriptions specify only what has changed since the previous state or since any of several previous states.

An important question is exactly when full descriptions have to be used. In short, the answer is that a full description must be sent whenever in the normal course of events the recipient would not be able to understand a differential description. In particular, full descriptions must be sent in the following situations:

A - When an object is initially created, a full description must be sent.

B - Whenever an object changes locales, a full description must be sent to the new locale. The message to the old locale can use a differential description.

C - When servers send object descriptions, they must in general use full ones, because there are very few situations where they can validly assume that the recipient already knows about the object.

C1 - When a Locale-Based Communication server informs a process of the state of a locale, all the descriptions must be full ones.

C2 - When a Content-based Communication server sends information about beacons, it must send full information.

1.5.6 In-Order Description Processing

The descriptions received about an object are handled in a *mostly in-order* fashion. This is supported by the Msgs variable (Section 15.26) in shared objects and the MsgRejectionQueue (Section 5.13) world model variable.

When spWMUpdate runs, it first takes all incoming object descriptions, discards those that are blocked by the MsgRejectionQueue, and puts the rest on the appropriate object Msgs queues in proper order. descriptions with counters smaller than the Counter of the object and descriptions that are duplicates are ignored. Any objects with non-empty Msgs queues are then inspected for possible description processing.

When processing a Msgs queue, the following searching is done. A scan is made from lowest numbered description to highest to find the highest numbered description that can be processed without processing any other description first. If any such description it found, it is processed and it is removed from the queue along with any lower numbered descriptions. The search is then resumed to see if any other description can be processed.

The processibility of descriptions is determined as follows:

A - A differential description cannot be processed unless the prior state it depends on has already been reached.
B - No description can be processed unless it was sent by the process that owns the object as currently shown in the recipient's local world model copy.
C - Except for (B) which imposes some ordering, a full description is always processible.

It happens from time to time that a process gets a description about an object that is not yet known to it. If the description is differential, it is shunted aside for possible future consideration. If the description is a full description, it is processed and the object created.

The above scheme guarantees to always reject late arriving and duplicate descriptions because (1) ISTP prevents such descriptions cannot appear more than abounded number of msecs late and (2) there is always something to guard against them either in the world model or the MsgRejectionQueue for at least this long.

1.6 Defining Shared Classes

No matter what API language is being used, shared object classes are defined using Java and a preprocessor called SPOT. (The following discussion assumes a basic understanding of Java.) In general, SPOT takes in a Java file containing a shared class definition and produces:

1 - A new Java file in which instance variables are manipulated via access methods instead of directly as variables. This file is used to add the shared class to the Java API.
2 - A class descriptor file that is used by the system core when communicating instances of the class between processes.

In addition, SPOT is capable of producing the following outputs that allow a class to be accessible from C. These outputs are produced based on comments provided as part of the Java input.

3 - A Java stub file that links the methods in the Java class to a C implementation.
4 - A file of automatically generated C functions (along with an appropriate .h file) that adds the shared class to the C API.

SPOT operates in 2 basic modes: Java-only and dual.

In Java-only mode, SPOT operates purely to extend the Java API. The input need not contain any comments describing linkages to C. Things are arranged so that the class can be used from Java without having to link anything into the system core. Only outputs (1) and (2) are produced, outputs (3) and (4) being unnecessary.

In dual mode, and all four outputs are produced so that the class can be used to maximum effect in both C and Java. In Java-only mode, there is very little reason to apply SPOT to anything other than a shared class. However, this can be useful in mixed mode to make non-shared classes available in C.

The classes described in this document are defined solely using native methods and are processed using the dual mode of SPOT so that they are available fully in both the Java and C APIs. An appendix (Appendix A) contains the SPOT input corresponding to the API described in this document.

1.6.1 Example

The following example is used in the explanation below. It is contrived to illustrate many different features in a small space.

```
public class spExample extends spThing {
  public float[] Orientation;
  public spAction Agent;                    //* readonly
  public transient int Timeout;             //* internal
  public static float[] StandardOrientation;
  public static final DefaultTimeout = 1000;

  public spExample(int Timeout) {...};
  public final void Setup(float [] Orientation) {...};
  public void Print() {...};
}
```

Syntactically, the class definition above is standard Java. The information that is used by SPOT is indicated using standard Java keywords (e.g., transient) and data supplied in comments. comments are needed. (spThing and spAction are shared object classes.)

1.6.2 Shared Classes

A class is a shared class if and only if it is the root shared class sp or extends another shared class. (In the case above, spExample extends the shared class spThing. In contrast, the class spWM (Section 5) is not a shared class.) The key feature of a shared class is that the objects that are created as instances of the class are shared between the processes participating in a session.

An instance variable is shared if and only if (1) it is in a shared class, (2) it is not transient, and (3) it is not static. The transient keyword is used to explicitly state that a variable is not shared. Static (and therefore static final) variables cannot be shared. When a shared variable is set in one process, the change is automatically reflected in all other processes. In contrast, non-shared variables exist in each process, but the values of non-shared variables are set separately by each process, with no effect on their values in any other processes. In this document, non-shared variables are typically referred to as *local*.

1.6.3 Variables and Access Methods

For the most part classes processed by SPOT can contain any variable declarations they like. However, two things are important to note. First, SPOT generates a static final variable called C (Section 15.1) for each class is processes. At run time, this variable contains the spClass descriptor for the class. In order to avoid conflict, a class definition cannot contain a variable named C.

Second, SPOT generates instance variable access methods (Section 1.5.1) so that the instance variables of a class are accessed via methods rather than directly as variables. For instance, when using an instance X of the class spExample, one cannot write X.Agent anywhere.

The only exception to the above is that static final variables are treated as variables. For instance, when using an instance X of the class spExample, one can write X.DefaultTimeout.

The access methods created by SPOT (Section 1.5.1) are given the same access control keyword as the variable that led to them. In the API described here, this keyword is always public.

The semicolon ending an instance variable declaration in a class processed by SPOT can be followed by a comment beginning with '//*'. This comment can be used to control the names of the accessors created by SPOT. In particular, the comment can contain the keywords "readonly" and "internal". If the keyword "internal" is used then all accessors start with the letter 'i'. If the keyword "readonly" is used then the Set accessor begins with 'i'. For instance, the Agent instance variable of spExample has the accessors GetAgent, GetOldAgent, and iSetAgent. This indicates that reading the Agent variable is part of the external API, but setting it is not.

If access methods are generated for a static (but not final) variable, then the access methods are also static. The only class in the API that uses static (but not final) variables is the class spWM (Section 5).

SPOT does not generate access methods for static final variables. Rather, they just act as constants as in Java in general.

The code below illustrates the access-method-introduction transformations performed by SPOT. In Java-only mode, the methods introduced are defined solely in Java. In mixed mode, the methods introduced are native and accessible from both Java and C.

```
public class spExample extends spThing {
  public static final spClass C = ...;

  protected float[] Orientation;
  protected float[] OldOrientation;
  protected spAction Agent;
  protected spAction OldAgent;
  protected int Timeout;
  protected static float[] StandardOrientation;
  public static final DefaultTimeout = 1000;

  public spTransform GetOrientation() {...}
  public spTransform GetOldOrientation() {...}
  public void SetOrientation(float[] y) {...}

  public spAction GetAgent() {...};
  public spAction GetOldAgent() {...};
  public void iSetAgent(spAction x) {...};

  public int iGetTimeout();
  public void iSetTimeout(int Timeout) {...};

  public static float[] GetStandardOrientation() {...};
  public static void SetStandardOrientation(float[] y) {...};
  ...
}
```

Access methods are introduced in order to support the easy sharing of objects and other features of the API. In particular, for many kinds of data, special processing must be performed when values are read or written. SPOT inserts function calls in the access methods it creates that perform this processing. Even from methods defined within a class processed by SPOT, instance variables should never be referred to directly, but rather only via access methods.

Note that ISTP specifies the position in descriptions of a few key shared object fields. When generating information about how to construct messages and perhaps when deciding how fields are arranged in memory, SPOT has to pay head to this.

It is possible for a programmer to override an accessor method that would have been created by SPOT by providing his own definition of a method with the same name. In that case, SPOT will not generate the method in question, relying on the programmer's definition instead. This feature is used a couple of times as part of the definition of the system core. It would be very dangerous for an application programmer to use this feature, because it would be all to easy to omit essential processing that had to occur for the type of data stored in the variable.

1.6.4 Shared Variable Types

If a variable is shared, then the data stored in the variable must be one of only a few permitted types. These types fall into two groups: scalar types and pointer types.

A shared variable can have as its type any of Java's eight primitive scalar data types (byte, short, int, long, float, double, char, and boolean). In addition, a shared variable can have one of the following pointer types.

A - The variable can have as its type a shared class. These classes include the basic shared class hierarchy described in this document (i.e., spThing and everything else descended from the base shared object class sp) as well as any new shared classes that are defined.
B - The variable can have as its type the special class spFn.
C - The variable can have as its type String.
D - The variable can be an array whose elements are one of Java's eight primitive scalar data types.

You can define any kind of Java object you want and have these objects refer to shared objects, but this new Java object can be used as the value of a shared instance variable only if the new object is itself an instance of a shared class.

The total memory size of all the shared instance variables of a class must fit in a single UDP message (i.e., be less than 600 bytes or so). A warning is issued if this restriction is violated.

1.6.5 Methods In Shared Classes

The definition of a shared class processed by SPOT can contain methods defined in any way that is acceptable to Java. In general, these methods are not altered by SPOT and are not utilized by the system. (An application can use them in any way it likes.) However, application programmers should be aware that a few methods are handled in special ways by SPOT and/or the system core. (Note that for all these methods, there can be at most one method with the indicated name in a given shared class definition.)

Initialization - You can define a method called Initialization (Section 15.29) that initializes some or all of the variables in a class. This is similar to using initialization expressions except that an Initialization method can be used to initialize inherited variables in addition to variables that are defined as part of the class itself. If no Initialization method is provided, SPOT defines one that does nothing. The Initialization method is recorded in the spClass object for a class (or interface) so that it can be called by the system core when a new object is created.

Initialization methods are also similar to initializing variables in an object constructor method except for two things. First, When the system core creates an object instance it calls the Initialization methods for the object's class and every class it inherits from in order from the most general to the most specific. As a result, the Initialization method for a class can override the actions of the initialization methods for the classes it inherits from, but does not have to duplicate the actions it agrees with. This contrasts with object constructors which trigger more abstract constructors only if they call them explicitly.

Second, when an object is created due to the receipt of an object description from a remote process, the system core creates the object by calling spClassNewObj for the appropriate class which calls the initialize methods. In particular, the system core does not call Java object constructors that are written by application programmers. Therefore, initializations that need to always occur when remote objects appear need to be included in Initialization methods, not object constructors.

Object constructors - A class definition can contain the definition of one or more object constructors. SPOT modifies these by adding a call on spClassNewObj as the first line, if one is not already present. This properly initializes the system core when the constructor is called and calls the Initialization methods. If there is no object constructor specified, then SPOT creates one with no arguments that merely calls spClassNewObj.

The code below illustrates the transformations of methods performed by SPOT. In Java-only mode, the methods introduced are defined solely in Java. In mixed mode, the methods introduced are native and accessible from both Java and C.

```
public class spExample extends spThing {
  ...
  protected Initialization() {};
  public spExample(int Timeout) {...spExample.C.NewObj(); ...};
  public final void Setup(float [] Orientation) {...};
  public void Print() {...};
}
```

1.6.6 Interfaces

In Java there is a big distinction between interfaces, which must be almost completely abstract, but can be multiply inherited and ordinary classes which can be concrete, but cannot be multiply inherited. (Note interfaces in Java allow multiple inheritance from the perspective of variable types, but do not allow anything of concrete to actually be inherited; rather everything must be explicitly implemented in each class that implements an interface.) This is awkward because it makes it difficult to make full use of multiple inheritance in an API.

By means of appropriate preprocessing, SPOT reduces the difference between interfaces and classes, allowing interfaces for shared objects to be concrete. This allows the API presented here to make much fuller use of multiple inheritance.

SPOT allows everything that Java allows in an interface and does not modify any of that information. In addition, if an interface is a shared interface, SPOT allows the inclusion of method implementations and variables that are not static final.

An interface is a shared interface if and only if it is the root shared interface sp or extends another shared interface. A shared interface is allowed to contain instance variables that follow exactly the restrictions for variables in shared classes. However, like everything else in an interface, these variables are required to be public.

SPOT removes all the information about variables that are not both static and final, saving it for future reference, and replaces it by abstract signatures of the corresponding access methods.

Note that in Java the syntax of interfaces specifies that every variable is public, static, and final by default. In particular, even if these keywords are omitted, they are considered to be implicitly present. SPOT continues the requirement that every variable and method be public, but allows the omission of the keywords static and/or final to have meaning.

A shared interface can contain non-static implemented methods. These can be native or have bodies. SPOT saves the method implementations aside for future reference and converts them to abstract signatures.

As in ordinary classes, note that there can be an Initialization method that gives initial values for variables. If there isn't an Initialization method, SPOT generates an Initialization method that does nothing. Since an interface is, per force, abstract, it cannot have a constructor method.

From the perspective of the API, interfaces are the same as ordinary classes except that they are abstract. SPOT creates an spClass object for interfaces just like it does for shared classes. The spClass object specifies the Initialization method and Spline makes sure that this method is called at the right time.

As an example of the way SPOT handles interfaces consider that the following:

```
public interface spOrienting extends sp {
  public float[] Orientation;
  public transient spAction Agent;  //* readonly
  public void Setup(float [] Orientation) {...};
}
```

is transformed by SPOT into:

```
public interface spOrienting extends sp {
  public static final spClass C = ...;
  public float[] GetOrientation();
  public float[] GetOldOrientation();
  public void SetOrientation(float [] Orientation);
  public spAction GetAgent();
  public spAction GetOldAgent();
  public void iSetAgent(spAction agent);
  public void Setup(float [] Orientation);
}
```

The result is a standard Java interface that is entirely abstract. SPOT maintains a record of the information that was removed so that it can be used to cause proper inheritance when the interface is used.

As an example of how an interface class can be used in conjunction with SPOT, consider that the interface spOrienting could be used when defining spExample above as follows. In this definition, the variables Orientation and Agent, and the method Setup are inherited from spOrienting instead of being defined as part of the class itself.

```
public class spExample2 extends spThing implements spOrienting {
  public transient int Timeout;              //* internal
  public static float[] StandardOrientation;
  public static final DefaultTimeout = 1000;

  public spExample(int Timeout) {...};
  public void Print() {...};
}
```

SPOT allows the use of the "implements" keyword when defining shared classes, but only if the interface is also shared. (This is the only situation where SPOT does anything special with the "implements" keyword.) It is further required that when a shared interface is used, there never be any variable or method name clashes.

SPOT supports multiple inheritance of interfaces by simply leaving the keyword clause unchanged for the benefit of the Java compiler and then including macro-style all the information that was saved from the interface definition itself. (Note that this does not include Initialization methods, which are handled correctly via spClass objects.)

If the same interface is inherited via two paths, then only one copy is inserted. In general, if a variable or method is inherited twice, only one copy is retained and an error is reported if the two copies are not identical.

After methods specified by interfaces have been inserted into a class definition, then SPOT processing continues, with the creation of appropriate access methods, etc.

Note that even when doing all the above, SPOT has the critical feature that it only has to run on the definition of a class. The class is used in completely standard Java ways. This means that while SPOT has to be used to define new shared classes, it does not have to be used in conjunction with programs that merely use these classes.

For example, the when spot processes the definition of spExample2 above, it creates the following, which is identical to what is created when processing spExample.

```
public class spExample2 extends spThing {
  public static final spClass C = ...;

  protected float[] Orientation;
  protected float[] OldOrientation;
  protected spAction Agent;
  protected spAction OldAgent;
  protected int Timeout;
  protected static float[] StandardOrientation;
  public static final DefaultTimeout = 1000;

  public spTransform GetOrientation() {...}
  public spTransform GetOldOrientation() {...}
  public void SetOrientation(float[] y) {...}

  public spAction GetAgent() {...};
  public spAction GetOldAgent() {...};
  public void iSetAgent(spAction x) {...};

  public int iGetTimeout();
  public void iSetTimeout(int Timeout) {...};

  public static float[] GetStandardOrientation() {...};
  public static void SetStandardOrientation(float[] y) {...};

  protected Initialization() {};
  public spExample(int Timeout) {...spExample.C.NewObj(); ...};
  public final void Setup(float [] Orientation) {...};
  public void Print() {...};
}
```

1.6.7 Variables Available In C

When used in mixed mode, SPOT can make instance variables of classes and interfaces that are specified in the Java input accessible from C programs. In order for this to be possible, a number of restrictions must be obeyed and a C type has to be specified for the variable.

Variables that are made available in C must follow the restrictions on types presented above (Section 1.6.4) whether or not the variables are shared. Note that a variable can be shared without being available in C and a variable that is not shared can be available in C.

Variables that are made available in C cannot redefine any variable in the class being extended. (This is because, for efficiency, the system assumes that such instance variables cannot have their definitions changed.)

Variables that are made available in C, are not allowed to have an initialization expressions. Rather, one uses an Initialization method (Section 15.29). The only exception to this is that static final variables (Section 1.6.10) must have initializations.

The semicolon ending an instance variable declaration in a class processed by SPOT can be followed by a comment beginning with '//*' that specifies a type to use for the variable in the C. This specification has the form [type] where the type is a C type (defined separately in a C file). SPOT makes the variable available in C if, and only if, such a comment is specified.

The following shows how comments could be added to the example above to make several of the variables available in C. The following sections discuss the contents of these comments in detail.

```
public class spExample extends spThing {
  public float[] Orientation;                //* [spTransform:17]
  public spAction Agent;                     //* [sp] readonly
  public transient int Timeout;              //* [spDuration] internal
  public static float[] StandardOrientation;
  public static final DefaultTimeout = 1000; //* [spDuration]
  ...
}
```

A shared class can contain instance variables that are neither shared nor available in C. Such a variable can be specified in any way that is acceptable to Java.

1.6.8 Scalar C Types

The only restriction on the C type specified for a variable that has one of Java's eight basic scalar types is that the C type occupy the same amount of storage as the Java type. For instance, an int in Java is represented using 32 bits. Any 32-bit C type can be used in conjunction with it. The following table shows the basic correspondence between Java and C types. Many other types (e.g. user defined ones) can be used.

Scalar Java type	**Default C type**
boolean	spBoolean
byte	char
short	short
char	unsigned short
int	long
long	double
float	float
double	double

In general, SPOT does nothing special with individual scalar types. It merely needs to know what types to use in Java and in C and what their sizes are. However, one scalar type gets special treatment.

If the C type of a shared or native instance variable is spBoolean, then the value is stored in a single bit using special internal variables of the class sp (Section 15.12). However, once the bits reserved in these variables have been exhausted, then a whole byte is used for an spBoolean value.

Whenever they are passed as arguments to functions or passed between C and Java, scalar values are copied.

1.6.9 Pointer C Types

If a variable made available in C has an object class as its Java type, then the corresponding C type must be a pointer. The following table shows some typical mappings. Note that as discussed above, there are only a few Java classes that can be used as the types of variables made available in C.

Permitted Java type - C type
shared object - sp
spFn - spFn
String - char *
arrays - scalarType *

In C, all shared objects are referred to by pointers of type sp and no other C type can be specified. All functional arguments are referred to by pointers of type spFn and no other C type can be specified.

For strings and arrays, the C type must be a pointer. In addition, for these types, the comment specifying the C type must also specify the memory size of the data pointed to. This is done by using a comment of the form [type:size] where type must be a C type that is a pointer (e.g., float*) and where size specifies how many elements there are in the string or array. For instance, in the spExample class, the C type spTransform points to a vector of 17 floats.

When pointer types are passed as arguments to functions, they are passed whenever possible by reference via a pointer. However, when pointer types are passed between C and Java, the data is typically copied so that appropriate data structures can be maintained separately in C and Java.

As with scalar types, SPOT typically does nothing special with individual pointer types. It merely needs to know what types to use in Java and in C and what the memory size in C is. (For ease of allocation and communication, strings and arrays are stored in-line in the C representation for an object.) However, several pointer types get special treatment.

The shared object types and the type spFn get special treatment so that appropriate objects will be available efficiently in both Java and C. In addition, Strings get special treatment.

A difference between Java and C is that Strings in Java use Unicode characters while strings in C use ASCII characters. To accommodate this, Java strings are encoded as null terminated UTF8 strings when communicated to C. This means that if all the characters in a Java string are merely ASCII characters, then there is a trivial one-to-one correspondence between the Java and C strings. If the Java string has special characters in it, then the C string has multi-character codes embedded in it signaled by special prefix characters.

The C type spFixedAscii specifies a string of variable length that can only be set at the moment when a shared object is being created and cannot be changed later. To minimize the memory used for spFixedAscii values, they are placed in a special variable-sized part of an object. (A consequence of this is that it is not possible to obtain the old value of an spFixedAscii variable.)

The type spFixedAscii can only be used for shared variables. Unlike other array-like types, no explicit size can be specified when using the type spFixedAscii.

As noted above, the total size (in C) of all the shared instance variables must fit in a single UDP message (i.e., be less than 600 bytes or so).

1.6.10 Static Final Variables

A variable made available in C has an initializer expression if and only if it is a static final variable. If the initialization of the value in C needs to be different from the initialization in Java (e.g., because it is not just a numeric constant) then the C initialization can be specified after an '=' sign in the //* comment that specifies the C type of the variable. For example the definition of spDEGREES (Section 15.2) could be:

```
native static final public float DEGREES = Math.PI/180.0; //* [float=M_PI/180.0]
```

1.6.11 Methods Available In C

When used in mixed mode, SPOT can make methods of classes and interfaces that are specified in the Java input simultaneously accessible from Java and C programs. In order for this to be possible, a number of restrictions must be obeyed and a C type signature has to be specified for the method.

A method that is made available in C API must be 'native'. That is to say, the application writer has to do the hard part of making the method available in C. SPOT merely creates proper stubs and interfaces.

A particular complexity here is that Java does not allow object constructors to be 'native'. Therefore, special effort has to be taken to define a constructor that is available in both the C and Java APIs. Specifically, a separate method has to be defined that contains everything you want in the constructor, and that can be native. By convention, this is done by defining a method called New. If there is only one method named New and no construction method then, SPOT automatically generates a constructor with the same arguments as the New method that calls the New method. It is expected that only programmers that are extending the system core will define constructors that are available in both the C and Java APIs.

When the above conditions are satisfied, the specification of a native method can be followed by a //* comment containing the signature of the C function that implements the method. When such a comment is present, SPOT generates appropriate C .h files and stub files linking the Java and C API's.

The following shows how the example class could be altered to make the constructor and the method Setup available in C.

```
public class spExample extends spThing {
  ...
  native public static int New(int Timeout);
   //* [sp spExampleNew(spDuration Timeout)]}
  native public final void Setup(float [] Orientation);
   //* [void spExampleSetup(sp ExampleObj, spTransform:17 Orientation)]
  public void Print() {...};
```

To make it possible to create stubs correctly, types in the C function signature must obey all the restrictions presented above for C types of variables made available in C (Section 1.6.4). In particular, C types that correspond to arrays must contain a specification of their sizes. In general, lengths must also be included for string types. However, they can be omitted if the string is null terminated.

For the generation of stubs, SPOT uses the names of the arguments to determine the correspondence between the arguments of the Java and C signatures. The Java and C type of an argument of a native method must be one of the types that are valid for native variables. The corresponding C type of the argument must be one of the C types that are permitted to correspond to the Java type. Automatic conversions are performed when passing values between Java and C.

If a method is not a static method, then the object the method is applied to is passed to the C function via the first argument whose name is not the same as the name of any argument in the Java method. Otherwise, if an argument appears only in the C signature, it is passed the value 0. If an argument appears only in the Java signature it is not be passed to the C function.

In the standard API classes, the convention is followed that if a class spK has a method M, then the name of the corresponding C function is spKM, the arguments are in the same order, and the argument to the C function that receives the object the method is applied to is the first argument.

1.7 Acknowledgments

The design and implementation of Spline is the result of a four-year effort by a large group of people. Richard Waters and David Anderson have been the principle architects of Spline from its earliest inception. In addition, they were the principal implementers of the initial version of Spline (Spline 1.5) and led the reimplemention effort that created Spline 3.0.

Major contributions to the design and implementation of Spline were made by John Barrus. In particular, he participated in the design from the very beginning, was co-designer of locales, designed and implemented the concept of terrain's, and implemented Spline 1.5's first visual renderer.

Several other people helped with the implementation of Spline 1.5. In particular, Joe Marks assisted with the initial design and wrote the first lines of code. David Marmor did early exploratory work on audio processing. Michael Casey was co-designer and co-implementor of Spline 1.5's audio renderer. William Yerazunis was co-designer and implementor of the smooth motion operations.

The design and implementation of Spline 1.5 proceeded in tandem with the design and implementation of Diamond Park, which was the most significant application built on top of this early version of Spline. The Diamond Park effort was lead by Barrus working closely with Stephan McKeown and Ilene Sterns. Additional major contributions were made by Anderson, David Brogan, Casey, Jessica Hodgins, Waters, Yerazunis, Altitude inc., and Boston Dynamics inc.

A large team of people participated in the implementation of Spline 3.0 under the leadership of Waters and Anderson, who were principally responsible for the design. William Lambert took over leadership of the overall Spline effort at the end of 1996. William Yerazunis managed the software engineering aspects of the project and implemented smooth motion operations. Derek Schwenke implemented the system core including much of ISTP and the first visual renderer. He took over management of the software engineering effort in mid 1997. Sam Shipman implemented the Java interface for defining shared classes (SPOT) and the audio renderer. Alex Greysukh implemented the underlying network communication code building on the work of Hiroshi Kozuka and Vu Phan. Rob Kooper and David Ratajczak implemented the routines for manipulating transformations. Christina Fyock, Evan Suits, and Barry Perlman created the first Spline 3.0 applications.

2 spApp

```
public class spApp
```

In Java, simple applications are subclasses of the class spApp. In C, simple applications are collections of function definitions rather than a class definition with specialized methods. Nevertheless, the same basic approach is taken. In particular, the documentation below describes how to write applications from the perspective of a group of functions that comprise a simple application.

The class spApp defines the following functions:

spAppChooseServer - Chooses session server (Section 2.5).
spAppInit - Initializes application (Section 2.6).
spAppBody - Body of application (Section 2.7).
spAppFinish - Cleans up at end of application (Section 2.8).

The way the functions above are used in a simple application can be seen by looking at the standard application skeletons provided.

2.1 Application Templates

A number of templates are provided for main programs that can be used for applications. The simplest of these is shown below. More complex templates allow for the combination of rendering with an application (Section 2.2) or the creation of an application that is a Netscape plugin.

```
void main(int argc, char **argv) {
  spWMNew(spAppChooseServer(), NULL);
  spAppInit();
  while (~ spAppBody()) spWMUpdate();
  spAppFinish();
  spWMRemove();
  exit(0);
}
```

The application code is integrated on the lines containing the calls on spAppChooseServer, spAppInit, spAppBody and spAppFinish. These calls decide what user server to connect to, initialize the application, perform the main application computation, and clean up at the end of the application. Before going into any more detail about what these functions do, it is instructive to see how the template above uses them.

spAppChooseServer is called first to determine what user server to connect to, if any. The result is then used when creating the world model. spAppInit is called to set things up for the application. The main body of the template is a loop that repetitively calls spAppBody and spWMUpdate. If spAppBody ever returns True, then the loop terminates. To end the application, the template calls spAppFinish and spWMRemove.

There is only one call on spWMUpdate in the template. No API function contains within itself a call on spWMUpdate. It is a general principle that all calls on spWMUpdate should be clearly visible—at best at top level as in the template above.

The goal of the templates provided is to make it very easy to write very simple applications. When implementing complex applications, it is expected that programmers will modify the templates, or create entirely new ones.

2.2 Interaction With Rendering

Stand-alone applications spVisual, spAudio, and spAudioVisual are provided that do visual rendering, audio rendering and both audio and visual rendering respectively. Under the control of spSpeaking, spHearing, and spSeeing beacons, these applications can be used to create visual and audio output in conjunction with any application.

Having visual and audio rendering be separate from an application has two main advantages. It allows an application to run (potentially on a separate machine) without any regard for the update rate constraints caused by visual and audio rendering. In addition, it allows a simulation application that does not require visual or audio rendering to run without incurring any rendering overhead.

The above notwithstanding, it is often advantageous to combine visual and audio rendering into an application process. There are two main reasons for this. First, if the application is going to run on the same machine with the renderers, then there is less total overhead if they all run in the same process. In particular, there is then only one copy of the world model instead of several. Second, if the application wants to do keyboard and or mouse I/O with the window used for visual rendering, then visual rendering must be included as part of the application (or at least included with the part of the application that does the I/O).

In order for rendering to be combined with an application, the application must meet two key criteria. First, the application must be relatively insensitive to the rate at which spWMUpdate is called. The reason for this is that for visual rendering to work, visual rendering must closely control the rate at which spWMUpdate is called. Second, the maximum amount of time the application runs when it gets control must be relatively short. The reason for this is that if the application ever runs for a long time, visual output will be frozen for this entire time.

While it is beneficial that an application not take very much CPU time in total, combining an application with rendering does not require this. The reason for this is that whatever amount of time the application takes, it will take, whether or not the application is combined with rendering. If the application takes a huge amount of time, it must be moved to a separate machine. Simply separating the application from rendering on a single machine only makes things worse by increasing overhead.

The following template shows how audio and visual rendering should be combined with an application. If only visual rendering is wanted, then the calls on spAudioInit and spAudioFinish can be omitted. If only audio rendering is wanted, then the calls on spVisualInit and spVisualFinish can be omitted.

```
void main(int argc, char **argv) {
  spWMNew(spAppChooseServer(), NULL);
  spAudioInit();
  spVisualInit();
  spAppInit();
  while (~ spAppBody()) spWMUpdate();
  spAudioFinish();
  spVisualFinish();
  spAppFinish();
  spWMRemove();
  exit(0);
}
```

The template above is very similar to the template used when rendering is not combined with an application (Section 2.1). The only differences are that spAudioInit and spVisualInit are called to get things started and spVisualInit supplies the window to use for interaction with the user.

Neither spVisualInit nor spAudioInit call spWMUpdate. What is more, they both assume that spWMUpdate is not called before they are. In general, spAppInit is free to assume the same. In addition, spAppInit can assume that it will be called after all other initialization is complete.

2.3 User I/O

The API does not provide any functions for keyboard input, textual output to a window, or mouse interaction. Rather, it is expected that an application will use whatever native facilities are available in the windowing environment being used (e.g., X or Windows95). If you want to write a portable application, then you should use a portable API such as Java and use the portable I/O facilities that are provided by the API language.

It is often the case that an application wants to do I/O via the window that is being used to display visual images. To do this, you should combine visual rendering into the application process (Section 2.2) and use the window created by spVisualInit.

If the application is such that it cannot be practically combined with visual rendering, then the application should be broken into two parts: a small part that does user I/O and a large part that does other operations. The user I/O part should then be combined with visual rendering. It may be possible to completely handle whatever needs to be done in response to user input in this part. If not, then the user I/O part can communicate information to the other part of the application through shared variables, or by some special side-communication channel.

2.4 Multi-Threaded Applications

It can be convenient to implement an application using multiple threads. However, lack of synchronization between these threads can lead to serious problems. In particular, if one thread calls spWMUpdate, then the other threads must obey all the same conditions as the thread calling spWMUpdate. In particular, they must not retain any pointers to shared objects or data returned by API functions. They must be sure to check for objects that are removed every time following a call on spWMUpdate. In addition, they cannot call API functions that operate on the world model until after the call on spWMUpdate in the other thread returns.

As a result of the above, a thread must either be very carefully synchronized with any thread that calls spWMUpdate, or not have any direct interaction with the world model at all. We strongly urge the second approach.

Basically, it is expected that only one thread is interacting with the world model. If there are other threads, then they should interact with the main thread using global variables in the application and should not call API functions themselves. In addition to avoiding the problems enumerated above, this avoids other problems stemming from asynchrony like multiple readers and writers of shared objects.

2.5 spAppChooseServer

```
char * spAppChooseServer()
```

Return value - DNS address string of session manager to connect to.

The spAppChooseServer function selects the user server to connect to, if any. It must return a DNS address string indicating the server to connect to (e.g., `"node.myU.edu"`). The default definition of this function returns Null. This indicates that no user server should be used. This is appropriate for any process that has a good enough network connection to operate as full ISTP peer. If a user server is necessary it is expected that an application will replace spAppChooseServer with some kind of interactive interface that allows the user to choose which user server to connect to.

A key restriction on spAppChooseServer is that it cannot use any operations on shared objects, because it is called before the world model is created.

2.6 spAppInit

```
void spAppInit()
```

Return value - There is no return value.

The purpose of the spAppInit function for a simple application is to do various things that only need to be done once before the application starts. For example, it might initialize various global values and set up callbacks. The default definition of this function does nothing.

spAppInit should use the value returned by spWMGetWindow, if any for interacting with the user. If there is no preexisting window, then the application is free to create whatever it wants to. One situation where a window is supplied is when an application is combined with visual rendering (Section 2.2).

2.7 spAppBody

```
spBoolean spAppBody()
```

Return value - True indicates application should be terminated.

The purpose of the spAppBody function for a simple application is to perform whatever repetitive computation is required once the application is up and running. For example, it might react to events that occur in the world model and/or interact with the user. The default definition of this function does nothing. This is an important degenerate case, because if spAppInit sets up enough callbacks (Section 42) and alerters (Section 43), it may not be necessary for spAppBody to do anything.

Each time spAppBody is called, the world model is updated (by calling spWMUpdate) to reflect changes caused by other processes. Callback functions (if any) are called just before spAppBody. Each time after spAppBody is called, messages are sent out communicating changes made by the application to other processes (by the next call on spWMUpdate). It is expected that spAppBody will not call spWMUpdate.

If spAppBody ever returns True, this signals that the application and the process should be brought to a graceful conclusion without spAppBody ever being called again.

2.8 spAppFinish

```
void spAppFinish()
```

Return value - There is no return value.

The purpose of the spAppFinish function for a simple application is to perform any finalization activities that are necessary in order for the application to be terminated gracefully. The default definition of this function does nothing.

3 spVisual

```
public class spVisual extends spApp
```

spVisual is the standard visual renderer supplied with the system. To facilitate its combination with application processes (Section 2.2), spVisual is organized in the same way as spApp.

The class spVisual inherits all the instance variables and functions of the class spApp (Section 2). The class spVisual defines the following functions:

spVisualInit - Initializes visual rendering (Section 3.1).
spVisualFinish - Cleans up after visual rendering (Section 3.2).

spVisual utilizes the standard definition of spAppChooseServer and therefore does not connect to any user server. When combined with an application, the application selects the user server, if any.

A key feature of spVisual is that it utilizes the same empty spAppBody function as the default simple application. This makes it much easier to combine spVisual with an application. This approach is possible because spVisual performs all its computation using callbacks and asynchronous threads set up by spVisualInit.

It is interesting to note that since spVisual is intended to be combined with a user application, it must be careful to never remove any objects it does not own. If it did remove such an object, this effect would be seen by the user process it was combined with.

3.1 spVisualInit

```
void spVisualInit()
```

Return value - There is no return value.

Performs appropriate initialization so that visual rendering can occur as part of a process. If spWMGetWindow returns non-Null, then the result of visual rendering is shown in that window. Otherwise, a new window is created and stored using spWMSetWindow. An application that is combined with spVisual should use the window used by spVisual for interaction with the user.

Visual rendering operates in two parts. A set of callbacks detects what needs to be rendered and creates a scene graph based on the relevant objects in the world model. An asynchronous thread does the actual image generation. In order to achieve the frame rate requested of it (through an spSeeing beacon), spVisual must take control of the rate that spWMUpdate is called. Therefore, an application that is combined with spVisual cannot exercise much control over the rate at which spWMUpdate is called.

spVisualInit makes use of a special local value of spWMGetMe. However, it insures that spWMGetMe is set back to spWMGetMainOwner before spVisualInit returns. spVisualInit does not call spWMUpdate and assumes that spWMUpdate will not be called before spVisualInit is called.

3.2 spVisualFinish

```
void spVisualFinish()
```

Return value - There is no return value.

Turns off visual processing and performs cleanup operations leaving things in a good state. spVisualFinish makes use of a special local value of spWMGetMe. However, it insures that spWMGetMe is set back to spWMGetMainOwner before spVisualFinish returns.

4 spAudio

```
public class spAudio extends spApp
```

spAudio is the standard audio renderer used with the system. To facilitate its combination with application processes (Section 2.2), spAudio is organized in the same way as spApp.

The class spAudio inherits all the instance variables and functions of the class spApp (Section 2). The class spAudio defines the following functions:

spAudioInit - Initializes audio rendering (Section 4.1).
spAudioFinish - Cleans up after audio processing (Section 4.2).

spAudio utilizes the standard definition of spAppChooseServer and therefore does not connect to any user server. When combined with an application, the application selects the user server, if any.

A key feature of spAudio is that it utilizes the same empty spAppBody function as the default simple application. This makes it much easier to combine spAudio with an application. This approach is possible because spAudio performs all its computation using callbacks and asynchronous threads set up by spAudioInit.

4.1 spAudioInit

```
void spAudioInit()
```

Return value - There is no return value.

Performs appropriate initialization so that audio I/O using the microphone and headphones can occur.

Audio rendering operates in two parts. A set of callbacks detects what needs to be rendered and what localization parameters should be used. An asynchronous thread does the actual sound calculations. For the callbacks to operate correctly, spWMUpdate must be called several times a second. For the asynchronous threads to operate correctly, the application process must not take exclusive control of the CPU for more than tens of milliseconds at a time.

spAudioInit makes use of a special local value of spWMGetMe. However, it insures that spWMGetMe is set back to spWMGetMainOwner before spAudioInit returns. spAudioInit does not call spWMUpdate and assumes that spWMUpdate will not be called before spAudioInit is.

4.2 spAudioFinish

```
void spAudioFinish()
```

Return value - There is no return value.

Turns off audio processing and performs cleanup operations leaving things in a good state. spAudioFinish makes use of a special local value of spWMGetMe. However, it insures that spWMGetMe is set back to spWMGetMainOwner before spAudioFinish returns.

5 spWM (Fundamental)

```
public class spWM
```

The central data structure in the API is the world model. There can be only one world model object at a time. It contains the various shared objects that are communicated among a group of processes. The spWM type does not extend the class sp and does not correspond to an object in the world model.

To start up a process, the first thing one does is create the world model to operate on. When the process wishes to terminate, it should remove the world model. In between, the world model is repetitively updated to reflect changes in the objects in the world model caused by other processes.

The class spWM defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

Me - Owner id of current activity (Section 5.1).
MainOwner - Main owner id of process (Section 5.2).
SystemOwner - System owner id of process (Section 5.3).
Error - Most recent error (Section 5.4).
LastError - Last error (Section 5.5).
Interval - Interval between last two updates (Section 5.6).
DesiredInterval - Desired update interval (Section 5.7).
Week - The current week (Section 5.8).
Msec - The current millisecond (Section 5.9).
Window - Interaction Window (Section 5.10).
DNSName - DNS name of local machine (Section 5.11).
Port - Port number associated with process (Section 5.12).
MsgRejectionQueue - Objects for which messages must be rejected (Section 5.13).

The class spWM defines the following functions:

spWMNew - Prepares to use world model (Section 5.14).
spWMRemove - Prepares to stop process (Section 5.15).
spWMUpdate - Gets new world model state (Section 5.16).
spWMGenerateOwner - Creates a new owner id (Section 5.17).
spWMReportError - Creates error report (Section 5.18).
spWMRegister - Assures safe access through pointer (Section 5.19).
spWMDeregister - Cancels registration of pointer (Section 5.20).

5.1 Me (Fundamental)

```
public static int Me; //* [spName]

      spName spWMGetMe()
      void spWMSetMe(spName X)

      spName sqWMGetMe()
      void sqWMSetMe(spName X)
```

Owners are identified by 32-bit ids of type spName. The *Me* variable of the spWM object contains the owner id of the activity currently in control.

When the world model is created, the Me value is set to a main owner id assigned by the system. It can be changed at will later. In general any activity that changes the value of Me is responsible for restoring the value of Me once it completes operation. Additional owner ids can be obtained using spWMGenerateOwner (Section 5.17).

5.2 MainOwner (Fundamental)

```
public static int MainOwner; //* [spName] readonly

      spName spWMGetMainOwner()

      void spWMiSetMainOwner(spName X)

      spName sqWMGetMainOwner()
      void sqWMSetMainOwner(spName X)
```

The *MainOwner* variable of the spWM object contains the owner id initially assigned to the local process. It is set when the world model is initially created and cannot be altered later.

5.3 SystemOwner (Fundamental and Internal)

```
public static int SystemOwner; //* [spName] internal

      spName spWMiGetSystemOwner()
      void spWMiSetSystemOwner(spName X)

      spName sqWMGetSystemOwner()
      void sqWMSetSystemOwner(spName X)
```

The *SystemOwner* variable of the spWM object contains the owner id created by the system core running in the local process for its internal use. The SystemOwner is set when the world model is initially created and cannot be altered later.

5.4 Error

```
public static String Error; //* [char *:500]

      char * spWMGetError()
      void spWMSetError(char * X)

      char * sqWMGetError()
      void sqWMSetError(char * X)
```

The *Error* instance variable of an spWM object records the most recent error to have occurred during the evaluation of any API function. When the world model is created, the Error value is set to Null. It is changed whenever an error occurs. You can set the Error value back to Null if you want to. However, it cannot be directly set to any other value, but rather only indirectly via spWMReportError (Section 5.18). Error strings are limited to being no more than 500 characters long.

5.5 LastError

```
public static String LastError; //* [char *:500] readonly

      char * spWMGetLastError()

      void spWMiSetLastError(char * X)

      char * sqWMGetLastError()
      void sqWMSetLastError(char * X)
```

The *LastError* instance variable of an spWM object is identical to the Error instance variable except that it cannot be directly modified by an application, but rather only indirectly via spWM-ReportError (Section 5.18). The LastError value can always be consulted to determine what the most recent error, if any, was.

A key reason for having two variables Error and LastError is to facilitate debugging by making sure that a program cannot accidentally remove all trace of errors having occurred.

5.6 Interval (Fundamental)

```
public static int Interval; //* [spDuration] readonly

      spDuration spWMGetInterval()

      void spWMiSetInterval(spDuration X)

      spDuration sqWMGetInterval()
      void sqWMSetInterval(spDuration X)
```

The *Interval* instance variable of a world model records the time in milliseconds between the ends of the last two calls on spWMUpdate. When the world model is created, the interval is set to zero. After the end of the second and subsequent calls on spWMUpdate, the interval value is updated to reflect the timing of events. It must not be modified by an application.

To be precise, the interval value is updated just before action processing begins and reflects the interval in time between corresponding points in spWMUpdate calls. Note particularly, that the interval is calculated after any waiting that spWMUpdate does in order to achieve the DesiredInterval.

5.7 DesiredInterval (Fundamental)

```
public static int DesiredInterval; //* [spDuration]

      spDuration spWMGetDesiredInterval()
      void spWMSetDesiredInterval(spDuration X)

      spDuration sqWMGetDesiredInterval()
      void sqWMSetDesiredInterval(spDuration X)
```

The *DesiredInterval* instance variable of a world model controls the interval between calls on spWMUpdate as follows. When spWMUpdate is just about to begin processing actions, it determines the elapsed time since the last time actions were processed. If this time is less than the DesiredInterval, then spWMUpdate waits until the interval is reached. If the elapsed time is greater than or equal to the DesiredInterval, spWMUpdate proceeds without waiting. There is nothing that spWMUpdate can do to make processing take less time than it is taking. However, spWMUpdate ensures that the actual interval will not be less than the DesiredInterval. Since there is a good deal of imprecision in the system's timing mechanisms, the system cannot guarantee that the actual interval will be equal to the DesiredInterval; however, as long as the DesiredInterval is long enough to be achievable, the system guarantees that the average error over time will be low.

It is permissible for the DesiredInterval to be set to a negative value. This has the same meaning as the corresponding positive value except that when spWMUpdate considers waiting it waits only so long as no outside information has been received that modifies the world model. For example, a DesiredInterval of -1000 specifies that processing should proceed as soon as any new information is received, but in any event should proceed after a second has passed.

When the world model is initially created, the DesiredInterval is set to zero. This causes spWMUpdate to run as fast as possible without ever waiting. You can change the DesiredInterval at any time.

It is important to think carefully about how often spWMUpdate gets executed. If it executes too often, time is wasted looking at data that has not changed in any useful way. Alternatively, If spWMUpdate executes infrequently, you will be operating on very stale data much of the time. Further, if spWMUpdate runs very infrequently (e.g., less than once a second or so) some of the information about world model changes will probably be lost as queues and buffers overflow. (Things are arranged, however, so that even if you never call spWMUpdate, the system will not crash.)

5.8 Week

```
public static int Week; //* [long] readonly

      long spWMGetWeek()

      void spWMiSetWeek(long X)

      long sqWMGetWeek()
      void sqWMSetWeek(long X)
```

The *Week* instance variable of a world model is an integer specifying how many weeks have passed since midnight on a Sunday an arbitrary time in the past. The Week value is initialized when the world model is created, updated (when needed) when spWMUpdate is called and never changed otherwise.

5.9 Msec

```
public static int Msec; //* [spDuration] readonly

      spDuration spWMGetMsec()

      void spWMiSetMsec(spDuration X)

      spDuration sqWMGetMsec()
      void sqWMSetMsec(spDuration X)
```

The *Msec* instance variable of a world model is an integer specifying how many milliseconds have elapsed since the beginning of the current week. The Msec value is initialized when the world model is created, updated every time spWMUpdate is called and never changed otherwise. It corresponds to the time in spWMUpdate just before action processing begins, and after any wait that had to be performed by spWMUpdate.

If you are writing an application that you wish to have work over Sunday nights, you have to be careful about the fact that the Msec value is a modular number between 0 and 604,800,000 (one week in milliseconds). When going from Sunday to Monday, Msec changes from large to small.

5.10 Window (Fundamental)

```
public static int Window; //* [spWindow]

      spWindow spWMGetWindow()
      void spWMSetWindow(spWindow X)

      spWindow sqWMGetWindow()
      void sqWMSetWindow(spWindow X)
```

The *Window* instance variable contains the current user interaction window. The exact type of the window object and the operations that can be applied to it depend on the operating environment. In Java they are one thing and in C under Windows95 they are another.

When the world model is first created, the Window is set to Null. The surrounding environment may force a particular value to be used subsequently. For instance, this is the case when using the system as a Netscape plugin. If spVisualInit is called when the Window value is Null, then spVisualInit creates a window to use and stores it in the Window variable. Otherwise, the application is free to create or choose a window to use.

5.11 DNSName (Fundamental)

```
public static String DNSName; //* [char *:500] readonly

      char * spWMGetDNSName()

      void spWMiSetDNSName(char * X)

      char * sqWMGetDNSName()
      void sqWMSetDNSName(char * X)
```

The *DNSName* instance variable of a world model is a string containing the DNS name of the local machine, e.g., `"node.myUniv.edu"`. Together with the Port variable, the DNSName variable can be used to uniquely identify an individual ISTP process.

The DNSName variable is set when a world model is created and cannot be altered later.

5.12 Port (Fundamental)

```
public static short Port; //* [short] readonly

      short spWMGetPort()

      void spWMiSetPort(short X)

      short sqWMGetPort()
      void sqWMSetPort(short X)
```

The *Port* instance variable of a world model is an integer specifying the port that is used for TCP communication with the process. This port is either 80, indicating that a process is the main ISTP process running on a given machine or some other randomly selected value. The Port variable is set when a world model is created and cannot be altered later.

When an ISTP process is started up, it attempts to use port 80. However, if it is in use, then some other random port number is selected. One consequence of this is that if several processes are to run on a single machine, the process (if any) that is going to act as a server must be run first.

5.13 MsgRejectionQueue (Fundamental and Internal)

```
public static int MsgRejectionQueue; //* [void *] internal

      void * spWMiGetMsgRejectionQueue()
      void spWMiSetMsgRejectionQueue(void * X)

      void * sqWMGetMsgRejectionQueue()
      void sqWMSetMsgRejectionQueue(void * X)
```

The *MsgRejectionQueue* instance variable of a world model contains the information necessary to reject out-of-order messages about objects that have been removed from the world model. The key problem that is being dealt with is receiving (out of order) a message about the state of an object after previously receiving a message that specifies that the object has been removed. Unless something is done, this would cause the object to erroneously reappear. The MsgRejectionQueue is maintained by the system core and must not be modified by an application.

The MsgRejectionQueue is a queue with entries containing the following information:

(spName) Name - Name of object.
(int) Counter - Counter of object.
(spName) Locale - Locale of object.
(spTimeStamp) Time - Time entry was created.

Whenever an object is entered on the MsgRejectionQueue, an entry is made containing its Name, current Counter value, and Locale as well as the current time. Whenever a description is received for an object in the MsgRejectionQueue, the Counter on the description is checked, and the description is rejected if the Counter value is smaller than the value saved in the queue.

The entries are ordered by time and popped off the queue as soon as they are older than the maximum message delay allowed by ISTP. Entries are placed on the MsgRejectionQueue whenever an object (1) has its IsRemoved bit set or (2) changes locale, in which case the entry contains the old locale.

Case 2 is needed among other things by a process that is providing Locale Based Communication service so that it can know what recent changes to inform other processes of. The entry contains the old locale. It is possible for there to be several entries for the same object; for example if the object moves quickly through several locales.

5.14 spWMNew (Fundamental)

```
spWM spWMNew(char * Server, spTransferVector V)
```

Server - DNS name of minimal communication server or Null.
V - Transfer vector specifying functions to use for key operations.
Return value - The world model created.

Creates the world model. A process can only create one world model at a time. A process must create the world model before using any shared object operation. After initialization, the world model contains nothing but definitions of the built-in shared classes and a few internal objects that are not observable by applications.

If the process wishes to be supported by an ISTP server that minimizes the bandwidth used to communicate with the process, then the process uses the Server argument to specify a DNS string naming the server (e.g., `"node.myUniv.edu"`). If the process wishes to operate as a stand-alone ISTP node, then either a zero length string, or the value Null is used as the Server value.

The transfer vector is used to specify key internal operations (e.g., allocating memory) that may have special definitions that are required by the surrounding environment. In simple applications, Null is an acceptable value for this argument. You can consult the detailed definition of the spTransferVector type in order to determine how to create a non-Null transfer vector.

5.15 spWMRemove (Fundamental)

```
void spWMRemove()
```

Return value - There is no return value.

Eliminates the world model and disconnects the process from the session it is in. Among other things, this causes all of the objects owned by the process to be removed from the world model and ensures that all the other processes in the session are informed of this fact. A process should always call spWMRemove before terminating. Once spWMRemove has been called, a process cannot use any shared object operation unless it first creates a new world model.

5.16 spWMUpdate (Fundamental)

```
void spWMUpdate()
```

Return value - There is no return value.

Causes the local world model copy to be updated to reflect changes made by other processes. The fundamental operation of an application process (Section 2.1) is based on a cycle of: updating the world model (based on information received from other processes); running the application, waiting until the desired update interval has been reached, updating the world model again, and so on.

A number of important things happen each time spWMUpdate is called:

(A) Other processes are notified of changes made in the local world model copy only when spWMUpdate is called. This gives the application control over when other processes see changes. (For instance, one can ensure that several instance variables of an object will change simultaneously).

(B) The world model is brought up to date by processing messages from other processes. It is guaranteed that the world model will not change in any way between calls to spWMUpdate, unless the application makes the changes itself.

(C) Any actions (Section 46), callbacks (Section 42), and alerters (Section 43) in the world model are run. (This is the only time they are run.)

(D) As part of updating the world model, some objects may be removed from the world model.

Specifically, spWMUpdate performs the following operations in the following order.

1) Run alerters on local objects.
2) Run interval callbacks for spVisual and spAudio.
3) Send out messages to other processes.
4) Wait so that the desired update interval is achieved.
5) Receive information from other processes.
6) Run actions processes.
7) Run alerters on remote objects.
8) Run interval callbacks.

Because running spAppBody occurs outside of spWMUpdate, it is not shown in the list of operations above, whereas the waiting that is needed is shown as an interior step. It is easier to understand what is happening if the steps are rolled around so that the waiting is at the top and spAppBody is an interior step as shown below. This retelling of the story makes the closeness in time of various steps clearer without changing what is happening.

4) Wait so that the desired update interval is achieved.
5) Receive information from other processes.
6) Run actions processes.
7) Run alerters on remote objects.
8) Run interval callbacks.
0) Run spAppBody
1) Run alerters on local objects.
2) Run interval callbacks for spVisual and spAudio.
3) Send out messages to other processes.

Note that waiting must include allowing other independent processes to run. Receiving outside information includes properly indexing new actions, interval callbacks, and alerters. Waiting and receiving information from other processes can be combined in order to reduce timing variations due to changes in the number of received messages. Running alerters includes running those that are triggered by the removal of remote and local objects respectively.

The primary factor that controls the order of events above is the desire to minimize latency between the time information is generated and the time it can be used.

(A) Information from other processes should be integrated into the world model after any waiting which is necessary so that the newest possible data will be available to the application. However, outside information should be integrated before operations are performed that are part of the application (i.e., spAppBody, but also locally created actions, alerters, and callbacks).

(B) Sending out messages about objects that have changed should happen after all operations are performed that are part of the application, but as soon after as possible, so that others hear about every change made as soon as possible.

(C) Actions from other processes want to occur after information from other processes is integrated, but before application operations start, because these actions make changes in objects from other processes and the application wants to respond to these changes just as if change messages about these objects were received.

(D) A primary use of alerters is to detect situations that the spAppBody wants to respond to. Alerters should therefore run before spAppBody, but of course after outside information is processed so they can operate based on current information.

(E) Any operations corresponding to spVisual and/or spAudio running in the same process should happen after all application operations have been completed so that they can respond to every change that the application makes.

The above constraints do not indicate whether interval callbacks associated an application should run before or after spAppBody. The system assumes that it makes sense to run them before.

An awkward implication of the constraints above is that alerters associated with spVisual and spAudio are required to run in a different place than those associated with the application—i.e. after spAppBody rather than before. This problem can be resolved by noting that alerters can be run at two different times as long as all the alerters on any given object are run at the same time. In particular, spWMUpdate runs alerters on external objects before spAppBody while running alerters on local objects after spAppBody.

Note that it does not matter to spVisual and spAudio that alerters on external objects are run before spAppBody, because the local application cannot modify an external object. Similarly, running alerters on local objects after spAppBody does not matter to the application, because an alerter cannot detect a change in a local object until after the change has been made, and only the local application can make such a change.

(When ownership of an object changes, the time at which alerters are checked changes, but the alerters being used do not change and the old values still correspond correctly to when alerters were last applied.)

A similar problem arises with regard to the interval callbacks for spVisual and spAudio. This can easily be handled by making special arrangements in the system core to run the interval callbacks owned by these processes at a special time.

5.17 spWMGenerateOwner (Fundamental)

```
spName spWMGenerateOwner()
```

Return value - A newly generated external owner id.

Activities in processes are identified by 32-bit session-wide unique ids of type spName. These *owner* ids are used to tag the objects created by different activities. The main owner id of a process is assigned when the world model is initially created. Additional ids can be created by using the function spWMGenerateOwner. Each time this function is called it returns an additional owner id.

There are two kinds of owner ids: system and ordinary. spWMGenerateOwner returns ordinary owner ids. System owner ids are only used by the system core; there is no function in the API for creating them.

The purpose for having multiple owner ids in a single process is as the basis for controlling the visibility of objects via world model view masks (Section 7).

5.18 spWMReportError

```
void spWMReportError(sp Object, long Code, char * Description)
```

Object - The object that is the target of the error.
Code - Integer identifying the error.
Description - Textual description of the error.
Return value - There is no return value.

The API utilizes a uniform approach to error reporting. This centers around the error reporting strings created by spWMReportError. Whenever an error occurs, spWMReportError is called and an error reporting string is stored so that it can be easily retrieved.

An error is created based on the data passed to spWMReportError as follows. The description string becomes the heart of the error report. It should be human readable, containing as much contextual information as possible. The system does not modify the descriptive string and does not retain a pointer to it.

The code is converted to ASCII and appended to the front of the description followed by a blank. The code should be a unique identifier of the error. (The codes for the errors reported by the system are all negative and every different error has its own unique error code. Applications are advised to use positive error codes.) Programs that want to handle errors can tell which error occurred by looking at the error code portion of the report string.

An ASCII version of the full GUID of the object argument, if any, is appended to the end of the error report. This makes it possible to locate the object in question during debugging. The GUID is used as an identifier so that the object can be located on multiple machines.

The error string created is stored so that it can be retrieved as the value of spWMGetError and spWMGetLastError.

As an example of the way error reporting is used, consider that a careful program that was not completely sure that the variable X contained an spDisplaying, might retrieve the VisualDefinition from X as follows:

```
spWMSetError(NULL);
appearance = spDisplayingGetVisualDefinition(X);
if (spWMGetError()) ...
```

5.19 spWMRegister (Fundamental)

```
void spWMRegister(sp * Pointer)
```

Pointer - Pointer to be registered.
Return value - There is no return value.

After a shared object has been removed, the storage associated with it is eventually freed. However, if a pointer is registered using the function spWMRegister, then the system will never free an object that is pointed to by the pointer. This means that it will be safe to save an object in this pointer indefinitely (Section 1.5.3). To stop protection, thereby allowing eventual freeing of the storage, call spWMDeregister.

The system essentially supports a restricted form of garbage collection where the function spWMRegister is used to specify exactly which pointers are taken into account for garbage collection purposes. Note that several threads can each register the same pointer and the pointer will be protected until after all the threads have deregistered it. Note also that it is pointers that are being protected, not objects *per se*.

There are several alternatives other than the scheme used here that could have been used to assure safe access through pointers. The best is full garbage collection, but this is not practical in C, because the system would have to support it entirely by itself. Another alternative would be to have reference counts. However, as a practical matter, it is difficult and error prone for an application to manipulate reference counts. The scheme used has the advantage of being simple to understand and robust in the presence of multiple activities holding pointers into the same set of objects.

5.20 spWMDeregister (Fundamental)

```
void spWMDeregister(sp * Pointer)
```

Pointer - Pointer to be deregistered.
Return value - There is no return value.

Cancels the protection of a pointer started by spWMRegister. It is important to cancel registration when a pointer no longer needs to be protected. This is particularly true if the pointer is stack allocated and the function it is declared in is about to return, rendering the address of the pointer meaningless. Calling spWMRemove cancels all pointer registration.

6 spFn (Fundamental)

```
public abstract class spFn
```

An important part of the API is functions that map functional arguments over objects. There are three basic kinds of mapping functions.

Examiners - Inspect objects existing in the local world model copy.
Monitors - Inspect objects existing in the local world model copy and then continue inspecting similar objects that appear in the local world model copy in the future. (Monitors are a combination of an examiner and a callback.)
Callbacks - Apply functions to objects in the local world model copy when events occur in the future.

The type spFn is used for the functional arguments to all the above functions. (A single type is used in all these situations because this is considerably more convenient than having to define a separate type for each purpose.) The spFn type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spFn data is stored in shared objects and used in intermediate computation.

In C, spFn is the following functional type:

```
typedef spBoolean spFn(sp Object, void * State)
```

Object - The object the mapped function is being applied to.
State - Passed to the operation each time it is called (modifiable).
Return value - True signals that mapping should stop.

The State argument is used to communicate state information between calls to an spFn. It can be used to accumulate a result. The return value is used for search-like operations. If an spFn ever returns True, then the mapping activity immediately halts. It is essential that an spFn be light weight in the sense that it runs quickly, and must not call spWMUpdate.

The following spFn could be used to count.

```
spBoolean spCount(sp ignore, void * state) {
  int * count = (int *)state;
  *count = *count+1;
  return FALSE;
}
```

For instance, the following code counts the number of children of an object X.

```
int Counter = 0;
spExamineChildren(X, spMaskNORMAL, spCount, &Counter);
Result = Counter;
```

The value of the count is obtained by observing the value of the state variable Counter after the examination is over.

Note that the state value that is passed to spExamineChildren and then on to the spFn spCount, is passed directly without copying. This is essential so that it can communicate information by side-effect. However, it means that the state must be in existence for the full period of time that spExamineChildren runs. This is not a problem for spExamineChildren, but requires more thought for operations like spBeaconMonitor where the operate continues asynchronously for a potentially long period of time.

6.1 spFn Predicates

Another key part of the API is the ability to install alerter functions that will automatically be called when certain events occur. Events are defined by predicates that test changes in the shared variables of objects. These predicates are defined using the same type spFn as functions that are mapped over objects.

When an spFn is used as a alerter predicate, the return value is interpreted as specifying whether an event has occurred. Specifically, the predicate should return True or False depending on whether it observes that the event it tests for has occurred.

Callback predicates define events in terms of changes to shared instance variables. In particular, they compare the current state of these variables with the state of these variables at the end of the last call on spWMUpdate. (Since alerter predicates are evaluated during each call on spWMUpdate, this ensures that any state change will be detected.)

The following shows an example of a simple alerter predicate.

```
spBoolean spChangedParent(sp object, void * ignore) {
  return spThingGetOldParent(object) != spThingGetParent(object);
}
```

There are two key things to note about alerter predicates. First, since alerters are only applied to objects whose shared variables have changed, events must involve some change in these variables. Second, alerter predicates should not modify the object passed to them.

The API includes the following predefined alerter predicates. Users can define any other predicates they want.

spJustNew -	True if object just created.
spJustRemoved -	True if object just removed.
spChanged -	True if object changed in any way (or just new or just removed).
spChangedTransform -	True if Transform of spPositioning changed.
spChangedVisualDefinition -	True if visual definition of spDisplaying changed.
spChangedParent -	True if Parent changed.
spChangedLocale -	True if the locale an object is in has changed.
spChangedBeaconWithTag -	True if object is a changed beacon with matching tag.
spRelevantOwnershipRequest -	True if object is a newly appeared spOwnershipRequest for an object owned by the local process.

Note that if there are spJustNew and an spJustRemoved alerters that are capable of firing on a given object X and if X is created and then removed so quickly that there is no opportunity for alerters to fire between the time X is created and removed, then the spJustRemoved alerter will fire, but the spJustNew alerter will not. This is an unusual situation, but spJustRemoved alerters should be prepared to deal with it. Note that allowing both alerters to fire would not be much help, because there would be no guarantee of the order in which they would fire—the lack of order guarantees being a weakness of alerters in general.

The same basic problem exists when alerters that monitor changes are in effect. Once an object has been removed, the only alerter above that can fire is an spJustRemoved alerter.

Alternatively, one could pretend that from the perspective of alerters, objects that come and go that quickly simply never exist. However, this would help only with regard to objects that are rapidly created and removed. It would not help with objects that are rapidly changed and then removed.

6.2 Old Values of Shared Variables

To make the comparisons done by alerter predicates possible, spWMUpdate maintains saved versions of the shared instance variables of every shared object. These versions reflect the state of the variables just before the last call on spWMUpdate returned. The saved values can be accessed using access functions of the form GetOld (Section 1.5.1). (At the moment an object is created, or first heard about in a particular world model copy, the old values are set equal to the new values.)

It must be kept in mind that the old variable values saved at the end of spWMUpdate reflect the state **after** information has been processed about changes in objects caused by other processes. Therefore in the absence of changes made by alerters, the old and current values of every variable are the same after spWMUpdate returns. This means that in the general application processing between calls of spWMUpdate, old values have little use. The only time they differ from current values is when the application itself has made a change, which is something the application can know without consulting the old values.

The time when the old values are useful is during the processing of alerters, just before spWMUpdate returns. When a alerter is under consideration for a given object, the difference between the old and new values reflect the net result of everything that has happened to the object since the alerter was last under consideration.

The way old values are saved is specifically designed for the convenience of alerters. It is not considered a defect of the API that they are not particularly useful in other contexts.

It is worthy of note that the way the old values are set guarantees that an old value, cannot point to a removed object for more than one cycle between calls on spWMUpdate. This is true, because removing an object registers as a change to every object that refers to the removed object. This causes the old pointer to the removed object to be written over with Null at the end of the next call on spWMUpdate.

For efficiency, the system focuses on being able to store old values rapidly rather than being able to retrieve them rapidly. This is important, since these values are typically written much more often then they are retrieved. However, one should be aware that reading them can be slower than reading current values.

Instead of old values one could simply have accessors that return boolean values specifying whether the corresponding variable could possibly have changed. This has slightly different semantics because it records whether there ever might have been a change rather than whether there actually is now a change. Also, there is of course less information available.

Using change bits, would save storage. It might also allow more efficient indexing to select which alerters could possibly apply in a given situation, which could therefore save time. Some might think that the bits were simpler to understand; however, others might think that they are inherently imprecise.

On the negative side using change bits would require applications that wanted to access old data to explicitly store it somewhere. A bigger problem with change bits is that there are many more situations where they would have to be computed than situations where they would be used. In particular, whenever a message is received, proper bits representing changes would have to be computed. This is needed for each field on each message receipt even if there is no code anywhere that ever looks at the bits, because such code could appear at any moment. (Whenever any message is received about an object, the Change bit is set, but this is easy.)

7 spMask

```
public class spMask
```

A fundamental feature of the API is the notion of a *view mask* for the world model. The purpose of these masks is to control the visibility of objects when using functions like spClassExamine. The spMask type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spMask data is stored in shared objects and used in intermediate computation.

The class spMask defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **MINE** - (1) Views your own objects (Section 7.1).
static final **OTHERS** - (2) Views other processes's objects (Section 7.1).
static final **NORMAL** - (3 = 1+2) Views all ordinary objects (Section 7.1).
static final SYSTEM - (8) Views system objects (Section 7.1).
static final CALLBACKS - (16) Views spIntervalCallbacks of other processes (Section 7.1).
static final ALL - (-1) Views all objects (Section 7.1).

There are several kinds of objects that an application typically should not see. First, they should not see objects that exist purely for the use of the system core and are not part of the external API. Second, they should not see objects that are created by another activity and exist in the local world model copy only because this activity happens to be running as part of the same process, rather than in a separate process. (For instance, an application does not want to see callbacks owned by other activities that are not communicated to other processes listening in the locales the objects are in.) Note that an application also typically does not want to see objects that have been removed. However, this is no problem because objects that have been removed are not in the local world model copy and therefore are not encountered by an application.

A world model view mask (spMask) is an integer bit mask that controls what objects are viewable. A view mask is created by adding (or or'ing together) the constants above.

Before discussing the exact meaning of these constants, it is useful to consider a simple example. The following expression applies F to every spThing object that is not the private object of some other activity.

```
spClassExamine(spThingC, spMaskNORMAL, F, NULL)
```

In contrast, the following expression applies F to every spThing object that is owned by the current activity. It does not apply F to any objects owned by other activities.

```
spClassExamine(spThingC, spMaskMINE, F, NULL)
```

The following pseudo-code shows the reasoning applied in functions like spExamineChildren to determine whether an object is compatible with a view mask.

```
boolean CompatibleWithMask(sp object, spMask mask) {
  if (spGetOwner(object) == spWMGetMe()) {
    return (spMaskMINE & mask)
  }
  else {
    return (spMaskOTHERS & mask) &&
           (spMaskSYSTEM & mask || ~ SystemOwner(spGetOwner(object))) &&
           (spMaskCALLBACKS & mask ||
            ~ spClassLeq(spGetClass(object), spIntervalCallbackC()))

  }
}
```

For a view mask to make any objects viewable, at least one of the spMaskMINE and spMaskOTHERS bits must be on. The three view masks constructible using spMaskMINE and spMaskOTHERS that view any objects are all of potential use to applications. The most common single case is spMaskNORMAL which combines spMaskMINE and spMaskOTHERS and views all ordinary objects. This is the default value for spMasks in most situations.

The spMaskSYSTEM and spMaskCALLBACKS bits are not part of the external interface and are intended only to be used by the system core and servers. These parts of the system often make use of the mask spMaskALL, which makes every object viewable.

The functions spExamineChildren, spExamineDescendants, spClassExamine, and spClassMonitor all have view mask arguments that limit the objects that are viewed. Alerters (Section 43) also make use of view masks.

View mask restrictions do not apply to the access functions (such as spGetParent) for obtaining the value of instance variables. This is not a problem because, in general, you cannot get from objects you should see to ones you should not see. Objects with ordinary owners should never point to objects with system owners. Objects that are communicated via locales never point to ones that are not. Note that no object ever points to an object that has been removed.

7.1 Constants

View masks are created by combining the following constants. (In C these constants are implemented as #defines.)

spMaskMINE - (1) View objects owned by spWMGetMe.
spMaskOTHERS - (2) View objects not owned by spWMGetMe.
spMaskNORMAL - (3) View all ordinary objects.
spMaskSYSTEM - (8) View objects with other system owners.
spMaskCALLBACKS - (16) View spIntervalCallback objects owned by others.
spMaskALL - (-1) View every object.

As illustrated above, view masks can be created by adding or or'ing these constants together.

8 spTransform

```
public class spTransform
```

The data type spTransform is used as the fundamental representation of the position, orientation, and scaling of objects. A key advantage of an spTransform is that the various components are easy to understand. The components are also well suited to interpolation. The spTransform type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spTransform data is stored in shared objects and used in intermediate computation.

The class spTransform defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **X** - (0) Index of translation X (Section 8.1).
static final **Y** - (1) Index of translation Y (Section 8.1).
static final **Z** - (2) Index of translation Z (Section 8.1).
static final **RX** - (3) Index of X coordinate of rotation axis (Section 8.1).
static final **RY** - (4) Index of Y coordinate of rotation axis (Section 8.1).
static final **RZ** - (5) Index of Z coordinate of rotation axis (Section 8.1).
static final **RA** - (6) Index of amount of rotation (Section 8.1).
static final **SX** - (7) Index of scaling along X axis (Section 8.1).
static final **SY** - (8) Index of scaling along Y axis (Section 8.1).
static final **SZ** - (9) Index of scaling along Z axis (Section 8.1).
static final **SOX** - (10) Index of X coordinate of scale axis (Section 8.1).
static final **SOY** - (11) Index of Y coordinate of scale axis (Section 8.1).
static final **SOZ** - (12) Index of Z coordinate of scale axis (Section 8.1).
static final **SOA** - (13) Index of amount of scale rotation (Section 8.1).
static final **CX** - (14) Index of center X (Section 8.1).
static final **CY** - (15) Index of center Y (Section 8.1).
static final **CZ** - (16) Index of center Z (Section 8.1).

The class spTransform defines the following functions:

spTransformCopy - Copies one spTransform into another (Section 8.2).
spTransformFromIdent - Initializes spTransform to identity transform (Section 8.3).
spTransformGetTranslation - Gets translation from transform (Section 8.4).
spTransformSetTranslation - Sets translation in transform (Section 8.5).
spTransformGetRotation - Gets rotation in transform (Section 8.6).
spTransformSetRotation - Sets rotation in transform (Section 8.7).
spTransformGetScale - Gets scale factors from transform (Section 8.8).
spTransformSetScale - Sets scale factors in transform (Section 8.9).
spTransformGetScaleOrientation - Gets scale orientation from transform (Section 8.10).
spTransformSetScaleOrientation - Sets scale orientation in transform (Section 8.11).
spTransformGetCenter - Gets center point from transform (Section 8.12).
spTransformSetCenter - Sets center point in transform (Section 8.13).

An spTransform contains the same information as a VRML transform node. Specifically, an spTransform is a vector of 17 floats representing 5 components as follows. (All units are in terms of meters and radians.)

Translation - (X,Y,Z) identity (0,0,0)
Rotation - (RX,RY,RZ,RA) identity (0,0,1,0)
Scale - (SX,SY,SZ) identity (1,1,1)
ScaleOrientation - (SOX,SOY,SOZ,SOA) identity (0,0,1,0)
Center - (CX,CY,CZ) identity (0,0,0)

The first three elements are a vector representing the translation. The next four elements are an spRotation vector representing a rotation. The next three elements specify the amount of scaling along the X, Y, and Z axes with the value 1.0 indicating no scaling. The next four elements are an spRotation that is applied before the scaling is performed. The final three elements specify a center point for both rotations.

As in VRML, the parts of an spTransform act together as follows. Note that because of the ScaleOrientation, the scale value can specify shear as well as scaling. It can be shown that the composition of any two spTransforms can be represented as an spTransform and the inverse of any spTransform can be represented as an spTransform.

```
(translate by Translation
 (translate by Center
  (rotate by Rotation
   (rotate by ScaleOrientation
    (scale by Scale
     (rotate by -ScaleOrientation
      (translate by -Center
       ...)))))))
```

By convention, a right-hand coordinate system is used with the Y axis up and objects facing down the negative Z axis. (No assumptions are made about the relationship of the X and Y axes to compass directions.) It should be noted that different graphics modeling languages disagree with each other about these conventions. For instance, some have the Z axis up. The way spVisualDefinition links are specified makes it easy to use any kind of graphic model, without having to modify it.

The origin of the coordinate system for an object should be in the middle of the object unless there is a compelling reason otherwise. This is needed so that the InRadius and OutRadius of spDisplaying objects will be meaningful. (An example of a compelling reason why the origin of an object would not be in the middle is that the origin of a subpart of an articulated form should be at the pivot point. This makes the mathematics of moving the subpart much easier.)

Any object that has a recognizable up direction should be oriented so that this up direction is parallel to the Y axis and points toward positive Y. Similarly, any object that has a recognizable front should have the front facing toward the negative Z axis. For instance, an object corresponding to the torso of an avatar would have the origin of its coordinate system in the middle of the chest, with the Y axis pointing up toward the head and the negative Z axis pointing out through the front of the torso. These axis conventions are needed both so that it is easy to combine different objects into a single scene and so that simulations that want to interact with objects can find where the tops and fronts of the objects are.

In C, an spTransform is the following type.

```
typedef float * spTransform;
```

In addition, the following type is available for allocating memory for an spTransform. When calling a function that operates on spTransforms, one can pass in a variable that is either of the type spTransform or spTransformData.

```
typedef float spTransformData[17];
```

There are several levels at which you can interact with an spTransform. First, you can alter the underlying vector directly, e.g., using the index constants (Section 8.1). This allows you to do everything that is possible, but nothing easily.

Second, you can use the functions described in the following subsections to operate on spTransform objects. These make it easier to do complex operations. These functions are particularly important to use when performing an operation that interacts with rotation, because the inherent ambiguity (Section 10.1) of rotations makes their direct manipulation quite error prone.

While advantages, the functions described below require a clear understanding of transform interactions in order to figure out how to obtain a desired effect. In addition, these operations are static in the sense they call for the computation of particular spTransforms, rather than sequences of transform values over time.

Third, you can move up to a higher level and use the smooth motion functions (Section 16.13) defined in spPositioning. These allow you to specify motion sequences rather than just calculating transforms. Whenever possible, it is advisable to operate at this higher level.

Several conventions are worthy of note about the functions on spTransform objects. None of the functions ever allocates memory. Rather, the control of memory is left entirely up to the application. All modifications are by side-effect. However, to make it easier to create nested expressions, the modified value is used as the return value.

8.1 Constants

The following constants are available for directly accessing the various elements of an spTransform. (In C, these constants are `#defines`.)

spTransformX - (0) Index of X coordinate.
spTransformY - (1) Index of Y coordinate.
spTransformZ - (2) Index of Z coordinate.
spTransformRX - (3) Index of X coordinate of rotation axis vector.
spTransformRY - (4) Index of Y coordinate of rotation axis vector.
spTransformRZ - (5) Index of Z coordinate of rotation axis vector.
spTransformRA - (6) Index of amount of rotation.
spTransformSX - (7) Index of scaling along X axis.
spTransformSY - (8) Index of scaling along Y axis.
spTransformSZ - (9) Index of scaling along Z axis.
spTransformSOX - (10) Index of X coordinate of scaling axis vector.
spTransformSOY - (11) Index of Y coordinate of scaling axis vector.
spTransformSOZ - (12) Index of Z coordinate of scaling axis vector.
spTransformSOA - (13) Index of amount of scaling rotation.
spTransformCX - (14) Index of X coordinate of center.
spTransformCY - (15) Index of Y coordinate of center.
spTransformCZ - (16) Index of Z coordinate of center.

For example, you might write.

```
spTransform P;
P[spTransformX] = P[spTransformY] + 2.0;
```

8.2 spTransformCopy

```
spTransform spTransformCopy(spTransform Destination, spTransform Source)
```

Destination - spTransform to be filled with copied data.
Source - spTransform that is the source of the data.
Return value - Modified spTransform.

Copies the data from a Source spTransform to another, which is returned.

8.3 spTransformFromIdent

```
spTransform spTransformFromIdent(spTransform Transform)
```

Transform - spTransform to be initialized.
Return value - Initialized spTransform.

Initializes the values in an spTransform so that they specify no translation, rotation, or scaling. That is to say, the spTransform is set to (0,0,0, 0,0,1,0, 1,1,1, 0,0,1,0, 0,0,0). It is important to initialize an spTransform before using it as a source of data, because the operations on spTransforms do not test that transforms are well formed before beginning their operations. In particular, they assume that the rotations in it are well formed.

8.4 spTransformGetTranslation

```
spVector spTransformGetTranslation(spTransform Transform)
```

Transform - spTransform from which to obtain translation.
Return value - Translation vector.

Returns the Translation portion of an spTransform. In C, the spVector returned shares memory with the spTransform.

8.5 spTransformSetTranslation

```
spTransform spTransformSetTranslation(spTransform Transform, spVector Translation)
```

Transform - spTransform whose translation is to be set.
Translation - Vector specifying translation.
Return value - Modified spTransform.

Sets the Translation portion of an spTransform. The other parts of the spTransform are not altered.

8.6 spTransformGetRotation

```
spRotation spTransformGetRotation(spTransform Transform)
```

Transform - spTransform from which to extract rotation information.
Return value - Rotation component of spTransform.

Returns the Rotation portion of an spTransform represented as an spRotation. In C, the spRotation returned shares memory with the spTransform.

8.7 spTransformSetRotation

```
spTransform spTransformSetRotation(spTransform Transform, spRotation Rotation)
```

Transform - spTransform whose Rotation is to be set.
Rotation - Rotation value.
Return value - Modified spTransform.

Sets the Rotation portion of an spTransform. The other parts of the spTransform are not altered.

8.8 spTransformGetScale

```
spVector spTransformGetScale(spTransform Transform)
```

Transform - spTransform from which to obtain Scale values.
Return value - Scale vector.

Returns the Scale portion of an spTransform. In C, the spVector returned shares memory with the spTransform.

The three elements of the spVector contain the amount of scale along the X, Y, and Z axes respectively. The value 1.0 indicates no change in scale.

8.9 spTransformSetScale

```
spTransform spTransformSetScale(spTransform Transform, spVector Vector)
```

Transform - spTransform whose Scale is to be set.
Vector - Vector of Scale values.
Return value - Modified spTransform.

Sets the Scale portion of an spTransform. The other parts of the spTransform are not altered.

8.10 spTransformGetScaleOrientation

```
spRotation spTransformGetScaleOrientation(spTransform Transform)
```

Transform - spTransform from which to obtain ScaleOrientation information.
Return value - ScaleOrientation rotation.

Returns the ScaleOrientation portion of an spTransform represented as an spRotation. In C, the spRotation returned shares memory with the spTransform.

8.11 spTransformSetScaleOrientation

```
spTransform spTransformSetScaleOrientation(spTransform Transform, spRotation R)
```

Transform - Transform to set ScaleOrientation in.
R - ScaleOrientation rotation.
Return value - Modified spTransform.

Sets the ScaleOrientation portion of an spTransform. The other parts of the spTransform are not altered.

8.12 spTransformGetCenter

```
spVector spTransformGetCenter(spTransform Transform)
```

Transform - spTransform from which to obtain center point.
Return value - Center point vector.

Returns the Center portion of an spTransform. In C, the spVector returned shares memory with the spTransform.

8.13 spTransformSetCenter

```
spTransform spTransformSetCenter(spTransform Transform, spVector Center)
```

Transform - Transform to set Center in.
Center - Center point vector.
Return value - Modified spTransform.

Sets the Center portion of an spTransform. The other parts of the spTransform are not altered.

9 spVector

```
public class spVector
```

The type spVector is a 3-element vector of floats. It is used to represent several distinct things. The spVector type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spVector data is stored in shared objects and used in intermediate computation.

The class spVector defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **X** - (0) Index of X component (Section 9.1).
static final **Y** - (1) Index of Y component (Section 9.1).
static final **Z** - (2) Index of Z component (Section 9.1).
static final **ZERO** - (0,0,0) Zero length vector (Section 9.1).
static final **AXISX** - (1,0,0) Unit vector along X axis (Section 9.1).
static final **AXISY** - (0,1,0) Unit vector along Y axis (Section 9.1).
static final **AXISZ** - (0,0,1) Unit vector along Z axis (Section 9.1).

The class spVector defines the following functions:

spVectorCopy - Copies vector (Section 9.2).
spVectorSetFromScalar - Sets vector elements from scalar value (Section 9.3).
spVectorEquals - Tests equality (Section 9.4).
spVectorEqualsDelta - Tests equality within delta (Section 9.5).
spVectorAdd - Adds two vectors (Section 9.6).
spVectorSubtract - Subtracts two vectors (Section 9.7).
spVectorMultiplyByScalar - Multiplies by a scalar (Section 9.8).
spVectorDivideByScalar - Divides by scalar (Section 9.9).
spVectorCrossProduct - Computes cross product of two vectors (Section 9.10).
spVectorDotProduct - Computes the dot product of two vectors (Section 9.11).
spVectorComposeScales - Computes element-by-element product (Section 9.12).
spVectorLength - Computes the length of a vector (Section 9.13).
spVectorNormalize - Normalizes vector to length 1.0 (Section 9.14).

There are three major uses of vectors. The first is as an ordinary vector in Cartesian coordinates, with the three elements being the X, Y, and Z coordinates respectively. These vectors are used to specify translations, rotation axes, and center points.

The second use of vectors is as a set of Euler angles. In this use, the first element is a rotation around the X axis in radians; the second element is a rotation around the Y axis in radians; and the third element is a rotation around the Z axis in radians. Since rotations do not commute, it is important to realize that these correspond to first rotating around Z, and then rotating around Y, and lastly rotating around X.

The third use of vectors is as a representation for scaling. In this use, the three elements represent the amount of scaling in the X, Y, and Z axes respectively. A value of 1.0 in an element specifies no scaling.

In C, an spVector is the following type.

```
typedef float * spVector;
```

In addition, the following type is available for stack allocating memory for an spVector. When calling a function that operates on spVectors, one can pass in a variable that is either of the type spVector or spVectorData.

```
typedef float spVectorData[3];
```

Several conventions are worthy of note about the functions on spVectors. None of the functions ever allocates memory. Rather, the control of memory is left entirely up to the application. All modifications are by side-effect. However, to make it easier to create nested expressions, the modified vector is used as the return value.

9.1 Constants

The following three constants are provided for accessing the components of an spVector. (In C, these constants are external variables initialized to appropriate values.)

spVectorX - (0) Index of X coordinate.
spVectorY - (1) Index of Y coordinate.
spVectorZ - (2) Index of Z coordinate.

For example, you might write.

```
spVector V;
V[spVectorX] = P[spVectorY] + 2.0;
```

The following four constants contain particular vectors that are useful as arguments to various API functions. (In C, these constants are external variables initialized to appropriate values.)

spVectorZERO - (0,0,0) Zero vector.
spVectorAXISX - (1,0,0) Vector along X axis.
spVectorAXISY - (0,1,0) Vector along Y axis.
spVectorAXISZ - (0,0,1) Vector along Z axis.

For example, to compute the rotation that is needed in order to look from the origin down the X axis with the Y axis up, you might write.

```
spRotation R;
spRotationLookAt(R, spVectorZERO, spVectorAXISX, spVectorAXISY);
```

9.2 spVectorCopy

```
spVector spVectorCopy(spVector Destination, spVector Source)
```

Destination - spVector to set.
Source - spVector to copy.
Return value - Modified vector.

Copies a vector into another.

9.3 spVectorSetFromScalar

```
spVector spVectorSetFromScalar(spVector Vector, float Scalar)
```

Vector - spVector whose elements are to be set.
Scalar - Value to set elements to.
Return value - Modified spVector.

Sets all three elements of an spVector to be equal to a given Scalar.

9.4 spVectorEquals

```
spBoolean spVectorEquals(spVector A, spVector B)
```

A - Vector to compare.
B - Vector to compare.
Return value - True if vectors are equal.

Returns True if two vectors are component-by-component equal within a small system-defined tolerance.

9.5 spVectorEqualsDelta

```
spBoolean spVectorEqualsDelta(spVector A, spVector B, float Tolerance)
```

A - Vector to compare.
B - Vector to compare.
Tolerance - Precision of comparison.
Return value - True if vectors are equal within delta.

Returns True if two vectors are component-by-component equal within a tolerance specified by the user.

9.6 spVectorAdd

```
spVector spVectorAdd(spVector A, spVector B)
```

A - spVector to add vector to.
B - spVector to add.
Return value - Modified vector.

Adds two vectors together and stores the result in the first vector.

9.7 spVectorSubtract

```
spVector spVectorSubtract(spVector A, spVector B)
```

A - spVector to subtract vector from.
B - spVector to subtract.
Return value - Modified vector.

Subtracts a second vector from a first vector and stores the result in the first vector.

9.8 spVectorMultiplyByScalar

```
spVector spVectorMultiplyByScalar(spVector Vector, float Scalar)
```

Vector - spVector to multiply by scalar.
Scalar - Value to multiply by.
Return value - Modified vector.

Modifies a vector by multiplying each element by a scalar.

9.9 spVectorDivideByScalar

```
spVector spVectorDivideByScalar(spVector Vector, float Scalar)
```

Vector - spVector to divide by scalar.
Scalar - Value to divide by.
Return value - Modified vector.

Modifies a vector by dividing each element by a scalar.

9.10 spVectorCrossProduct

```
spVector spVectorCrossProduct(spVector A, spVector B)
```

A - First spVector to multiply.
B - Second spVector to multiply.
Return value - Modified vector.

Computes the cross product of two vectors and stores it in the first vector.

9.11 spVectorDotProduct

```
float spVectorDotProduct(spVector A, spVector B)
```

A - The first vector to multiply.
B - The second spVector to multiply.
Return value - The dot product.

Computes the dot product of two vectors and stores it in the first vector.

9.12 spVectorComposeScales

```
spVector spVectorComposeScales(spVector A, spVector B)
```

A - spVector of scale values that is to be modified.
B - spVector of values to compose with existing scale values.
Return value - Modified spVector.

Modifies an spVector by multiplying each element by the corresponding element of another spVector.

9.13 spVectorLength

```
float spVectorLength(spVector Vector)
```

Vector - The vector whose length is desired.
Return value - The length.

Computes the length of a vector. That is to say, the square root of the sum of the squares of the elements.

9.14 spVectorNormalize

```
spVector spVectorNormalize(spVector Vector)
```

Vector - Vector to normalize.
Return value - Modified vector.

Converts a vector into a unit length vector pointing in the same direction. That is to say, divides each element by the length of the vector.

10 spRotation

```
public class spRotation
```

The principle way that rotations are specified is in terms of a rotation axis passing through the origin and a rotation angle about this axis in radians. Typically, the rotation angle is between plus or minus π. A key advantage of an spRotation is that the various components are easy to understand. In addition, because the components are relatively independent, they are well suited to interpolation. The spRotation type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spRotation data is stored in shared objects and used in intermediate computation.

The class spRotation defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **X** - (0) Index of X value (Section 10.2).
static final **Y** - (1) Index of Y value (Section 10.2).
static final **Z** - (2) Index of Z value (Section 10.2).
static final **A** - (3) Index of Angle value (Section 10.2).

The class spRotation defines the following functions:

spRotationCopy - Copies one spRotation into another (Section 10.3).
spRotationFromIdent - Initializes spRotation (Section 10.4).
spRotationGetAxis - Returns Axis in spRotation (Section 10.5).
spRotationSetAxis - Sets Axis in spRotation (Section 10.6).
spRotationGetAngle - Returns Angle in spRotation (Section 10.7).
spRotationSetAngle - Sets Angle in spRotation (Section 10.8).
spRotationToQuat - Converts spRotation to spQuaternion (Section 10.9).
spRotationFromQuat - Converts spQuaternion to spRotation (Section 10.10).
spRotationToAngles - Converts spRotation to Euler angles (Section 10.11).
spRotationFromAngles - Converts Euler angles to spRotation (Section 10.12).
spRotationMult - Computes the composition of two spRotations (Section 10.13).
spRotationLookAt - Computes viewing rotation (Section 10.14).

An spRotation is a vector of 4 floats representing two components as follows.

Axis - (RX,RY,RZ) identity (0,0,1)
Angle - RA identity 0

The first three elements are a vector representing an axis through the origin to rotate about. The last element is an amount of rotation in radians. Radians are used because they are more natural for numerical calculations. However, most people think more easily in degrees. In recognition of this, the following constant is provided.

spDEGREES - Number of radians in a degree.

Using this constant you can specify $\pi/2$ radians (90 degrees) as follows.

```
90.0f * spDEGREES
```

In C, an spRotation is the following type.

```
typedef float * spRotation;
```

In addition, the following type is available for allocating memory for an spRotation. When calling a function that operates on spRotations, one can pass in a variable that is either of the type spRotation or spRotationData.

```
typedef float spRotationData[4];
```

Several conventions are worthy of note about the functions on spRotations. None of the functions ever allocates memory. Rather, the control of memory is left entirely up to the application. All modifications are by side-effect. However, to make it easier to create nested expressions, the modified value is used as the return value.

A key complexity in the API is that four different representations for rotations are supported. This is necessary because different communities of people are accustomed to different representations and justified because each representation has a situation where it is particularly convenient. The primary rotation representation is spRotation vectors such as those included in an spTransform. The second rotation representation is a vector of 3 Euler angles (in an spVector). The Third rotation representation is a vector of 4 floats representing a quaternion (spQuaternion). The fourth rotation representation is the rotation part of an spMatrix.

10.1 Rotation Ambiguity

A fundamental difficulty with rotations in general and spRotations in particular is ambiguity. For example, no matter how it is represented, a rotation of 2π is the same as no rotation at all. More specifically:

```
(X, Y, Z, A) = (X, Y, Z, A+2pi)  ; rotations are cyclic
(X, Y, Z, A) = (X, Y, Z, A-2pi)  ; rotations are cyclic
(X, Y, Z, A) = (nX, nY, nZ, A)   ; axis length is irrelevant
(X, Y, Z, A) = (-X, -Y, -Z, -A)  ; opposite rotation about opposite axis is same
(X, Y, Z, 0) = (X2, Y2, Z2, 0)   ; if angle is zero, all axes are the same
```

One can introduce conventions to reduce the ambiguity such as limiting the range of angles, requiring unit length axes, specifying a fixed zero rotation axis, etc. However, these do not really get rid of the fundamental problem. Rather, they merely convert ambiguous situations into ones where abrupt (and awkward) changes occur. For example, ones where a small increment in the angle suddenly causes a jump to a far away value or a jump to a radically different rotation axis if the angle becomes zero.

It is important to note that these ambiguities are only a problem if you operate directly on the parts of a rotation. For example, a program might set the axis and angle in a rotation and then periodically add a small increment to the angle in order to rotate the object around the originally specified axis.

Hidden in this approach is the assumption that the axis will not change and therefore that incrementing the angle means rotating a small amount around this fixed axis. Doing a rotation this way is a hidden time bomb in a program unless you can be totally sure that the axis really will not be changed by any other operation.

Verifying this assumption is made very difficult by the presence of the ambiguities outlined above. To verify the assumption one would presumably start by verifying that the rotation as a whole will not be altered by anything other than the program you are writing. However, this is not enough, because there are a variety of situations where an operation could be applied somewhere else that does not change the overall value of the rotation, but converts it into a different representation for the same value. For example, if the angle happens to be zero at a given moment, such an operation could make an arbitrary change to the axis.

An important example where this problem could arise is when your program causes rotations to be converted from one form to another and then back again. This includes using inverse pairs of the functions spRotationToQuat, spRotationFromQuat, spRotationToAngles, spRotationFromAngles, and spRotationFromAngles, but also spMatrixToTransform, and spMatrixFromTransform, (and therefore spPositioningMatrix and spPositioningRelativeMatrix). For example, if you compute the spMatrix corresponding to an spTransform, change something in the matrix that has nothing to do with rotation, and then convert back to a transform, you might get a rotation that has been canonicalized in some way and therefore internally is not the same as the rotation you started with, even though it has the same net rotational effect.

Another example is the operation spPositioningLocalize, which has to adjust transforms when the coordinate system changes as an object moves from one locale to another. Even if there is no change in rotation between the coordinate systems, a rotational canonicalization can occur.

The way to avoid problems with the ambiguity of rotations is to not make assumptions! Instead of just adding an increment to an angle, to do a rotation, a program should explicitly specify the axis to use each time rather than assume it. This means that an operation like spRotationMult will have to be used instead of just doing an addition. But it means that the right rotation will occur no matter how the initial state of the rotation is represented.

Given how dangerous it is to directly modify parts of an spRotation while assuming that other parts have not been modified, it could have been decide to omit direct modification of parts from the API. Direct modification of parts is supported for two reasons. First, there are situations where an assumption of a fixed axis is reasonable (e.g., moving the hands of clock that is resting in a fixed position) and the efficiency gains in these situations are significant. Second, since rotations are directly represented as vectors in the API, programmers can always modify individual elements no matter what the API includes; therefore, the API might as well make it straightforward to at least modify the right elements.

The moral of this section is that it is unwise to directly modify an individual element of a rotation except in very special situations. Rather, one should make fully specified incremental changes in the rotation as a whole that do not make any assumptions about the prior state. Note that the incrementality is important to avoid a second pitfall that goes beyond mere ambiguity.

If you want to write a program that will work correctly, even if a rotation gets converted into a different form representing the same rotation, you could save in your program a representation of the rotation that is totally under the control of your program. For this local copy of the rotation it would be much easier to verify that just changing a single component would work correctly. You could then make fast incremental changes, and copy the stored rotation to the rotation you wish to modify each time the stored rotation is changed.

This approach works better than directly modifying just one element of a rotation stored outside your program. However, you would still be making the assumption that the rest of the system would never change the rotation itself, but rather only the way it is represented. This assumption can be wrong when an object changes locales and when multiple programs interact to effect the a rotation. It is therefore better to create operations that read the current rotational state, modify it in fully-specified incremental ways, and then write it back again, rather than making any assumptions about its current value.

10.2 Constants

The class spRotation includes the following constants for accessing the X, Y, Z, and Angle components of an spRotation. (In C, these constants are #defines.)

spRotationX - (0) Index of X value in spRotation.
spRotationY - (1) Index of Y value in spRotation.
spRotationZ - (2) Index of Z value in spRotation.
spRotationA - (3) Index of Angle value in spRotation.

For example, you might write.

```
spRotation R;
R[spRotationX] = A[spRotationY] + 2.0;
```

10.3 spRotationCopy

```
spRotation spRotationCopy(spRotation Destination, spRotation Source)
```

Destination - spRotation to be modified.
Source - spRotation to be copied.
Return value - Modified spRotation.

Copies the data from a Source spRotation to another, which is returned.

10.4 spRotationFromIdent

```
spRotation spRotationFromIdent(spRotation Rotation)
```

Rotation - spRotation to be initialized.
Return value - Initialized spRotation.

Initializes the values in an spRotation so that they specify no rotation about the Z axis. That is to say, the spRotation is set to (0,0,1,0). It is important to initialize an spRotation before using it as a source of data because the operations on spRotations do not test that rotations are well formed before beginning their operations. In particular, they assume that the axis does not have zero length.

10.5 spRotationGetAxis

```
spVector spRotationGetAxis(spRotation Rotation)
```

Rotation - spRotation to obtain Axis from.
Return value - Axis portion of spRotation.

Returns the Axis portion of an spRotation. In C, the spVector returned shares memory with the spRotation.

10.6 spRotationSetAxis

```
spRotation spRotationSetAxis(spRotation Rotation, spVector Axis)
```

Rotation - spRotation to modify.
Axis - Rotation axis.
Return value - Modified spRotation.

Sets the Axis portion of an spTransform. The angle is not altered.

10.7 spRotationGetAngle

```
float spRotationGetAngle(spRotation Rotation)
```

Rotation - spRotation to get Angle from.
Return value - Rotation Angle.

Returns the Angle in an spRotation.

10.8 spRotationSetAngle

```
spRotation spRotationSetAngle(spRotation Rotation, float Angle)
```

Rotation - spRotation to modify.
Angle - Rotation Angle.
Return value - Modified spRotation.

Sets the Angle in an spTransform. The axis is not altered.

10.9 spRotationToQuat

```
spQuaternion spRotationToQuat(spRotation Rotation, spQuaternion Quat)
```

Rotation - spRotation to be converted.
Quat - spQuaternion to modify.
Return value - Modified spQuaternion.

Computes the spQuaternion corresponding to an spRotation.

10.10 spRotationFromQuat

```
spRotation spRotationFromQuat(spRotation Rotation, spQuaternion Quat)
```

Rotation - spRotation to be modified.
Quat - spQuaternion to get rotation information from.
Return value - Modified spRotation.

Computes the spRotation corresponding to an spQuaternion. Note that this is an inherently one-to-many operation that involves an implicit canonicalization (Section 10.1) of the rotation as a consequence of picking just one value to actually return. Therefore, identity operations that include this function may alter the internal elements of a rotation even though they do not change its overall value.

10.11 spRotationToAngles

```
spVector spRotationToAngles(spRotation Rotation, spVector Vector)
```

Rotation - spRotation to convert.
Vector - spVector to be modified.
Return value - Modified spVector containing Euler angles.

Computes the Euler angles corresponding to an spRotation.

10.12 spRotationFromAngles

```
spRotation spRotationFromAngles(spRotation Rotation, spVector Angles)
```

Rotation - spRotation to be modified.
Angles - spVector of Euler angles.
Return value - Modified spRotation.

Computes the spRotation corresponding to a vector of Euler angles. Note that this is an inherently one-to-many operation that involves an implicit canonicalization (Section 10.1) of the rotation as a consequence of picking just one value to actually return. Therefore, identity operations that include this function may alter the internal elements of a rotation even though they do not change its overall value.

10.13 spRotationMult

```
spRotation spRotationMult(spRotation A, spRotation B)
```

A - spRotation to be altered.
B - Additional spRotation to be applied.
Return value - Modified spRotation.

Computes an spRotation that represents the composition of two spRotations. Specifically, an spRotation A is modified to represent the effect of applying a second rotation B after A. Note that this means that B is effectively multiplied on the left, not the right.

The composition is done by converting both angles to spQuaternions and then converting the result back. This last step is an inherently one-to-many operation that involves an implicit canonicalization (Section 10.1) of the rotation as a consequence of picking just one value to actually return. Therefore, even if B specifies no rotation, or both A and B have the same rotation axis, the result might not have the same rotation axis as A.

10.14 spRotationLookAt

```
spRotation spRotationLookAt(spRotation R, spVector From, spVector To, spVector Up)
```

R - spRotation to contain result.
From - (X,Y,Z) position to look from.
To - (X,Y,Z) point to look at.
Up - Vector specifying up direction.
Return value - Modified spRotation.

Computes the absolute spRotation that should be used to view one position from another. In particular, an spRotation R is computed so that if the rotation is used for an object A at the From position, then the negative Z axis of A's coordinate system points to the specified To position, and A's Y axis and the specified Up vector are co-planar. Typically, the Up vector is chosen to be parallel to the Y axis of the main coordinate system. This causes objects that are upright to appear upright if the rotation is used to position an spSeeing beacon.

The rotation that is the target of spRotationLookAt is modified by filling it in with the viewing rotation and then returned.

11 spQuaternion

```
public class spQuaternion
```

Several different representations for rotations are supported. One of these is spQuaternion, which is a 4-element vector of floats representing a quaternion. There are very few functions in the class spQuaternion, because spRotation is much more central to the API. The spQuaternion type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spQuaternion data is stored in shared objects and used in intermediate computation.

The class spQuaternion defines the following functions:

spQuaternionCopy - Copies one spQuaternion into another (Section 11.1).
spQuaternionFromIdent - Initializes spQuaternion (Section 11.2).
spQuaternionMult - Computes the composition of two spQuaternions (Section 11.3).

The four elements of an spQuaternion represent a quaternion number. (The vector is required to represent a unit quaternion, i.e., the sum of the squares of the elements of an spQuaternion is 1.0.)

```
q[0]*i + q[1]*j + q[2]*k + q[3]
```

In C, an spQuaternion is the following type.

```
typedef float * spQuaternion;
```

In addition, the following type is available for stack allocating memory for an spQuaternion. When calling a function that operates on spQuaternions, one can pass in a variable that is either of the type spQuaternion or spQuaternionData.

```
typedef float spQuaternionData[4];
```

11.1 spQuaternionCopy

```
spQuaternion spQuaternionCopy(spQuaternion Destination, spQuaternion Source)
```

Destination - spQuaternion to be modified.
Source - spQuaternion to be copied.
Return value - Modified spQuaternion.

Copies the data from a Source spQuaternion to another, which is returned.

11.2 spQuaternionFromIdent

```
spQuaternion spQuaternionFromIdent(spQuaternion Quaternion)
```

Quaternion - spQuaternion to be initialized.
Return value - Initialized spQuaternion.

Initializes the values in an spQuaternion so that they specify no rotation. That is to say, the spQuaternion is set to (0,0,0,1). It is important to initialize an spQuaternion before using it as a source of data because the operations on spQuaternions do not test that the spQuaternions are well formed before beginning their operations.

11.3 spQuaternionMult

```
spQuaternion spQuaternionMult(spQuaternion A, spQuaternion B)
```

A - spQuaternion to be altered.
B - Additional rotation to be applied.
Return value - Modified spQuaternion.

Computes an spQuaternion that represents the composition of two spQuaternions. Specifically, an spQuaternion A is modified to represent the effect of applying a second rotation B after A. Note that B is effectively multiplied on the left, not the right.

12 spMatrix

```
public class spMatrix
```

Following standard graphics practice, the positions, orientations, scaling, and shear of objects can be represented using 4x4 transformation matrices called *spMatrix* objects. These matrices are also capable of representing other effects such as taper; however, because spTransforms are used as the fundamental representation, these additional effects cannot be used. The various operations on spMatrix objects assume that the matrix corresponds purely to translation, rotation, scaling, and shear. The spMatrix type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spMatrix data is stored in shared objects and used in intermediate computation.

The class spMatrix defines the following functions:

spMatrixCopy - Sets one matrix from another (Section 12.1).
spMatrixFromIdent - Initializes spMatrix to identity matrix (Section 12.2).
spMatrixGetTranslation - Gets translation component (Section 12.3).
spMatrixSetTranslation - Sets translation component (Section 12.4).
spMatrixFromTransform - Sets matrix from transform (Section 12.5).
spMatrixToTransform - Computes spTransform from matrix (Section 12.6).
spMatrixInverse - Computes inverse of matrix (Section 12.7).
spMatrixMult - Multiplies two matrix objects (Section 12.8).
spMatrixMultVector - Multiplies matrix times vector (Section 12.9).

An spMatrix is a matrix of the following form. The upper right hand quadrant is a 3x3 rotation matrix specifying orientation, scaling, and other effects. The last column specifies translation (i.e., position).

```
R R R X
R R R Y
R R R Z
0 0 0 1
```

An spMatrix is stored in column major order as a vector of 16 floats. In C, an spMatrix is the following type.

```
typedef float * spMatrix;
```

In addition, the following type is available for stack allocating memory for an spMatrix. When calling a function that operates on spMatrix objects, one can pass in a variable that is either of the type spMatrix or spMatrixData.

```
typedef float spMatrixData[16];
```

Several conventions are worthy of note about the functions on spMatrix objects. All of the functions assume that an spMatrix represents only rotation, translation, scaling, and shear with no other effects like taper or reflection. None of the functions ever allocates memory. Rather, the control of memory is left entirely up to the application. All modifications are by side-effect. However, to make it easier to create nested expressions, the modified value is used as the return value.

12.1 spMatrixCopy

```
spMatrix spMatrixCopy(spMatrix Destination, spMatrix Source)
```

Destination - spMatrix to set.
Source - Value to copy into matrix.
Return value - Modified matrix.

Modifies one matrix to equal another.

12.2 spMatrixFromIdent

```
spMatrix spMatrixFromIdent(spMatrix Matrix)
```

Matrix - spMatrix to be set to the identity matrix.
Return value - Initialized spMatrix.

Modifies an spMatrix to be the identity matrix. The identity matrix is one where the rotation matrix is the identity matrix specifying no changes and there is no translation. That is to say, the spMatrix is set to (1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1). It is important to initialize an spMatrix before using it as a source of data because the operations on spMatrix objects do not test that the matrices are well formed before beginning their operations.

12.3 spMatrixGetTranslation

```
spVector spMatrixGetTranslation(spMatrix Matrix)
```

Matrix - spMatrix from which to obtain translation.
Return value - Translation vector.

Returns the translation component of an spMatrix. In C, the spVector returned shares memory with the spMatrix.

12.4 spMatrixSetTranslation

```
spMatrix spMatrixSetTranslation(spMatrix Matrix, spVector Translation)
```

Matrix - spMatrix whose translation is to be set.
Translation - spVector specifying translation.
Return value - Modified spMatrix.

Sets the translation component of an spMatrix, leaving the rest of the spMatrix unmodified.

12.5 spMatrixFromTransform

```
spMatrix spMatrixFromTransform(spMatrix Matrix, spTransform Transform)
```

Matrix - spMatrix to set.
Transform - spTransform to copy into the matrix.
Return value - Modified spMatrix.

Modifies a matrix so that it has the translation, rotation, scaling, and shear specified by an spTransform.

12.6 spMatrixToTransform

```
spTransform spMatrixToTransform(spMatrix Matrix, spTransform Transform)
```

Matrix - spMatrix to get information from.
Transform - spTransform to set.
Return value - Modified transform.

Computes the spTransform that corresponds to an spMatrix and stores the result in the indicated spTransform. It should be noted that this operation is computationally quite expensive. In addition, it is an inherently one-to-many operation that involves an implicit canonicalization (Section 10.1) of the rotation as a consequence of picking just one value to actually return. Therefore, identity operations that include this function may alter the internal elements of a rotation even though they do not change its overall value.

Not every spMatrix can be converted into an spTransform. For example, an spTransform that specifies taper cannot be. However, any spMatrix that can be computed by using the functions provided here can be converted into an spTransform. The only exception to this is if the individual elements of an spMatrix are set directly. This is not recommended.

12.7 spMatrixInverse

```
spMatrix spMatrixInverse(spMatrix spMatrix)
```

spMatrix - Matrix to be inverted.
Return value - Modified matrix.

Modifies a matrix so that it becomes the inverse of its previous value. That is to say, so that it specifies an opposite rotation about the same axis, inverse scaling, and the negation of the translation. The product of a matrix and its inverse is the identity matrix.

12.8 spMatrixMult

```
spMatrix spMatrixMult(spMatrix A, spMatrix B)
```

A - spMatrix to be multiplied on the left.
B - spMatrix to be multiplied on the right.
Return value - Modified spMatrix.

Computes the product of two matrix objects. This is the sum of the translations, the product of the scaling and the result of performing one rotation after the other. A matrix A is multiplied by another matrix B (with A on the left and B on the right) and A is modified to contain the result and returned.

12.9 spMatrixMultVector

```
spVector spMatrixMultVector(spMatrix Matrix, spVector Vector)
```

Matrix - spMatrix to be multiplied on the left.
Vector - spVector to be multiplied on the right and modified.
Return value - Modified spVector.

Multiply an spMatrix times an spVector to determine what the transformed vector is. This determines the result of the transformation specified by an spMatrix on the given vector. The result is stored in the argument vector and returned.

As an example, the following code shows how to calculate a vector Up in the local coordinate system of an spThing A that corresponds to the global up direction spVectorAXISY. The code multiplies the spMatrix relating the topmost coordinates to A's coordinates by a copy of spVectorAXISY to determine a vector in A's coordinates that corresponds to up.

```
spMatrix M;
spVector Up;
Up = spVectorCopy(Up, spVectorAXISY);
spMatrixMultVector(spPositioningRelativeMatrix(A, spTopmost(A), M), Up);
```

13 spPath

```
public class spPath
```

An spPath is a stored sequence of spTransforms separated in time that can be used to represent a motion path. Facilities are provided for recording (Section 13.3) and playing back (Section 16.16) these paths. The spPath type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spPath data is stored in shared objects and used in intermediate computation.

The class spPath defines the following functions:

spPathNew - Creates spPath (Section 13.1).
spPathFree - Frees spPath (Section 13.2).
spPathAppendTransform - Adds spTransform to end of path (Section 13.3).
spPathGetTransform - Gets spTransform from path (Section 13.4).
spPathCopy - Copies path (Section 13.5).
spPathSave - Writes path to file (Section 13.6).
spPathLoad - Loads path from file (Section 13.7).
spPathChangeStartPoint - Transforms path (Section 13.8).
spPathThin - Compresses path data (Section 13.9).

In C, an spPath is a linked list containing spTransforms and durations. The duration associated with an spTransform in an spPath specifies how much time should intervene between the previous spTransform and the spTransform tagged with the duration. For the first spTransform, the duration specifies how much time should elapse for an object moving from wherever it previously was to the first position in the spPath. Functions are provided for putting data into paths and getting the data back out. Functions are also provided for storing paths in the file system and subsequently retrieving them.

13.1 spPathNew

```
spPath spPathNew()
```

Return value - Newly created path.

Creates a new path containing no spTransforms.

13.2 spPathFree

```
void spPathFree(spPath Path)
```

Path - Path to remove.
Return value - There is no return value.

Frees the storage corresponding to an spPath.

13.3 spPathAppendTransform

```
spPath spPathAppendTransform(spPath Path, spTransform Point, spDuration Duration)
```

Path - Path to add spTransform onto end of.
Point - spTransform to add onto path.
Duration - Time delta associated with point in milliseconds.
Return value - Modified path.

Adds an spTransform and time delta to the end of an spPath. The extended path is returned. The appending may not happen entirely by modifying the path input. As a result, you must be sure to record the returned path. For example you might write the following:

```
P = spPathAppendTransform(P, Transform, 20)
```

13.4 spPathGetTransform

```
spDuration spPathGetTransform(spPath Path, long Index, spTransform Transform)
```

Path - Path to fetch spTransform out of.
Index - Zero-based index of spTransform in path.
Transform - spTransform to store result in.
Return value - Time interval corresponding to spTransform in path.

Obtains an spTransform from an spPath. The spTransform is returned by modifying the Transform argument. The corresponding time delta is returned as the value. The Index counts from zero as the index of the first spTransform. A negative duration is returned if Index is beyond the end of the path.

13.5 spPathCopy

```
spPath spPathCopy(spPath Path)
```

Path - Path to copy.
Return value - Newly created copy of path.

Creates a new path identical to an existing path. The existing path is not modified.

13.6 spPathSave

```
void spPathSave(spPath Path, char * Name, char * File)
```

Path - Path to save.
Name - Identifying name for path in the file.
File - Name of file to hold path.
Return value - There is no return value.

Saves an spPath in a disk file. Path files can contain several paths identified by names. This is convenient when recording a number of paths corresponding to the different parts of an articulated figure.

13.7 spPathLoad

```
spPath spPathLoad(spPath Path, char * Name, char * File)
```

Path - Path to load data into.
Name - Identifying name of path in the file.
File - Name of file holding path.
Return value - Modified path.

Reads the path with the specified name from the specified file. This is done by replacing the contents (if any) of an existing path which is returned.

13.8 spPathChangeStartPoint

```
spPath spPathChangeStartPoint(spPath Path, spTransform Transform)
```

Path - Path to convert.
Transform - New starting point.
Return value - Modified path.

An spPath specifies a trajectory starting from a particular absolute position. Often it is valuable to be able to follow a relative trajectory from a separately chosen starting point. spPathNewStart-Point accommodates this by changing the starting point of a path. This is done by multiplying each transform in the path by an appropriate conversion matrix. The change is made by modifying the path, which is returned.

13.9 spPathThin

```
spPath spPathThin(spPath Path, float Tolerance)
```

Path - Path to compress.
Tolerance - Error tolerance.
Return value - Modified path.

Often an spPath will contain much more data than necessary. For example, there may be many spTransforms in a row that correspond to very nearly the same position. spPathThin eliminates as many spTransforms as possible from the path without causing an unreasonable difference between the old and new paths. How great a difference is allowed is specified by the tolerance parameter. This places a maximum allowed delta between preexisting and interpolated data points. One can often use a quite large tolerance value without altering the visual appearance of a path. The change is made by modifying an existing path, which is returned.

14 spFormat

```
public class spFormat
```

The spFormat type does not extend the class sp and does not correspond to objects in the shared world model. Rather, spFormat data is stored in shared objects and used in intermediate computation. spFormats specify various sound data formats including sampling rate and encoding.

The class spFormat defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **LINEAR8MONO16** - 8k linear samples per second (Section 14.1).
static final **LINEAR16MONO16** - 16k linear samples per second (Section 14.1).
static final **LINEAR32MONO16** - 32k linear samples per second (Section 14.1).

The class spFormat defines the following functions:

spFormatDurationFromLength - Computes time duration (Section 14.2).
spFormatLengthFromDuration - Computes length in bytes (Section 14.3).

In C, An spFormat is the following structure. The various fields can be independently manipulated; however, it is primarily intended that applications use a small set of constant values and the small number of methods described below.

```
typedef struct _spFormat {
  unsigned short rate;
  short encoding;
  unsigned short bitsPerSample;
  unsigned short samplesPerFrame;
} spFormat;
```

The rate specifies how many times per second the sound data is sampled. Popular rates include 8000, 16000, 32000, and 44100. The encoding (an integer code) specifies what kind of data encoding is used. The bits per sample specifies how many bits (e.g., 8, 16) are in each sample. The samples per frame specifies how many channels of data are represented (e.g., 1 (mono), 2 (stereo), 4 (quad)).

All sound data accessible to applications is linearly encoded with 16 bits per sample. (More compact encodings are used when communicating between processes.) Typically, sound is output through spAudioSources in mono and rendered through a user's headphones in stereo.

14.1 Constants

The following constants contain useful spFormat values. (In C these constants are #defines.)

spFormat8LINEAR16MONO - 8k samples per second, linear encoding, 16 bit samples, mono.
spFormat16LINEAR16MONO - 16k samples per second, linear encoding, 16 bit samples, mono.
spFormat32LINEAR16MONO - 32k samples per second, linear encoding, 16 bit samples, mono.

14.2 spFormatDurationFromLength

```
spDuration spFormatDurationFromLength(spFormat Format, long Bytes)
```

Format - Format sound data is stored in.
Bytes - Number of bytes of data.
Return value - Time required to play sound in milliseconds.

Computes the duration in milliseconds (rounded up to the nearest millisecond) of the specified number of bytes of sound data stored using the indicated format.

14.3 spFormatLengthFromDuration

```
long spFormatLengthFromDuration(spFormat Format, spDuration Duration)
```

Format - Format sound data is stored in.
Duration - Time required to play sound in milliseconds.
Return value - Number of bytes of data.

Computes the number of bytes required to represent the specified number of milliseconds of sound data using the indicated format.

15 sp (Fundamental)

```
public abstract class sp
```

This is the root of the shared object hierarchy. The class is abstract in the sense that applications do not create objects of the class sp. However, the functions associated with this class are applicable to every shared object.

The class sp defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

static final **C** - spClass object for class (Section 15.1).
static final **DEGREES** - (π/180.0) Number of radians in a degree (Section 15.2).
LocalPtr - Local data (Section 15.3).
NextPtr - Hash table bucket chain (Section 15.4).
Marker - Validation code (Section 15.5).
shared DescriptionLength - Shared data length (Section 15.6).
shared Counter - Message counter (Section 15.7).
shared Name - Unique ID (Section 15.8).
shared **Class** - Class description (Section 15.9).
shared **Owner** - Unique ID of owner (Section 15.10).
shared Locale - Containing locale (Section 15.11).
shared SharedBits - Shared boolean value storage (Section 15.12).
shared **Parent** - Containing object (Section 15.13).
shared **IsRemoved** - Removal indicator (Section 15.14).
shared ForceReliable - Indicates forced reliable communication (Section 15.15).
shared InhibitReliable - Indicates blocking of messages repairs (Section 15.16).
LocalBits - Local boolean value storage (Section 15.17).
IsNew - Newness indicator (Section 15.18).
AppData - Data for use by application (Section 15.19).
MessageNeeded - Message required indicator (Section 15.20).
Change - Shared-data change-indicator (Section 15.21).
OldPtr - Old shared data (Section 15.22).
JavaPtr - Companion Java object (Section 15.23).
Referrers - Reverse index (Section 15.24).
Alerters - Applicable actions (Section 15.25).
Msgs - Queue of messages to be processed (Section 15.26).
LastUpdateTime - Last time at which object state was updated (Section 15.27).

The class sp defines the following functions:

spNew - Creates new object (Section 15.28).
spInitialization - Initializes instance variables (Section 15.29).
spRemove - Eliminates object (Section 15.30).
spExamineChildren - Looks at child objects (Section 15.31).
spExamineDescendants - Looks at all descendant objects (Section 15.32).
spTopmost - Returns highest level ancestor of object (Section 15.33).
spPrint - Prints object readably (Section 15.34).
spLocallyOwned - Queries whether object is owned by local process (Section 15.35).
spSetParent - Sets Parent of object (Section 15.36).

It is not possible to create instances of the class sp. Rather, one can only create instances of particular subclasses of the class sp.

15.1 C

```
public static final spClass C; //* [sp]
```

Given an object, one can obtain the descriptor of the object's class by using the accessor spGetClass. This is all that is needed in many situations. However, it is also often convenient to be able to obtain the description of a particular class even though you do not have any particular instance of the class. For example, you might want to use spClassExamine to find any instances that exist.

For each class spK, the system automatically generates a constant spKC that contains the class descriptor for the class. For example, sp includes the variable spC, which can be used as follows:

```
spClassExamine(spC, spMaskNORMAL, F, NULL)
```

15.2 DEGREES

All angles are represented in radians. Radians are used because they are more natural for numerical calculations. However, most people think more easily in degrees. In recognition of this, the following constant is provided. (In C, this constant is a #define.)

spDEGREES - Number of radians in a degree.

Using this constant you can specify $\pi/2$ radians (90 degrees) as follows.

```
90.0f * spDEGREES
```

15.3 LocalPtr (Fundamental and Internal)

```
transient public int LocalPtr; //* [void *] internal

      void * spiGetLocalPtr(sp Object)
      void spiSetLocalPtr(sp Object, void * X)

      void * sqGetLocalPtr(sp Object)
      void sqSetLocalPtr(sp Object, void * X)
```

The data associated with shared objects is stored in three parts. One part contains the shared instance variables. This part is sent in messages from one process to another. The second part contains the local instance variables. The third part contains a record of the state of the shared variables just after alerters were last applied to the object.

Shared objects are directly identified by a pointer to the part containing shared values. For rapid access to the local data, a local instance variable called the *LocalPtr* contains a pointer to the part of the object containing the local data. (The LocalPtr is stored adjacent to the shared data so that it can be accessed from a pointer to the shared data, but it is not part of the shared data and is not transmitted between machines.)

The LocalPtr of a shared object is initialized when the object is created and must never by changed. Information about the LocalPtr is maintained separately in each process.

15.4 NextPtr (Fundamental and Internal)

```
transient public int NextPtr; //* [void *] internal

      void * spiGetNextPtr(sp Object)
      void spiSetNextPtr(sp Object, void * X)

      void * sqGetNextPtr(sp Object)
      void sqSetNextPtr(sp Object, void * X)
```

The system uses a hash table as an index into the shared objects in the world model based on GUIDs for the objects. The buckets of the hash table are chained through shared objects using a local instance variable called the *NextPtr*. (The NextPtr is stored adjacent to the shared data so that it can be rapidly accessed from a pointer to the shared data, but it is not part of the shared data and is not transmitted between machines.)

The NextPtr of a shared object is manipulated by the hash table algorithms used by the system and must not be modified in any other way. Information about the NextPtr is maintained separately in each process.

15.5 Marker (Fundamental and Internal)

```
transient public int Marker; //* [long] internal

      long spiGetMarker(sp Object)
      void spiSetMarker(sp Object, long X)

      long sqGetMarker(sp Object)
      void sqSetMarker(sp Object, long X)
```

To facilitate error checking and debugging, each shared object contains a special marker that is unique to shared objects. This makes it possible to determine with high probability whether a given piece of memory does or does not represent a shared object. This marker is stored in a local instance variable called the *Marker*. (The marker is stored adjacent to the shared data so that it can be accessed from a pointer to the shared data, but it is not part of the shared data and is not transmitted between machines.)

The Marker variable of a shared object is set to a special value at the moment when the object is created and set to a different value when the object is removed. It must not be modified at any other time. Information about the Marker is maintained separately in each process.

15.6 DescriptionLength (Fundamental and Internal)

```
public short DescriptionLength; //* [short] internal

      short spiGetDescriptionLength(sp Object)
      short spiGetOldDescriptionLength(sp Object)
      void spiSetDescriptionLength(sp Object, short X)

      short sqGetDescriptionLength(sp Object)
      short sqGetOldDescriptionLength(sp Object)
      void sqSetDescriptionLength(sp Object, short X)
```

Changes in the shared data associated with a shared object are communicated between processes using messages that contain a snapshot of the state of the shared data. To facilitate the construction of these messages, each shared object has a shared instance variable called the *DescriptionLength* that specifies the number of bytes occupied by the shared data. (The DescriptionLength is required to be sufficiently small to satisfy the general limitation that single UDP messages must be less than 600 bytes or so long. As part of this, spFixedAscii strings are required to be less than 500 bytes long.)

The DescriptionLength must be an integer. When an object is created, its DescriptionLength is initialized by the system to the appropriate value. The DescriptionLength never changes after it has been initialized. Information about the DescriptionLength is shared between processes.

15.7 Counter (Fundamental and Internal)

```
public short Counter; //* [short] internal

      short spiGetCounter(sp Object)
      short spiGetOldCounter(sp Object)
      void spiSetCounter(sp Object, short X)

      short sqGetCounter(sp Object)
      short sqGetOldCounter(sp Object)
      void sqSetCounter(sp Object, short X)
```

Every shared object has a shared instance variable called the *Counter* that is incremented every time a message about the object is sent to other processes. The purpose of this is to allow a process that receives messages about an object to identify and ignore duplicate and out-of-order messages. Information about the Counter is shared between processes.

15.8 Name (Fundamental and Internal)

```
public int Name; //* [spName] internal

      spName spiGetName(sp Object)
      void spiSetName(sp Object, spName X)

      spName sqGetName(sp Object)
      spName sqGetOldName(sp Object)
      void sqSetName(sp Object, spName X)
```

To support the unambiguous communication of shared objects between processes over a network, each shared object is assigned a Globally Unique ID (GUID) that is guaranteed to be unique in the world and in time for 100 years. Taking advantage of the fact that objects typically do not have long lives, names are represented using only 32 bits of type spName in most situations. Using this 32-bit compressed representation saves substantial amounts of storage and communication bandwidth in comparison with many other unique naming schemes.

Shared objects are assigned unique names when they are initially created. It is guaranteed that the Name of an object will never change. Information about the Name is shared between processes.

Note that the unique name of an object is not available in the external interface. If an application wants to uniquely identify an object, it should associate the object with an spBeacon.

The fundamental representation of object IDs uses 96 bits (12 bytes, 3 words) composed of two parts:

Process ID: 80 bits corresponding to a process. This value is assigned whenever a new process starts and is guaranteed to be unique in space and time (for a century). This value is opaque. No way is specified for obtaining any information about a process if one only has a process identifier. The process identifier 0 (zero) is reserved for indicating built-in objects—i.e., the built-in classes.

Object ID: 16 bits. As a process creates new objects, it generates names for them by changing the object ID part of the name, holding the process identifier constant. Names are never reused. Once 64k names have been generated, the process ID is changed. (As explained below, a process has available to it 64k names per second.)

To promote memory and communication efficiency, object names are represented at all times in the following compressed form:

Process ID table pointer: 16 bits that indicates an entry in a table of process ids. (The process id table pointer 0 (zero) is reserved for indicating built-in objects.)

Object ID: The 16-bit object ID for the object.

These are combined to form 32-bit values of types spName and spOwner. Note that the compressed representation for an object ID on two different processes will typically not be the same because the process ID table entries will not be in the same order.

A table of process IDs is used to interpret the process-ID-table-pointers in compressed object names. In a world model, the process-ID-table-pointers are indexes into this table. In a message, the part of the whole table that is needed in order to understand the compressed object names in the message is sparsely represented as a vector of process-ID-table-pointer/process-ID pairs. Note that in a message containing several object descriptions there need only be one unified vector of process-ID-table-pointer/process-ID pairs.

The above promotes efficiency because typically, the world model (or a message) will refer to many objects, but these objects will share only a small number of process IDs. In memory, everything is the same as now, still allowing block copying when messages are sent, with only a small overhead necessitated by the process-ID table.

The cost of the above is centered on message receipt when compressed IDs have to be translated to correspond to the receiver's process-ID table and to a lesser degree when messages are sent and the appropriate sparse process-ID table in each message has to be created.

The GUIDs above are designed so that it is possible to use one indefinitely without having to worry about name collisions. However, it is pragmatically important not to do so.

In particular, the benefits of the compression scheme above depends critically on the assumption that almost all the names owned by a given process have the same process ID.

If, in the extreme, every object had a different process ID, then the compression scheme would actually increase total memory usage slightly. If names were used permanently, then as the days wore on, the ratio of process IDs to names in use would relentlessly rise toward 1.0 with unfortunate consequences. Rather than let this happen, one should take the opportunity to remove old objects and create new ones with new names from currently active name spaces whenever possible.

Note that beacons provide a conveniently method for making a persistent mark. Because they use URLs, they provide an infinite number of tags. These URLs take up a significant amount of space. However, this space is only used when a persistent tag is actually needed.

Process IDs are composed of 80 bits in order to allow the following basic scheme for generating them based on Internet address. The uniqueness of process IDs follows from the uniqueness of Internet addresses.

Process IDs are created by concatenating an Internet Address uniquely identifying a machine (32 bits but slated soon to become 128 bits), a port number uniquely identifying a process on the machine (16 bits), and a generation counter uniquely identifying the moment the process started (32 bits). A free port number is obtained when the process starts. The generation counter is obtained by estimating the time in seconds. Absent a reliable method of obtaining the time, some file based method must be used to obtained guaranteed unique counter values.

15.9 Class (Fundamental)

```
public spClass Class; //* [sp] readonly

      sp spGetClass(sp Object)

      void spiSetClass(sp Object, sp X)

      sp sqGetClass(sp Object)
      sp sqGetOldClass(sp Object)
      void sqSetClass(sp Object, sp X)
```

Each shared object has a shared instance variable called its *Class* that contains an spClass object describing the local and shared instance variables associated with the object.

The Class of an object is set when it is initially created and cannot change after that time. Information about the Class is shared between processes.

15.10 Owner (Fundamental)

```
public int Owner; //* [spName]

      spName spGetOwner(sp Object)
      spName spGetOldOwner(sp Object)
      void spSetOwner(sp Object, spName X)

      spName sqGetOwner(sp Object)
      spName sqGetOldOwner(sp Object)
      void sqSetOwner(sp Object, spName X)
```

An important feature of the API is that every shared object has a single owner and the owning activity is the only process that can modify any of the shared data associated with the object. This avoids readers/writers conflicts when using shared objects. The only exception to the above is that a process can create an spAction object that can run in other processes and remotely modify objects that are owned by the same process as the spAction.

Each shared object has a shared instance variable called its *Owner* that contains the unique ID of the activity that owns it. The name space of owner IDs is the same as the name space of object IDs. Owner IDs are created using the function spWMGenerateOwner. To determine whether you are the owner of an object X perform the test:

```
spGetOwner(X) == spWMGetMe()
```

The Owner of a shared object is set to the value of spWMGetMe when the object is created. After that time, the Owner of an object is free to change the Owner to a different owner ID, thereby giving up ownership of the object to another process. A process must be careful to set the Owner to a valid owner ID. If not, the object will be irrecoverably lost in limbo, until it eventually times out. Information about the Owner is shared between processes.

A process can request that ownership of an object be transferred to it by creating an spOwnershipRequest object (Section 47). The owner may or may not satisfy the request.

15.11 Locale (Fundamental and Internal)

```
public spLocale Locale; //* [sp] internal

      sp spiGetLocale(sp Object)
      sp spiGetOldLocale(sp Object)
      void spiSetLocale(sp Object, sp X)

      sp sqGetLocale(sp Object)
      sp sqGetOldLocale(sp Object)
      void sqSetLocale(sp Object, sp X)
```

The locale (Section 26) an object is in depends on the Parent of the object. If the Parent is a locale, then the object is in that locale. Otherwise the object is in the same locale as its Parent. (If there is no Parent then the object is not in any locale and information about it will not be communicated to any other process.)

In the interest of efficiency, the system caches the locale an object is in using a local instance variable called the *Locale*. This cache is incrementally maintained whenever the Parent of an object (or any of its ancestors) is changed. It must not be altered in any other way. Information about the locale is shared between processes.

This variable is shared, because it is maintained by the system in the process that owns an object and used by other processes. In particular, it may be the case that at a given moment, the owner of an object and another process might disagree about the Parent of an object. (For example, if the other process has not yet obtained information about the Parent.) In that situation, the other process simply believes what the owner says about the locale the object is in, rather than computing for itself what locale the object is in.

The system core running in the process that owns an object computes the Locale value and keeps it up to date due to changed Parents and changing Locales of Parents. The Locale controls what addresses are used when sending messages. Receiving processes don't compute Locale information, but rather just believe whatever they are told via this variable. (A side benefit of this is that for every object that is in a locale, every process always knows which locale it is in. Therefore, there are never any objects in limbo merely because a receiver has not yet found out what locale they are in.)

Whenever the value of the Locale variable changes, a message about the new state of the object is sent both to the old locale and to the new locale. The message to the old locale can be differential. The message to the new locale must be full. Processes that are interested in only the old, but not the new locale, purge the object from their world models when they process this message. Processes that are interested only in the new locale but not the old, hear about the object for the first time when it enters the locale. They then create the object in their world model copies. Processes that are interest in both the old and new locales merely change which locale the object is in. (These processes receive two messages, one differential and one full, specifying the same information.)

15.12 SharedBits (Internal)

```
public int SharedBits; //* [long] internal

      long spiGetSharedBits(sp Object)
      long spiGetOldSharedBits(sp Object)
      void spiSetSharedBits(sp Object, long X)

      long sqGetSharedBits(sp Object)
      long sqGetOldSharedBits(sp Object)
      void sqSetSharedBits(sp Object, long X)
```

Instance variables include a number of boolean variables. For efficiency, the boolean variables that are shared between processes are stored together in a single multi-bit shared variable called the *SharedBits*. The meanings of the individual bits are described elsewhere. They should never be manipulated as a group, but rather only by using the appropriate accessors for individual bits. Information about the SharedBits is shared between processes.

15.13 Parent

```
public sp Parent; //* [sp]

      sp spGetParent(sp Object)
      sp spGetOldParent(sp Object)
      void spSetParent(sp Object, sp X)

      sp sqGetParent(sp Object)
      sp sqGetOldParent(sp Object)
      void sqSetParent(sp Object, sp X)
```

A key feature of shared objects is that they can be hierarchically arranged. For example, an articulated humanoid figure might be constructed so that the top level object corresponds to the torso. The torso might have a head, two upper arms, and two thighs attached to it. Each upper arm might have a lower arm attached to it and so on.

Hierarchical relationships between shared objects are represented by using a shared instance variable called the *Parent*. In the example above, the lower arms of the humanoid figure would have as Parents the corresponding upper arms which would have as their Parents the torso.

An equally important feature of the Parent of an object is that it determines what locale (Section 26) the object is in. This in turn determines how the object is communicated, because information about an object is sent only to the locale it is in. If the Parent of an object is Null, information about it will not be sent anywhere.

There are no explicit access functions for the inverse of the Parent relationship. However, this information can be indirectly obtained using the functions spExamineChildren (Section 15.31) and spExamineDescendants (Section 15.32).

The Parent of a shared object must be another shared object, or Null, meaning that there is no Parent. When an object is created, its Parent is initialized to Null. Information about the Parent is shared between processes.

15.14 IsRemoved (Fundamental)

```
public boolean IsRemoved; //* [spBoolean] readonly

      spBoolean spGetIsRemoved(sp Object)
      spBoolean spGetOldIsRemoved(sp Object)

      void spiSetIsRemoved(sp Object, spBoolean X)

      spBoolean sqGetIsRemoved(sp Object)
      spBoolean sqGetOldIsRemoved(sp Object)
      void sqSetIsRemoved(sp Object, spBoolean X)
```

The shared instance variable *IsRemoved* has the value True if and only if the object has been removed by its owner. It is still permissible to look at the values of the instance variables of an object after the object has been removed. (This is important in alerters.) However, one has to be very careful about retaining pointers to objects that have been removed, because the system frees the storage corresponding to an object soon after it is removed.

The IsRemoved bit is set to False when an object first appears in the world model. It is set to True when the object is removed. The IsRemoved bit must never be altered in any other way. Information about the IsRemoved bit is shared between processes.

By comparing the current and old values of the IsRemoved bit, one can determine whether an object was recently removed. In particular, if the current value is True while the old value is still False, then the object was removed since alerters were last run on the object.

15.15 ForceReliable (Fundamental and Internal)

```
public boolean ForceReliable; //* [spBoolean] internal

      spBoolean spiGetForceReliable(sp Object)
      spBoolean spiGetOldForceReliable(sp Object)
      void spiSetForceReliable(sp Object, spBoolean X)

      spBoolean sqGetForceReliable(sp Object)
      spBoolean sqGetOldForceReliable(sp Object)
      void sqSetForceReliable(sp Object, spBoolean X)
```

The shared instance variable *ForceReliable* controls whether an object is communicated using TCP. When ForceReliable is True, the communication of the object using TCP via the appropriate Locale-Based Communication server is forced. Otherwise, it is not. See the documentation of ISTP.

The ForceReliable bit has a default value of False. Information about the ForceReliable bit is shared between processes.

15.16 InhibitReliable (Fundamental and Internal)

```
public boolean InhibitReliable; //* [spBoolean] internal

      spBoolean spiGetInhibitReliable(sp Object)
      spBoolean spiGetOldInhibitReliable(sp Object)
      void spiSetInhibitReliable(sp Object, spBoolean X)

      spBoolean sqGetInhibitReliable(sp Object)
      spBoolean sqGetOldInhibitReliable(sp Object)
      void sqSetInhibitReliable(sp Object, spBoolean X)
```

The shared instance variable *InhibitReliable* controls whether the appropriate Locale-Based Communication server sends out positive acknowledgments so that processes can determine whether they have the correct information about an object. When InhibitReliable is True, the sending of these acknowledgments is blocked. This is appropriate when the object is changing so rapidly that there is no point in trying to retransmit lost information. See the documentation of ISTP.

The InhibitReliable bit has a default value of False. Information about the InhibitReliable bit is shared between processes.

15.17 LocalBits (Internal)

```
transient public short LocalBits; //* [short] internal

      short spiGetLocalBits(sp Object)
      void spiSetLocalBits(sp Object, short X)

      short sqGetLocalBits(sp Object)
      void sqSetLocalBits(sp Object, short X)
```

Instance variables include a number of boolean variables. For efficiency, the local boolean variables of shared objects are stored together in a single multi-bit local variable called the *LocalBits*. The meanings of the individual bits are described elsewhere. They should never be manipulated as a group, but rather only by using the appropriate accessors for individual bits. Information about the LocalBits is maintained separately in each process.

15.18 IsNew (Fundamental)

```
transient public boolean IsNew; //* [spBoolean] readonly

      spBoolean spGetIsNew(sp Object)

      void spiSetIsNew(sp Object, spBoolean X)

      spBoolean sqGetIsNew(sp Object)
      void sqSetIsNew(sp Object, spBoolean X)
```

The local instance variable *IsNew* has the value True if and only if the object appeared in the local world model copy since the last time alerters were run on that kind of object. The IsNew bit is only intended to be used by alerters and has little value in other situations.

The IsNew bit is set to True when an object first appears in the world model. It is set to False at the end of the next call on spWMUpdate. Information about the IsNew bit is maintained separately in each process.

Note that if an object has just newly appeared, then the old values of its shared instance variables contain the values that the shared instance values had at the moment the object appeared.

15.19 AppData

```
transient public int AppData; //* [void *]

      void * spGetAppData(sp Object)
      void spSetAppData(sp Object, void * X)

      void * sqGetAppData(sp Object)
      void sqSetAppData(sp Object, void * X)
```

Every shared object has a local instance variable called *AppData*, which applications can use to store any 32-bit quantity they want to. This can be very convenient in many situations. However, note that if you store pointers to dynamically allocated structures, you must carefully monitor the removal of objects (e.g., with a callback) or risk having a memory leak due to the allocated structures not getting freed when objects are removed. (Since the system does not understand anything about what is in this variable, it cannot take any action to prevent memory leaks.) In addition, if two applications running on the same machine and sharing the same world model copy both try to use this variable, severe problems will arise unless they cooperate closely.

When an object is created, the AppData value is set to Null. Applications can do whatever they want with it after that time. Information about the AppData is maintained separately in each process.

15.20 MessageNeeded (Fundamental and Internal)

```
transient public boolean MessageNeeded; //* [spBoolean] internal

      spBoolean spiGetMessageNeeded(sp Object)
      void spiSetMessageNeeded(sp Object, spBoolean X)

      spBoolean sqGetMessageNeeded(sp Object)
      void sqSetMessageNeeded(sp Object, spBoolean X)
```

When a process modifies the shared data associated with an object, a message containing the new state of the object's shared data is sent to other processes. Detailed control over when a message is sent is exercised via a local boolean instance variable called the *MessageNeeded* bit. When (and only when) the MessageNeeded bit is True for an object owned by the local process, a message is sent describing the state of the object and the MessageNeeded bit is set back to False. Information about the MessageNeeded bit is maintained separately in each process.

The MessageNeeded bit is set to False when an object is initially created. Whenever one of the external API functions for modifying shared instance variables (such as spSetParent) is used, the function sets the MessageNeeded bit to True. Due to this automatic maintenance of the MessageNeeded bit, the typical application writer need not be concerned with the MessageNeeded bit.

However, those that extend the system need to think carefully about when the MessageNeeded bit should be set. The reason for this is that the fast internal shared data modification functions (such as sqSetParent) do not set the MessageNeeded bit. If the MessageNeeded bit should be set, then it must be set separately.

One reason for not always setting the MessageNeeded bit is that several shared instance variables of an object may be being modified at once and therefore there is no need to set the MessageNeeded bit more than once.

Another reason for not setting the MessageNeeded bit is that when an action modifies an object, this typically should not trigger the sending of a message, because the basic idea behind actions is to have them exercise remote control over objects in lieu of sending messages.

15.21 Change (Fundamental and Internal)

```
transient public boolean Change; //* [spBoolean] internal

      spBoolean spiGetChange(sp Object)
      void spiSetChange(sp Object, spBoolean X)

      spBoolean sqGetChange(sp Object)
      void sqSetChange(sp Object, spBoolean X)
```

The processing of alerters (Section 43) is triggered by observing that the shared data of a shared object has changed. Since changes in shared data are relatively rare events (on the scale of time that the system checks whether alerters should be applied) special steps are taken to make it very efficient to detect whether a change has, or has not, occurred. In particular, each shared object has a local instance variable called the *Change* bit that specifies whether there has been any change in the shared data since the last time alerters were run on the object. Alerter processing is applied to an object only when the Change bit is True. The Change bit is then set back to False. Information about the Change bit is maintained separately in each process.

The Change bit is set to False when an object is initially created. Whenever one of the external API functions for modifying shared instance variables (such as spSetParent) is used, the function sets the Change bit to True. Similarly, whenever a message arrives specifying a new state for an object, the Change bit is also set to True. Due to this automatic maintenance of the Change bit, the typical application writer need not be concerned with the Change bit.

However, those that extend the system need to think carefully about when the Change bit should be set. The reason for this is that the fast internal shared variable modification functions (such as sqSetParent) do not set the Change bit. If the Change bit should be set, then it must be set separately.

One reason for not always setting the Change bit is that several shared instance variables of an object may be being modified at once and therefore there is no need to set the Change bit more than once. Note that actions set the Change bit even though they do not set the MessageNeeded bit so that alerters will be correctly triggered.

15.22 OldPtr (Fundamental and Internal)

```
transient public int OldPtr; //* [void *] internal

      void * spiGetOldPtr(sp Object)
      void spiSetOldPtr(sp Object, void * X)

      void * sqGetOldPtr(sp Object)
      void sqSetOldPtr(sp Object, void * X)
```

Shared objects are identified by a pointer to the part containing shared values. For rapid access to the old shared data, a local instance variable called the *OldPtr* contains a pointer to the part of the object containing the old data.

The OldPtr of a shared object is initialized when the object is created and must never by changed. Information about the OldPtr is maintained separately in each process.

15.23 JavaPtr (Fundamental and Internal)

```
transient public int JavaPtr; //* [void *] internal

      void * spiGetJavaPtr(sp Object)
      void spiSetJavaPtr(sp Object, void * X)

      void * sqGetJavaPtr(sp Object)
      void sqSetJavaPtr(sp Object, void * X)
```

As part of the support for the Java API, each shared object has a local instance variable called the *JavaPtr*, which points to the corresponding Java object, if any.

This pointer is maintained by the interface between the system and Java and must not be modified in any other way. Information about the JavaPtr is maintained separately in each process.

15.24 Referrers (Fundamental and Internal)

```
transient public int Referrers; //* [spNameInfoPtr] internal

      spNameInfoPtr spiGetReferrers(sp Object)
      void spiSetReferrers(sp Object, spNameInfoPtr X)

      spNameInfoPtr sqGetReferrers(sp Object)
      void sqSetReferrers(sp Object, spNameInfoPtr X)
```

Each shared object X has a local instance variable called the *Referrers* that contains a list of every other object that refers to X. Each list entry specifies an object that points to X and which instance variable of the object points to X. For instance, if 12 objects in the world model refer to X and one of these objects refers to X from two different places, then the referrers list of X will contain 13 entries indicating exactly how the 12 objects refer to X. One use of this reverse index is to rapidly determine which objects are the children of a given object X by determining which objects refer to X as their Parent.

The Referrers index is incrementally maintained by the functions (such as spSetParent) that alter instance variables pointing to shared objects. It is also changed when messages are received describing a new state for an object. It is essential that referrers lists not be altered in any other way. Information about the Referrers is maintained separately in each process.

15.25 Alerters (Fundamental and Internal)

```
transient public int Alerters; //* [void *] internal

      void * spiGetAlerters(sp Object)
      void spiSetAlerters(sp Object, void * X)

      void * sqGetAlerters(sp Object)
      void sqSetAlerters(sp Object, void * X)
```

As part of the support for alerters (Section 43) the system maintains indexes of what alerters are monitoring which objects. In particular, each shared object X has a local instance variable called *Alerters* that contains a list of all the alerters that are monitoring X. If X is not a class, then the alerters are specifically monitoring X as opposed to any other object. If X is a class, then the alerters monitor every object that is an instance of X or any subclass of X. These indexes are incrementally maintained as alerters are created and removed. Information about the Alerters list is maintained separately in each process.

15.26 Msgs (Fundamental and Internal)

```
transient public int Msgs; //* [void *] internal

      void * spiGetMsgs(sp Object)
      void spiSetMsgs(sp Object, void * X)

      void * sqGetMsgs(sp Object)
      void sqSetMsgs(sp Object, void * X)
```

The *Msgs* local instance variable of an object contains a list of messages about the object that are waiting to be processed (Section 1.5.6). The list is used to ensure that messages are processed in order even when they arrive out of order. The Msgs variable is manipulated by the system core and must not be altered by an application process. Information about the Msgs list is maintained separately in each process.

15.27 LastUpdateTime (Fundamental and Internal)

```
transient public int LastUpdateTime; //* [spTimeStamp] internal

      spTimeStamp spiGetLastUpdateTime(sp Object)
      void spiSetLastUpdateTime(sp Object, spTimeStamp X)

      spTimeStamp sqGetLastUpdateTime(sp Object)
      void sqSetLastUpdateTime(sp Object, spTimeStamp X)
```

The *LastUpdateTime* local instance variable of an object contains the most recent time at which a message about the object was processed. This is only relevant to objects that are not owned by the local process. Among other things, this value is used to determine whether an unreasonably long time has elapsed without getting any messages about the object. The LastUpdateTime variable is manipulated by the system core and must not be altered by an application process. Information about the LastUpdateTime is maintained separately in each process.

15.28 spNew

```
sp spNew()
```

Return value - The newly constructed object.

With regard to constructing objects, shared classes fall into three categories. A couple of classes (such as sp and spLinking) are *abstract* in the sense that they merely group together some useful operations or specify a pattern that must be specialized before it can be used. It is not possible to construct instances of these classes. (As a result, the function spNew does not actually exist. It is described here in order to present what New functions must look like for shared classes that are not abstract.)

For most classes spK, instances are created using an automatically generated zero-argument constructor named spKNew. When this is the case, no special comment is made in this documentation beyond noting the existence of the constructor. The way spKNew is implemented is shown below.

```
sp spKNew() {
  return spClassNewObj(spKC(), 0);
}
```

Some classes (for example spLocale) have constructors that take arguments. In this case, the constructor is not automatically generated. Nevertheless, it must begin by calling spClassNewObj as in the automatically generated constructor shown above.

Whenever a special New function exists, this document contains a subsection describing it. A key reason for having constructors with arguments is that a number of classes have instance variables that must be set at the moment an object is created and cannot be changed later.

15.29 spInitialization

```
void spInitialization(sp Object)
```

Object - Object to be initialized.
Return value - There is no return value.

Initializes the values of instance variables. Whenever an instance of a shared class is created, by a New function, The values of the instance variables are all set to all bits zero. After this, the Initialization functions are called for the class and all its ancestors in order to properly initialize any variables that need to have non-zero values. The Initialization functions are called starting with spInitialization and then working down to the most specific class so that the Initialization function for a class can override the Initialization function(s) for its superclass(es).

In order to reduce the number of Initialization functions that have to be written, the API is designed so that as much as possible, all bits zero is an appropriate initial value. The only major exception to this is that spTransforms (Section 8) are initialized to the identity transform, which is not all zero. Wherever a non-zero initialization is needed, this fact is pointed out in this documentation.

Whenever the a new shared class is defined that requires a variable to be initialized to something other than zero, an Initialization function must be defined for the class. If no Initialization function is defined, then the system automatically generates one that does nothing.

15.30 spRemove (Fundamental)

```
void spRemove(sp Object)
```

Object - The object to be removed.
Return value - There is no return value.

Removes an object from the world model. The exact behavior depends on whether the object is or is not owned by the local process. If an object X is removed by the process that owns it, then the IsRemoved bit is set to True in X, every field in X that points to another object is set to Null, every field in every other object that points to X is set to Null, and these changes are communicated to all the other processes that know about the object, causing the object to be removed in these processes as well. (One must be very careful when accessing objects that may have been removed (Section 1.5.3).)

In contrast if an object is removed by a process that does not own it, then X is taken out of the world model without changing any fields and without communicating anything to other processes. That is so say, the world model is simply put into the state that it would have been in if it had never heard about X. In particular, if the process later hears about the object again, it will be restored to the world model exactly as it was, with all references intact.

Removal is different from freeing the storage for an object. Freeing happens separately and not necessarily immediately. Removal of an object by its owner signals the explicit desire to remove an object from the world model and therefore from the view of other processes. It is essential that it occur immediately, not at some later time.

15.31 spExamineChildren

```
void spExamineChildren(sp Object, spMask Mask, spFn F, void * Data)
```

Object - The object whose children are to be examined.
Mask - spMask (Section 7) limiting the objects considered.
F - Operation (Section 6) applied to children of the object.
Data - Passed to the operation each time it is called (modifiable).
Return value - There is no return value.

Applies an operation to all the direct children of an object that are compatible with the Mask. (An object C is a child of an object P if and only if P is the Parent of C.) There are no guarantees as to the order in which the children will be accessed.

There is no explicit representation of the children of an object, and no direct way to access the children of an object. However, the indirect approach provided by spExamineChildren is typically at least as convenient as some more direct approach would be.

15.32 spExamineDescendants

```
void spExamineDescendants(sp Object, spMask Mask, spFn F, void * Data)
```

Object - The object whose descendants are to be operated on.
Mask - spMask (Section 7) limiting the objects considered.
F - Operation (Section 6) applied to descendants of the object.
Data - Passed to the operation each time it is called (modifiable).
Return value - There is no return value.

Applies an operation to all the children of an object that are compatible with the Mask and then to all their children that are compatible with the Mask and so on. The order of application is undefined except that the operation will be applied to a given descendent D before it is applied to any of the descendants of D.

15.33 spTopmost

```
sp spTopmost(sp Object)
```

Object - Thing whose ancestor is to be located.
Return value - Highest level ancestor.

Computes the highest level ancestor of an object that defines the coordinate system the object is in. This value is often useful when using operations like spPositioningRelativeMatrix (Section 16.9). If the object is contained in an spLocale, then this locale is returned. If the object is not in any locale, then Null is returned.

15.34 spPrint (Fundamental)

```
void spPrint(sp Object)
```

Object - The object to print.
Return value - There is no return value.

Prints out the contents of all the instance variables of a shared object. This is very useful for debugging.

15.35 spLocallyOwned (Fundamental and Internal)

```
spBoolean spLocallyOwned(sp Object)
```

Object - Object whose ownership is being tested.
Return value - True of object locally owned.

Returns True if and only if the object in question is owned by the local process. That is to say, it checks whether the process ID of the owner ID is associated with the local process. This can be done quickly because the local process knows exactly what process IDs it has generated.

15.36 spSetParent (Fundamental and Internal)

```
void spSetParent(sp Object, sp Parent)
```

Object - Object whose Parent is to be changed.
Parent - New Parent.
Return value - There is no return value.

Sets the Parent of an object. This is basically just the standard setting method for the Parent instance variable of all sp objects. However, a special method has to be provided because special internal processing has to be applied when the Parent of an object changes. This need not concern application programmers.

16 spPositioning

```
public abstract class spPositioning extends sp
```

spPositioning groups together data and functions that are related to positions and motion.

The shared class spPositioning inherits all the instance variables and functions of the class sp (Section 15). The class spPositioning defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spPositioningC - Class descriptor (Section 15.1).
shared **Transform** - Position and orientation (Section 16.1).
Matrix - Matrix corresponding to Transform (Section 16.2).
MatrixOK - Indicates cached matrix is accurate (Section 16.3).
MatrixInverse - Matrix corresponding to inverse of Transform (Section 16.4).
MatrixInverseOK - Indicates cached matrix inverse is accurate (Section 16.5).

The class spPositioning defines the following functions:

spPositioningMatrix - Computes matrix for transform (Section 16.6).
spPositioningMatrixInverse - Computes inverse of matrix for transform (Section 16.7).
spPositioningLocalize - Puts object in appropriate locale (Section 16.8).
spPositioningRelativeMatrix - Computes matrix relating two objects (Section 16.9).
spPositioningRelativeVector - Computes vector between two objects (Section 16.10).
spPositioningDistance - Computes distance between two objects (Section 16.11).
spPositioningLookAt - Orients object to look at another (Section 16.12).
spPositioningGoThru - Moves smoothly through position (Section 16.13).
spPositioningStopAt - Moves smoothly to position and stops (Section 16.14).
spPositioningStop - Stops object where it is. (Section 16.15).
spPositioningFollowPath - Moves smoothly along a predefined path (Section 16.16).
spPositioningMotionTimeLeft - Computes duration of queued smooth motion (Section 16.17).
spPositioningGetMotionQueue - Obtains smooth motion queue as path (Section 16.18).
spPositioningFlushMotionQueue - Discards remainder of smooth motion queue (Section 16.19).
spPositioningSetTransform - Sets Transform (Section 16.20).
spPositioningInitialization - Initializes Transform (Section 16.21).

It is not possible to create instances of the class spPositioning. Rather, one can only create instances of particular subclasses of the class spPositioning.

16.1 Transform

```
public float[] Transform; //* [spTransform:17]

      spTransform spPositioningGetTransform(sp Object)
      spTransform spPositioningGetOldTransform(sp Object)
      void spPositioningSetTransform(sp Object, spTransform X)

      spTransform sqPositioningGetTransform(sp Object)
      spTransform sqPositioningGetOldTransform(sp Object)
      void sqPositioningSetTransform(sp Object, spTransform X)
```

The fundamental representation of the position, orientation, and scaling of an spPositioning is an spTransform (Section 8). This is stored in a shared instance variable called the *Transform*. This form of representation is chosen because it is easy to understand and each feature is represented separately, so that individual features are easy to change without effecting any other feature.

The Transform of an spPositioning is interpreted relative to the coordinate system of the object's Parent. For instance, if the head of a humanoid figure is represented as a child of the torso, then the Transform of the head specifies the position, orientation and scaling of the head relative to the torso. This is convenient both because it makes it easy to move the head relative to the torso and because it makes it possible to move and scale the torso and head together relative to the surrounding world without modifying the Transform of the head.

Note that the Transform effects many features of subclasses of spPositioning. For instance, it effects the appearance of an spDisplaying and other pieces of information as well, such as the size of the units of the InRadius.

The Transform is of type spTransform. When an spPositioning object is created, its Transform is initialized to a value that indicates, no translation, no rotation, and no scaling. It can be changed whenever desired later. Information about the Transform is shared between processes.

The value returned by spPositioningGetTransform shares memory with the spPositioning itself and in general, cannot be retained across a call to spWMUpdate (Section 1.4).

16.2 Matrix (Internal)

```
transient public float[] Matrix; //* [spMatrix:16] internal

      spMatrix spPositioningiGetMatrix(sp Object)
      void spPositioningiSetMatrix(sp Object, spMatrix X)

      spMatrix sqPositioningGetMatrix(sp Object)
      void sqPositioningSetMatrix(sp Object, spMatrix X)
```

The *Matrix* local instance variable of an spPositioning stores the most recent value returned by spPositioningMatrix. It is set to all zeros when an spPositioning is initially created. Information about the Matrix is maintained separately in each process.

16.3 MatrixOK (Internal)

```
transient public boolean MatrixOK; //* [spBoolean] internal

      spBoolean spPositioningiGetMatrixOK(sp Object)
      void spPositioningiSetMatrixOK(sp Object, spBoolean X)

      spBoolean sqPositioningGetMatrixOK(sp Object)
      void sqPositioningSetMatrixOK(sp Object, spBoolean X)
```

The *MatrixOK* local instance variable of an spPositioning specifies whether the Matrix value accurately corresponds to the current Transform value. It is set to False when an spPositioning is initially created. It is set to True whenever spPositioningMatrix computes a new matrix value. It is set back to False whenever the Transform of an spPositioning is modified. Special care must be taken to ensure that this happens when the Transform is modified by an spAction or by the receipt a message that modifies the spPositioning. Information about the MatrixOK bit is maintained separately in each process.

16.4 MatrixInverse (Internal)

```
transient public float[] MatrixInverse; //* [spMatrix:16] internal

      spMatrix spPositioningiGetMatrixInverse(sp Object)
      void spPositioningiSetMatrixInverse(sp Object, spMatrix X)

      spMatrix sqPositioningGetMatrixInverse(sp Object)
      void sqPositioningSetMatrixInverse(sp Object, spMatrix X)
```

The *MatrixInverse* local instance variable of an spPositioning stores the most recent value returned by spPositioningMatrixInverse. It is set to all zeros when an spPositioning is initially created. Information about the MatrixInverse is maintained separately in each process.

16.5 MatrixInverseOK (Internal)

```
transient public boolean MatrixInverseOK; //* [spBoolean] internal

      spBoolean spPositioningiGetMatrixInverseOK(sp Object)
      void spPositioningiSetMatrixInverseOK(sp Object, spBoolean X)

      spBoolean sqPositioningGetMatrixInverseOK(sp Object)
      void sqPositioningSetMatrixInverseOK(sp Object, spBoolean X)
```

The *MatrixInverseOK* local instance variable of an spPositioning specifies whether the Matrix-Inverse value accurately corresponds to the current Transform value. It is set to False when an spPositioning is initially created. It is set to True whenever spPositioningMatrixInverse compues a new inverse matrix value. It is set back to False whenever the Transform of an spPositioning is modified. Special care must be taken to ensure that this happens when the Transform is modified by an spAction or by the receipt a message that modifies the spPositioning. Information about the MatrixInverseOK bit is maintained separately in each process.

16.6 spPositioningMatrix

```
spMatrix spPositioningMatrix(sp Object)
```

Object - Object for which matrix is being calculated.
Return value - spMatrix corresponding to transform of spThing.

The fundamental representation of the position, orientation, and scaling of an spPositioning is an spTransform. However, in many computational situations, it is more convenient to manipulate this information when represented as a 4x4 transformation matrix (Section 12). The function spPositioningMatrix computes the spMatrix corresponding to the Transform of an spPositioning. Since this is relatively expensive to compute, the result is cached so that it only has to be recomputed when the Transform changes.

The value returned by spPositioningMatrix shares memory with the spPositioning itself and in general, cannot be retained across a call to spWMUpdate (Section 1.4).

There is no operation for directly setting the Transform of an spPositioning from an spMatrix. Rather one should use spPositioningSetTransform and spMatrixToTransform. This makes the expense of this operation more obvious to the application writer. As much as possible, it is wise to operate solely in terms of spTransforms.

16.7 spPositioningMatrixInverse

```
spMatrix spPositioningMatrixInverse(sp Object)
```

Object - Object for which inverse matrix is being calculated.
Return value - Inverse of spMatrix corresponding to transform of spThing.

Computes the inverse of the spMatrix corresponding to the Transform of an spPositioning. Since this is relatively expensive to compute, the result is cached so that it only has to be recomputed when the Transform changes.

The value returned by spPositioningMatrixInverse shares memory with the spPositioning itself and in general, cannot be retained across a call to spWMUpdate (Section 1.4).

16.8 spPositioningLocalize

```
spBoolean spPositioningLocalize(sp Object, sp Destination, spBoolean ChooseSmallest)
```

Object - The object to be put in an appropriate locale.
Destination - The locale the object is to be placed in.
ChooseSmallest - True forces positioning in smallest containing locale.
Return value - True if the object changes locale.

Places an spPositioning object in an appropriate locale (Section 26). The object must have as its direct Parent a locale L. The object is either left in L or placed in one of the locales *neighboring* (Section 26.1) L. The determination of where to put the object is done in one of three modes by calling spLocaleChoose.

(1) If the Destination argument is not Null, then the object is placed in this locale, which must either equal L or neighbors L.

(2) If the Destination argument is Null and the ChooseSmallest argument is True, then L and all the locales neighboring L are checked to see which ones have boundaries that encompass the location of the object. If any of these locales contain the object, then the object is placed in the containing locale whose boundary has the smallest Volume.

(3) If the Destination argument is Null and the chooseSmallest argument is False, then the system first checks whether the object is inside L. If it is, no further action is taken. If not, a new locale is found for the object as in mode (2).

(The reason why this third mode of behavior is included is that it is usually sufficient and is a great deal more efficient than the second mode of operation. The reason that the second mode of operation is included is that the third mode will never move an object from a locale L into another locale that is contained within L.)

No matter how a new locale for the object is chosen, the new locale is made the Parent of the object. True is returned if object changes locale.

If the Parent the object changes from one locale to another, the Transform of the object is adjusted based on the export transform relating the two locales. The fact that spPositioningLocalize automatically performs this adjustment is an important part of its value. This adjustment involves conversion between spTransform and spMatrix objects. As a result, there can be problems with rotational ambiguity (Section 10.1).

Whenever an object whose Parent is a locale moves in such a way that it might have moved out of the boundary of its locale, spPositioningLocalize should be called with a locale argument of Null to make sure that the object ends up in an appropriate locale. To teleport someplace, you can just set the parent link and matrix of an object yourself to some particular position in some locale.

For the convenience of the application writer, spPositioningLocalize is automatically called every cycle on every locally owned spRoot object. This is done with the Destination argument Null and the ChooseSmallest argument False. In general, application writers can control when spPositioningLocalize is called merely be deciding which objects should be spRoot objects. The typical application need not contain any calls on spPositioningLocalize.

16.9 spPositioningRelativeMatrix

```
spMatrix spPositioningRelativeMatrix(sp X, sp Object, spMatrix Matrix)
```

X - spPositioning object specifying reference coordinate system.
Object - spPositioning object whose relative position is to be computed.
Matrix - spMatrix in which to store result.
Return value - Modified spMatrix.

Computes an spMatrix that specifies the position, orientation, and scaling of an spPositioning object in the coordinate system of another object X. If X is the Parent of the object, then this is just the matrix of the object. If is an ancestor of the object, then several matrices have to be multiplied together. If the object and have a more complex relationship, then some matrices have to be inverted. If the object and are in different locales, then the relationships between the two locales have to be taken into account, by importing the object into 's locale. If the locales are not immediate neighbors, then the relationship between the locales may be ambiguous non existent. In either of these cases, and an error is posted and Null is returned. The result is stored in the Matrix argument which is returned.

16.10 spPositioningRelativeVector

```
spVector spPositioningRelativeVector(sp Object, sp X, spVector Vector)
```

Object - spPositioning object whose relative position is to be computed.
X - spPositioning object specifying reference coordinate system.
Vector - spVector in which to store result.
Return value - Modified spVector.

Computes the translation portion of the matrix computed by spPositioningRelativeMatrix, if any. If there is a value, it is stored in the Vector argument and returned. Otherwise an error is posted and Null is returned.

16.11 spPositioningDistance

```
float spPositioningDistance(sp Object, sp X)
```

Object - spPositioning object whose relative position is to be computed.
X - spPositioning object specifying reference coordinate system.
Return value - Distance between objects.

Returns the length of the vector computed by spPositioningRelativeVector, if any. If there is no vector, then an error is posted and -1 is returned.

16.12 spPositioningLookAt

```
void spPositioningLookAt(sp Object, sp Target)
```

Object - spPositioning object to be oriented looking at another.
Target - spPositioning to be looked at.
Return value - There is no return value.

Modifies the Transform of an spPositioning object so that the object is looking at another spPositioning object Target. That is to say, the rotational orientation of the object is altered so that the negative Z axis of the object's coordinate system points to the origin of Target's coordinate system. In addition, the Y axis of the object is oriented so that it and the Y axis of the highest level coordinate system containing the object are co-planar. If the object is an spSeeing object controlling visual rendering, this causes objects that are upright from the perspective of the highest level coordinate system containing the object to appear upright in the images generated.

16.13 spPositioningGoThru

```
void spPositioningGoThru(sp Object, spTransform Transform, spDuration Time)
```

Object - spPositioning to be moved.
Transform - Position and orientation to move to.
Time - Duration of move.
Return value - There is no return value.

Causes an spPositioning object to move smoothly so as to pass through the specified position and orientation after the specified Time in milliseconds. An action (Section 46) is used to support the motion so that a high degree of smoothness can be achieved using low communication bandwidth. The action sets the Transform of the object to an appropriate intermediate value every time spWMUpdate is called,setting the Change bit but not the MessageNeeded bit.

Multiple motion requests for a given object are queued up one after the other. If there is a motion request in the queue whose time interval has not yet been completed, then the time interval of the motion request being added is interpreted as occurring after the last request in the queue has been completed. If the queue is empty or all the requests in the queue have been completed, then the time interval is interpreted as starting at the current moment (rather than as starting at the time in the possibly distant past when the last request to be completed ended).

Smooth motion is achieved by interpolating between (and extrapolating beyond) the positions and orientations specified. (spTransforms are used because they are more suitable for interpolation than spMatrix objects.) It should be noted that while a smooth motion action is in control of an object, its activity will override any attempt to set the transform of the object directly. The smooth motion action relinquishes control of the object as soon as it comes to rest with an empty smooth motion request queue. Note that the queue can be empty, with the object still in motion, extrapolating beyond the last requested position. If you want to force an object to stop immediately, call spPositioningStop.

The accessor spPositioningSetTransform can also be used to move an spPositioning to a particular position. However, it causes the object to jump abruptly to that position, rather than moving along a smooth path from start to finish.

16.14 spPositioningStopAt

```
void spPositioningStopAt(sp Object, spTransform Transform, spDuration Time)
```

Object - spPositioning to be moved.
Transform - Position to stop at.
Time - Duration of move.
Return value - There is no return value.

Causes an spPositioning to move smoothly so as to reach and stop at the specified position and orientation after the specified time in milliseconds. spPositioningStopAt is identical to spPositioningGoThru except that the object comes to rest at the specified position instead of extrapolating beyond it once the time interval has expired.

If several motion segments are to be chained together, then you should call spPositioningGoThru. If you just want to smoothly move an object to some place, then you should call spPositioningStopAt.

16.15 spPositioningStop

```
void spPositioningStop(sp Object)
```

Object - spPositioning to stop.
Return value - There is no return value.

This causes an spPositioning to stop moving and stay wherever it currently is. This is useful to stop an object that is moving in extrapolative mode after the end of a number of smooth motion requests, without causing any abrupt change in position.

16.16 spPositioningFollowPath

```
void spPositioningFollowPath(sp Object, spPath Path)
```

Object - The object to be moved.
Path - The path to follow.
Return value - There is no return value.

Causes an object to smoothly follow the specified prerecorded path. If there is an already existing queue of smooth motion requests, then the path is appended to the end of the queue. Otherwise, it becomes the queue.

An interesting issue regarding smooth motion can be most clearly seen in the context of paths. An spTransform is only meaningful given some particular coordinate system. For simplicity and reusability, all the spTransforms in an spPath are assumed to be relative to a single coordinate system. This makes it easy to reuse a path originally designed in one locale in another locale, but special care must be taken when a path is long enough that it extends through several locales. This is dealt with by transforming the path each time a locale boundary is crossed as described below.

For all smooth motion requests, including following paths, the spTransforms specified are interpreted as being in the coordinate system of the object X being moved. In particular, all the spTransforms in the queue of smooth motion requests are relative to the coordinate system of X. When the object X is transferred from one locale to another, the Transform of X must be adjusted to take account of the change of coordinate system. The spTransforms in the queue of smooth motion requests have to be adjusted in just the same way. If the locale of X is changed using spPositioningLocalize, the Transform of X and the transforms in the smooth motion queue are all adjusted automatically. If the locale of X is changed in some other way, the adjustment of the transforms is up to the application.

Note that smooth motion itself never causes a change of locale. The application itself must call spPositioningLocalize if it thinks that changing locales might be necessary. The reason this is left up to the application is that only the application knows whether this might be necessary. In particular, many motions are confined to a single locale and the system does not want to force the overhead of calling spPositioningLocalize in such cases. If a motion might transit between locales, then the application should call spPositioningLocalize regularly (e.g., a few times a second). Exactly how often spPositioningLocalize needs to be called depends on the relationship between the speed of the motion and the size of the locales being transited.

16.17 spPositioningMotionTimeLeft

```
spDuration spPositioningMotionTimeLeft(sp Object)
```

Object - Object in motion.
Return value - Remaining milliseconds of motion.

Returns the number of milliseconds required to complete the smooth motion requests queued up for an object. That is to say, the amount of time remaining before extrapolation beyond the end of the queue will begin.

16.18 spPositioningGetMotionQueue

```
spPath spPositioningGetMotionQueue(sp Object)
```

Object - spPositioning object to get motion queue for.
Return value - Copied contents of motion queue.

Returns the current contents of the smooth motion queue in the form of an spPath. The queue is copied so that the returned path can be manipulated without having any effect on the motion of the object. (If you want to have an effect, get the queue as a path, flush the queue, alter the path, and then have the object follow the new path.)

16.19 spPositioningFlushMotionQueue

```
void spPositioningFlushMotionQueue(sp Object)
```

Object - spPositioning object whose motion queue is to be flushed.
Return value - There is no return value.

Get rid of the remaining queue of smooth motion requests for an object, while retaining the history of past motion. Flushing the queue is the appropriate thing to do if one wants to substitute a new series of smooth motion requests. The past history is retained so that a smooth transition can be made. If no new requests are specified, the object will extrapolate forward at its current velocity. (If you just want the object to stop where it is, call spPositioningStop.

Because the current position the object is at may not be an explicit part of the motion history, it is possible for a substitute series of smooth motion requests to cause an abrupt jerk when they start to take effect. This can be prevented by first specifying that the object should move to its current position in zero milliseconds. This makes the current position be part of the motion history.

16.20 spPositioningSetTransform (Internal)

```
void spPositioningSetTransform(sp Object, spTransform Transform)
```

Object - spPositioning whose Transform is to be set.
Transform - New spTransform.
Return value - There is no return value.

Sets the Transform variable in an spPositioning, doing everything a standard Set accessor would do, but also sets indicators specifying that the corresponding spMatrix must be recalculated (I.e., sets the MatrixOK and MatrixInverseOK bits to False). This accessor method is explicitly described in order to indicate that it does not have just the standard default behavior.

Note that if smooth motion is in effect, directly setting the Transform of an object, will have little effect, because it will immediately be overridden the next time a smooth motion position is calculated. If you want to start setting the Transform directly, first stop smooth motion.

16.21 spPositioningInitialization (Internal)

```
void spPositioningInitialization(sp Object)
```

Object - spPositioning object to initialize.
Return value - There is no return value.

Initializes the Transform of an spPositioning to the identity spTransform.

17 spDisplaying

```
public abstract class spDisplaying extends sp
```

spDisplaying groups together data that specifies appearance and extent.

The shared class spDisplaying inherits all the instance variables and functions of the class sp (Section 15). The class spDisplaying defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spDisplayingC - Class descriptor (Section 15.1).
shared **VisualDefinition** - Graphical model (Section 17.1).
shared **InRadius** - Contained bounding sphere size (Section 17.2).
shared **OutRadius** - Containing bounding sphere size (Section 17.3).
GraphicsNode - Corresponding scene graph node (Section 17.4).

It is not possible to create instances of the class spDisplaying. Rather, one can only create instances of particular subclasses of the class spDisplaying.

17.1 VisualDefinition

```
public spVisualDefinition VisualDefinition; //* [sp]

      sp spDisplayingGetVisualDefinition(sp Object)
      sp spDisplayingGetOldVisualDefinition(sp Object)
      void spDisplayingSetVisualDefinition(sp Object, sp X)

      sp sqDisplayingGetVisualDefinition(sp Object)
      sp sqDisplayingGetOldVisualDefinition(sp Object)
      void sqDisplayingSetVisualDefinition(sp Object, sp X)
```

An spDisplaying object has a shared instance variable called the *VisualDefinition* that specifies how it looks. The VisualDefinition is a 3D graphical model that is used by the visual renderer when generating images of the virtual world.

The VisualDefinition of an spDisplaying must be an spVisualDefinition object (Section 24) or Null, indicating that the spDisplaying has no appearance. When an spDisplaying is created, its VisualDefinition is initialized to Null. Information about the VisualDefinition is shared between processes.

17.2 InRadius

```
public float InRadius; //* [float]

      float spDisplayingGetInRadius(sp Object)
      float spDisplayingGetOldInRadius(sp Object)
      void spDisplayingSetInRadius(sp Object, float X)

      float sqDisplayingGetInRadius(sp Object)
      float sqDisplayingGetOldInRadius(sp Object)
      void sqDisplayingSetInRadius(sp Object, float X)
```

An spDisplaying object has a shared instance variable called the *InRadius* that specifies the radius of the largest sphere (positioned at the center of the object's coordinate system) that fits entirely within the object. This is included to provide a crude, but rapid, basis for applications to reason about collisions and other interactions between spDisplaying objects. Note that since the InRadius is with respect to the coordinate system of the object itself, it is scaled by the Transforms of the object and its ancestors. Therefore, from the perspective of an ancestor coordinate system, the InRadius actually specifies an ellipsoid, not a sphere.

The InRadius of an spDisplaying must be a float. When an spDisplaying is created, its InRadius is initialized to zero. Information about the InRadius is shared between processes.

17.3 OutRadius

```
public float OutRadius; //* [float]

      float spDisplayingGetOutRadius(sp Object)
      float spDisplayingGetOldOutRadius(sp Object)
      void spDisplayingSetOutRadius(sp Object, float X)

      float sqDisplayingGetOutRadius(sp Object)
      float sqDisplayingGetOldOutRadius(sp Object)
      void sqDisplayingSetOutRadius(sp Object, float X)
```

An spDisplaying object has a shared instance variable called the *OutRadius* that specifies the radius of the smallest sphere (positioned at the center of the object's coordinate system) in which the object can fit. This is included to provide a crude, but rapid, basis for applications to reason about collisions and other interactions between spDisplaying objects. Like the InRadius, the OutRadius is scaled by the Transforms of the object and its ancestors and therefore from the perspective of an ancestor coordinate system, actually specifies an ellipsoid, not a sphere.

The OutRadius of an spDisplaying must be a float. When an spDisplaying is created, its OutRadius is initialized to zero. Information about the OutRadius is shared between processes.

The InRadius, and OutRadius, of spDisplaying objects are included to provide a quick minimal step in the direction of describing the volume occupied by an object. Many other things are possible, like bounding boxes and BSP trees. These were not included because they would take up too much space to support as a general thing. Users can define specializations of spDisplaying that have more detailed volume descriptions if they want.

17.4 GraphicsNode (Internal)

```
transient public int GraphicsNode; //* [void *] internal

      void * spDisplayingiGetGraphicsNode(sp Object)
      void spDisplayingiSetGraphicsNode(sp Object, void * X)

      void * sqDisplayingGetGraphicsNode(sp Object)
      void sqDisplayingSetGraphicsNode(sp Object, void * X)
```

For the convenience of the visual renderer, spDisplaying objects have a local instance variable called the *GraphicsNode* that is used to point to the corresponding node (if any) in the scene graph being rendered. This facilitates the incremental modification of the scene graph when the information associated with an spDisplaying changes.

The GraphicsNode of an spDisplaying is an unrestricted pointer that is not manipulated by the system. When an spDisplaying is created, its graphics node is initialized to Null. Information about the GraphicsNode is maintained separately in each process.

18 spLinking (Fundamental)

```
public abstract class spLinking extends sp
```

Shared objects are divided into two major categories: small objects that can be communicated very rapidly between processes (on the order of a hundred milliseconds) and large slowly changing objects that are communicated more slowly between processes (on the order of a few seconds). The various subclasses of the class spLinking comprise the large slowly changing objects.

The key feature of spLinking objects is that they refer to an arbitrarily large piece of data via a Uniform Resource Locator (URL). The link object itself, including the URL, is communicated to other processes in the same way, and just as fast, as any other object. However, once the link itself has been received, additional work has to be done to retrieve the data specified by the URL. This can take several seconds.

As discussed in the documentation of ISTP, the data pointed to by the URL is retrieved over the World Wide Web using standard web protocols. Checksums and caching of local files are used to minimize unnecessary fetching of data while insuring that the data being used is always up to date.

Like any object, the Parent of a link controls how the link is communicated. In particular, the Parent of a link must ensure that the link is known in the locale where it is used. Typically, the Parent of a link is set to the object that uses it. To support prefetching of data, one can place links to the data in locales adjacent to the locale where it is used. Note that having multiple copies of a link does not cause the associated data to be loaded into memory multiple times.

An application may create links that it is not using at the moment but wishes to be able to use rapidly. For example, suppose that there is an avatar that wishes to be able to switch rapidly between several visual appearances. This can be supported by creating a link for each appearance and making the avatar be the Parent of these links. This will cause the links to be communicated to the locale the avatar is in and therefore preloaded in this locale. The avatar can then change its appearance instantaneously by changing which link its VisualDefinition variable points to.

The shared class spLinking inherits all the instance variables and functions of the class sp (Section 15). The class spLinking defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spLinkingC - Class descriptor (Section 15.1).
shared **URL** - URL string (Section 18.1).
shared Checksum - Compact data summary (Section 18.2).
FileName - Name of local cached copy of URL data (Section 18.3).
Data - In memory data (Section 18.4).

The class spLinking defines the following functions:

spLinkingNew - Creates link with given URL (Section 18.5).
spLinkingURLAltered - Signals alteration of URL data (Section 18.6).
spLinkingReadData - Reads URL data into memory (Section 18.7).

It is not possible to create instances of the class spLinking. Rather, one can only create instances of particular subclasses of the class spLinking.

The processing of links is a two level operation: getting the URL data into a local file cache and getting this data into memory.

A cache of local files corresponding to URLs is maintained in the local file system. Critically, this cache contains not only the files, but also version identifying checksums.

In-memory representations of URL data are created when necessary. When there is an in-memory copy, then every link with a given URL and checksum all point to just one copy of the associated in-memory data. This allows many links to point to the same data with little more cost than having just one link pointing to the data.

When a process hears about a link L, it proceeds it as follows.

(1) If the LoadData bit in the spClass object for L is False, then no data is ever read and no action is taken. (The system has to be careful to take correct action if the LoadData bit changes when several instances of the associated class already exist.)

(2) Otherwise, if L is already in the world model and the checksum has not changed, no action need be taken. The old Data value must be correct.

(3) Otherwise, if there is another link K of the same type as L with the same values for the URL and Checksum, then the data pointer is copied from K to L. This must be the correct value to use for L as well.

(4) Otherwise, the local file cache is checked to see whether the data corresponding to the specified URL and Checksum has already been fetched. If not, this data is fetched. Once the data has been fetched, the appropriate ReadData operation is used to create an in-memory representation of this data. The processing above is done in a separate thread so that it will not cause delays in the local process. During the (possibly long) time it takes for the new Data to be computed, the fact that L is about to change (or to appear) is hidden from the application. Once the Data becomes available, L is created if necessary and its checksum is set to the proper value. This delay is introduced so that the application will be able to use the new Data the moment it is able to detect that the link has appeared or changed. An application that does something with Link data must be an the lookout (e.g., with alerters) for links that appear and/or change so that it can respond appropriately.

18.1 URL (Fundamental)

```
public String URL; //* [spFixedAscii] readonly

      spFixedAscii spLinkingGetURL(sp Object)

      void spLinkingiSetURL(sp Object, spFixedAscii X)

      spFixedAscii sqLinkingGetURL(sp Object)
      void sqLinkingSetURL(sp Object, spFixedAscii X)
```

The central piece of information in an spLinking object is a shared instance variable called the *URL*, which contains a Uniform Resource Locator (URL) identifying the large piece of data being linked to. For example, spVisualDefinition links (Section 24) point to 3D graphical models. When a process is informed of the existence of an spVisualDefinition link, it fetches the data referred to by the link's URL for later use.

The URL of an spLinking must be a string. It must be less than 500 characters in length, so that the containing object can fit into a single UDP message. It must be specified at the time the link is initially created and cannot be modified after that time. Information about the URL is shared between processes.

It is inadvisable to have an object you own refer to a link created by another process because the link may get removed at any time. Rather, you should create your own link with the same URL and refer to that. (The system takes care of making sure that there are not two copies of the same URL data in memory at once, even if there are multiple links containing the URL.)

You really should not refer to someone else's URL either, unless you can trust them not to change or delete either the URL or the data it points to. To be completely safe, you should make a copy of the URL data and then refer to your own copy. (Note that if you are pointing to a URL that someone else changes, then you will not know that you should call spLinkingURLAltered (Section 18.6) and therefore your link will end up referring to an obsolete version of the data that can no longer be obtained.)

18.2 Checksum (Fundamental and Internal)

```
public int Checksum; //* [long] internal

      long spLinkingiGetChecksum(sp Object)
      long spLinkingiGetOldChecksum(sp Object)
      void spLinkingiSetChecksum(sp Object, long X)

      long sqLinkingGetChecksum(sp Object)
      long sqLinkingGetOldChecksum(sp Object)
      void sqLinkingSetChecksum(sp Object, long X)
```

Each spLinking object contains a shared instance variable called the *Checksum*, which contains a checksum of the data referred to by the URL. Communicating the Checksum in a link in addition to the URL makes it possible for a process to determine whether a cached copy of the URL data is up to date without refetching the URL data. If the system detects that the Checksum in a link has changed, then it immediately refetches the data. By means of the Checksums in links, communicating processes ensure that URL data is fetched when, but only when, necessary.

The Checksum is a 32-bit integer. The Checksum is automatically computed when an spLinking object is first created using a CRC algorithm. If the URL data is later modified, the Checksum needs to be changed. Changing the Checksum is triggered by using the function spLinkingURLAltered (Section 18.6). The Checksum should never be altered directly. Information about the Checksum is shared between processes.

Since the Checksum is computed directly from the actual data referred to by the link, it is guaranteed (with extremely high probability) to change whenever the data changes. This makes it more reliable and more useful than the various kinds of freshness information available form HTTP servers.

18.3 FileName

```
transient public String FileName; //* [char *]

      char * spLinkingGetFileName(sp Object)
      void spLinkingSetFileName(sp Object, char * X)

      char * sqLinkingGetFileName(sp Object)
      void sqLinkingSetFileName(sp Object, char * X)
```

The *FileName* local variable of an spLinking object contains the name of the local cached copy of the URL data. If the data has not been fetched (I.e., because the LoadData bit of the corresponding spClass object is False), then the FileName is a zero length string. If several links have the same URL, then they all point to the same cached copy of the data. The ReadData function typically operates by processing the data indicated by the FileName variable.

The FileName is automatically filled in by the system when the URL data is identified in the local cache (perhaps after having to be fetched from a remote source) and is automatically updated if the underlying data changes. The FileName and the data it points to should never be altered in any way. Information about the FileName is maintained separately in each process.

18.4 Data

```
transient public int Data; //* [void *] readonly

      void * spLinkingGetData(sp Object)

      void spLinkingiSetData(sp Object, void * X)

      void * sqLinkingGetData(sp Object)
      void sqLinkingSetData(sp Object, void * X)
```

The data fetched from the URL of an spLinking is stored in a local instance variable called the *Data*. If several links have the same URL, then the data is only read into memory once and each link points to the same block of data.

The kind of data pointed to by the Data pointer of a link differs from one kind of link to another. The Data pointer is automatically filled in by the system when a new link appears in the world model and is automatically updated when the underlying data changes. The Data pointer and the data it points to should never be altered in any way. Information about the Data is maintained separately in each process.

A key feature of the Data variable is that it is guaranteed that the Data value will be filled in before the Link becomes visible in the world model. Specifically, whenever a new link is created, the Data is read in by the New function. In addition, whenever a message is received describing a new link, the act of actually entering the link into the local world model is delayed until after the Data has been obtained. This delay makes things more convenient for processes that want to do something with the data and is irrelevant to ones that don't.

18.5 spLinkingNew

```
sp spLinkingNew(spFixedAscii URL)
```

URL - The URL of the link to be created.
Return value - The newly constructed object.

This function does not exist for the class spLinking itself. It is described here because it is an abstract prototype of the creation function that must be written when a new subclass of spLinking is defined. In particular, link creation functions must take a URL string as their first argument and create a link object containing the URL. This is the only way the URL in a link can be set.

The code below shows the general form that the New function for a subclass spL of spLinking must have. In particular, it must call spClassNewLink to create the link itself. This insures that Checksum is properly computed and the ReadData function is called if necessary. (Note that even if the LoadData bit for the link class is False, the URL data has to be fetched in order to calculate the proper Checksum, but an in-memory data image is created only if LoadData is True.)

```
sp spLNew(spFixedAscii URL) {
  return spClassNewLink(spLC(), URL);
}
```

18.6 spLinkingURLAltered

```
void spLinkingURLAltered(sp Link)
```

Link - The link whose URL has been altered.
Return value - There is no return value.

The purpose of this function is to notify the system that the data referred to by the URL in a link has changed. In the interest of efficiency, the system assumes that URL data never changes except when explicitly notified by an application.

When a link is created, the system retrieves the URL data, obtaining it from a local cache if possible. After that time, the system assumes that the data will not change unless explicitly instructed otherwise. If you change a file underlying the URL in a link you own, then you must notify the system by calling spLinkingURLAltered. This causes the system to update its URL caches, recalculate checksums, reload the data into memory (if needed), and notify all users of the link that the associated data that it has changed. This causes them to refetch it as well.

The processing above is mediated by the Checksums stored in links. These Checksums allow the user of a link to determine whether a cached data value is correct without refetching the URL data. The fact that spLinkingURLAltered causes the Checksum in the link to be updated, causes the link to be recommunicated to other processes, which causes them to refetch and reload the changed data if necessary.

18.7 spLinkingReadData

```
void spLinkingReadData(sp Link)
```

Link - The link object whose URL data is to be read into memory.
Return value - There is no return value.

This function does not exist for the class spLinking itself. It is described here because it is an abstract prototype of the URL data reading function that must be written when a new subclass of spLinking is defined.

The purpose of the data reading functions for link classes is to create an in-memory representation of the URL data based on a local disk copy of this data. The system automatically creates the local disk copy and stores the name of the local file in the FileName field. The ReadData function creates a memory image based on this data and stores a pointer to the result in the Data variable of the link object. It may initialize other local fields of the object as well.

The ReadData function can do whatever it wants with the local file to create an in-memory image of the specified data. The only requirement is that if called twice on the same local file, the ReadData function must create identical in-memory images of the data. This assumption is necessary so that the system can omit calling the ReadData function on duplicate copies of the same data.

The ReadData method for a class should be looked at as a default way to read in the associated data. An application process can override this by providing a different operation via the ReadDataFn of the associated spClass object. Further, whether the data should be read at all is under the control of the LoadData variable in the associated spClass object.

Every link class has the identical Data field so that the system can use a uniform approach to handling all links. In particular, the system decides whether and when to call the data-reading functions for links in order to avoid having to read the same data into memory twice. As part of this, multiple links can end up pointing to the same copy of the data.

The system handles the freeing of the data read in a similarly uniform way. When the last link referring to a particular piece of data is freed, the data is freed as well. To support this, data reading functions must allocate the space for what they read in one contiguous chunk.

19 spMultilinking (Fundamental)

```
public abstract class spMultilinking extends spLinking
```

An spMultilinking is a link that servers as an index into several pieces of data. Such an index is useful in situations where several equivalent pieces of data exist—e.g., 3D models at different levels of detail and in different formats.

When an spMultilinking is used, each process that encounters it picks a single piece of data to use from the index. After that, the processing of an spMultilinking is essentially identical to the processing that would be applied to an ordinary link pointing to the selected data.

The shared class spMultilinking inherits all the instance variables and functions of the classes: spLinking (Section 18) and sp (Section 15). The class spMultilinking defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spMultilinkingC - Class descriptor (Section 15.1).
Multipart - Specifies which data item is being used (Section 19.1).
IndexName - Name of local cached copy of index data. (Section 19.2).

The class spMultilinking defines the following functions:

spMultilinkingNew - Creates multilink with given URL (Section 19.3).
spMultilinkingSelect - Select data to use (Section 19.4).

It is not possible to create instances of the class spMultilinking. Rather, one can only create instances of particular subclasses of the class spMultilinking.

The data pointed to by the URL in an spMultilinking object is required to have the following standard format:

```
URL checksum
 <additional indented lines can be interpreted in any way by the ReadData method.>
URL checksum
 <additional indented lines can be interpreted in any way by the ReadData method.>
...
```

A multilink is an index of up to 256 alternative pieces of data. (The limit of 256 is imposed to facilitate the compact communication of changes in multilink data by the system core.)

Each entry in a multilink begins with a line containing a URL and checksum, which are logically equivalent to the URL and checksum in a simple link.

When multilinks are processed, exactly one of the entries in the index is used to retrieve the associated data. The entry to use is chosen by the spMultilinkingSelect function. A non-shared field called Multipart records which entry was chosen as the basis of the in-memory representation of the link data on a given occasion.

The Select function is called to choose an item if, and only if, the LoadData bit is True in the spClass object for the multilinking class. Local URL caching is applied only to the single item chosen. The other items are not fetched. (The index itself is of course fetched before the Select function is called.) If LoadData is False, nothing is fetched and no action is taken.

Multilinks have ReadData functions just like simple links. Once a particular data item has been selected, an in-memory image is created using the ReadData function.

The internal structure of the top-level data for a multilink has to be specified so that the data caching mechanism for links can be applied to the data referred to by the individual entries as well as to the index structure itself.

In particular, note that spLinkingURLAltered and spClassNewLink understand multilinks as well as links and operate on the items within a multilink as well as the index as a whole.

19.1 Multipart (Fundamental and Internal)

```
transient public int Multipart; //* [long] internal

      long spMultilinkingiGetMultipart(sp Object)
      void spMultilinkingiSetMultipart(sp Object, long X)

      long sqMultilinkingGetMultipart(sp Object)
      void sqMultilinkingSetMultipart(sp Object, long X)
```

The local variable *Multipart* specifies which data item in an spMultilinking index is in use, if any. The Multipart is a 32-bit integer. The Multipart must be set before the data associated with an spMultilinking can be loaded. Once the Multipart is set, it cannot be altered. Information about the Multipart is maintained separately in each process.

19.2 IndexName

```
transient public String IndexName; //* [char *]

      char * spMultilinkingGetIndexName(sp Object)
      void spMultilinkingSetIndexName(sp Object, char * X)

      char * sqMultilinkingGetIndexName(sp Object)
      void sqMultilinkingSetIndexName(sp Object, char * X)
```

The *IndexName* local variable of an spMultilinking object contains the name of the local cached copy of the index pointed to by the URL. If the index has not been fetched (I.e., because the LoadData bit of the corresponding spClass object is False), then the IndexName is a zero length string. If several multilinks have the same URL, then they all point to the same cached copy of the data. The Select function typically operates by processing the data indicated by the IndexName variable.

The IndexName is automatically filled in by the system when the URL data is identified in the local cache (perhaps after having to be fetched from a remote source) and is automatically updated if the underlying data changes. The IndexName and the data it points to should never be altered in any way. Information about the IndexName is maintained separately in each process.

19.3 spMultilinkingNew

```
sp spMultilinkingNew(v URL)
```

URL - The URL of the link to be created.
Return value - The newly constructed object.

This function does not exist for the class spMultilinking itself. It is described here because it is an abstract prototype of the creation function that must be written when a new subclass of spMultilinking is defined. In particular, multilink creation functions must take a URL string as their first argument and create a multilink object containing the URL. This is the only way the URL in a multilink can be set.

The code below shows the general form that the New function for a subclass spM of spMultilinking must have. In particular, it must call spClassNewLink to create the link itself. spClassNewLink understand multilinks and well as links and insures that Checksum is properly computed for the object as a whole and for each entry in the index. It also insures that the Select and ReadData functions are called if necessary. (Note that even if the LoadData bit for the multilink class is False, the URL data has to be fetched in order to calculate the proper Checksum and the URL data for each item must be fetched to calculate/check the appropriate checksum for each item. However, an in-memory data image is created only if LoadData is True.)

```
sp spMNew(spFixedAscii URL) {
  return spClassNewLink(spLC, URL);
}
```

19.4 spMultilinkingSelect

```
long spMultilinkingSelect(sp Link)
```

Link - The multilink object for which a choice must be made.
Return value - Zero-based index of item chosen (-1 means none).

This function does not exist for the class spMultilinking itself. It is described here because it is an abstract prototype of the URL data reading function that must be written when a new subclass of spMultilinking is defined.

The purpose of the selection functions for multilink classes is to decide which of the items in the index should be read into memory. The ReadData function does the actual reading. (The system fills in the FileName variable based on what the Select function chooses.) The Select function must set the Multipart field in order to specify which item has been chosen. The value -1 is used to indicate that no item has been chosen. If no item is chosen, then there will be no fetching of data over the net and no loading into memory even if LoadData True.

When a Select function is called, it typically operates by inspecting the index pointed to by the IndexName variable. (That is to say, the Select function is not called until after the URL data has been fetched over the net.) The Select function can do whatever it wants with the local file in order to make a decision, but should do so based solely on this file without fetching any of the URLs in it. The only requirement is that if called twice on the same local file, the Select function must make the same choice. This assumption is necessary so that the system can omit calling the Select function on duplicate copies of the same data.

The Select method for a class should be looked at as a default way to make a choice. An application process can override this by providing a different operation via the SelectFn of the associated spClass object. Further, whether a selection should be made at all is under the control of the LoadData variable in the associated spClass object.

20 spBeaconing (Fundamental)

```
public abstract class spBeaconing extends sp
```

Beacons is to provide a way for application processes to communicate with each other independent of the location-addressable communication operating in conjunction with locales. In particular, they provide a way to locate an object even when there is no knowledge about what locale the object is in.

The shared class spBeaconing inherits all the instance variables and functions of the class sp (Section 15). The class spBeaconing defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spBeaconingC - Class descriptor (Section 15.1).
shared **Tag** - Tag string (Section 20.2).
shared **Suppress** - Suppress processes triggered by the beacon (Section 20.3).

It is not possible to create instances of the class spBeaconing. Rather, one can only create instances of particular subclasses of the class spBeaconing.

A discussion of the concepts underlying Beacons can be found in [Barrus J.W., Waters R.C., and Anderson D.B., "Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments," *IEEE Virtual Reality Annual International Symposium*, (Santa Clara CA, March 1996), pages 204–213]. Detailed information about beacon communication is presented in the separate description of ISTP.

Whenever a process creates, removes, modifies or obtains ownership of a beacon, it communicates the beacon in the normal locale-based way. In addition, it sends information about the beacon to the Content-Based communication server specified by the DNS name in the beacon's Tag. A process can receive messages about spBeacons from: peer processes via locale-based communication, Content-Based communication servers as answers to queries, and Locale-Based Communication servers in response to connecting to new locales.

20.1 Using Beacons

Beacons provide a way to bootstrap from an initial state of knowing nothing about a part of a virtual world to a state of having located a locale of interest and become aware of the objects in that part of the world model. This is done in a three part process.

(1) Decide on a beacon tag that is of interest.

(2) Decide which if any of the beacons with that tag are of interest.

(3) Connect to the part(s) of the world model that contain the beacon(s) of interest.

The first step concerns solely the tags of beacons. In a given situation, there must be an a priori method for selecting tags of interest. Once a tag has been selected, spBeaconMonitor is used to get access to all the beacons with the tag no matter what locale they are in.

When a beacon B has been obtained by spBeaconMonitor it is likely that the local world model copy will not contain any information about the locale B is in except for B itself. Therefore the decision in the second step above of what beacons are interesting must be done solely based on data that is in the beacons themselves. To make this easier in a given application, a new subclass of beacons might be defined with some additional information.

To obtain information about the other objects in the locale B is in, a process must create an observer object (Section 21) and put it in the same locale. The only reliable way to do this is to initially make the Observer be a descendant of B. The reason that this is the only reliable approach is that B may well be the only object in the local world model copy that is in the locale in question.

Note that B has a Parent P. It might well be the case that it would be more convenient to make the Observer be a direct child of P. This typically cannot be done directly, because P is typically not known. However, the effect can easily be obtained by using spBeaconGoto instead of spBeaconMonitor to find the beacon in the first place.

There are several standard ways in which beacons are used. Perhaps the most common is to connect to a service provider such as a visual rendering process. In order to communicate with a server, a format for tags must be agreed on in advance. For instance to communicate between visual renderers and processes, the API uses tags of the form `"//ip-address/spVisual"` (e.g., `"//earth.merl.com/spVisual"`).

To contact a visual renderer, you create an spSeeing beacon with an appropriate tag. The renderer monitors beacons looking for tags with its fully qualified DNS address string in it. When it finds one, it creates an spVisualObserver as a child of the beacon and goes to work. If the beacon it is following gets removed, the renderer will then look for another beacon. (If two beacons both attempt to use the same renderer, the renderer will render based on the beacon it encounters first.)

A second prototypical example is using beacons to mark a fixed location. For example, this is done to indicate the landing pad that is the entry point for Diamond Park. In particular, the outside locale in Diamond Park contains a beacon whose Transform indicates the position and orientation of the center of the landing pad. This beacon is created by the process that creates the park. The beacon has a tag like `"/www.merl.com/Diamond-Park"` that is communicated (e.g., in Email) to people that might be interested in entering the park.

Someone who wants to visit the park creates an avatar, and then uses spBeaconGoto to put this avatar into the park. The avatar can then move freely about the park and into other locales such as the insides of the various buildings.

20.2 Tag (Fundamental)

```
public String Tag; //* [spFixedAscii] readonly

      spFixedAscii spBeaconingGetTag(sp Object)

      void spBeaconingiSetTag(sp Object, spFixedAscii X)

      spFixedAscii sqBeaconingGetTag(sp Object)
      void sqBeaconingSetTag(sp Object, spFixedAscii X)
```

The central instance variable of an spBeacon is a shared variable called the *Tag*. The key feature of beacons is that a global name service is maintained that maps from Tags to the beacon objects so that beacons can be accessed based on their Tags no matter where they are located. To allow many different people to create Tags that do not conflict with each other, beacon Tags are URLs. It is generally not a good idea to have lots of beacons with the same Tag, but there can be several beacons with the same Tag.

A beacon Tag is a string of type spFixedAscii. It must be less than 500 characters in length, so that the containing object can fit into a single UDP message. It must be specified when a beacon is initially created and cannot be changed after that time. Information about the Tag is shared between processes.

Note that the DNS/port part of a Tag URL (which must be present) selects the process that provides beacon service for the beacon.

20.3 Suppress

```
public boolean Suppress; //* [spBoolean]

      spBoolean spBeaconingGetSuppress(sp Object)
      spBoolean spBeaconingGetOldSuppress(sp Object)
      void spBeaconingSetSuppress(sp Object, spBoolean X)

      spBoolean sqBeaconingGetSuppress(sp Object)
      spBoolean sqBeaconingGetOldSuppress(sp Object)
      void sqBeaconingSetSuppress(sp Object, spBoolean X)
```

spBeacon objects have a shared instance variable called the *Suppress* bit that can be used to suppress the activity of processes that would otherwise by triggered by the beacon. Specifically, a process that sees a beacon targeted at it that has the Suppress bit True should not perform its normal activity, but should be ready at any moment to start doing so if the Suppress bit changes to False.

The Suppress bit of a beacon is set to False when the beacon is created. It can be changed at any time later. Information about the Suppress bit is shared between processes.

As an example of the use of the Suppress bit, consider the interaction of spSeeing beacons and visual rendering. Normally, an spSeeing beacon triggers the creation of an image. However, if the Suppress bit is on, then no image is created. Nevertheless, all the loading of model files and the like that is needed to create an image is done, so that the creation of an image can begin instantaneously when requested.

This can be used in two ways. First, you an toggle Suppress bits in order to jump rapidly from one view to another. Second, you can utilize an spSeeing beacon whose suppress bit is always False to trigger the preloading of visual information by moving it into locales in the direction you are going in advance of your actually going to these locales. This will not change how much is rendered at any one time, but will mean changes of scene can happen faster.

21 spObserving (Fundamental)

```
public abstract class spObserving extends sp
```

spObserving objects control Locale-Based communication between processes and therefore which objects are in the local world model copy. spObserving objects exercise this control by triggering the receipt of information about the locale they are in and typically about all neighboring locales as well. Different subclasses of spObserving give access to different kinds of information.

The shared class spObserving inherits all the instance variables and functions of the class sp (Section 15). The class spObserving defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spObservingC - Class descriptor (Section 15.1).
shared **Audio** - Indicates whether audio data is desired. (Section 21.1).
shared **IgnoreNearby** - Neighbor locale observation indication (Section 21.2).

It is not possible to create instances of the class spObserving. Rather, one can only create instances of particular subclasses of the class spObserving.

A critical part of Locale-Based Communication is that in addition to having a process only receive information about the parts of the world model it is attending to, it is also the case that when a process stops attending to part of the world model, then information about that part is purged from the local world model copy. To understand how this works, consider that if an object is in the local world model copy it is there for one of the following reasons:

(1) The object is owned by the local process.

(2) The object is a beacon whose Tag matches the Pattern of a locally owned spBeaconMonitor object, or the class of such a beacon or the locale of such a beacon.

(3) The object is a locale that contains a locally owned object or is the neighbor of a locale that contains a locally owned spObserving object with the IgnoreNearby bit off.

(4) The object is in a locale that is present due to (3).

(5) The object is a locale that is a neighbor of a locale present due to (2) or (3) or an spBoundary of a locale present due to (2) or (3) or (4).

Locale-Based Communication servers guarantee that when a connection is opened, the local process has every object corresponding to (3–5) and Content-Based Communication servers guarantee that you have every object corresponding to (2). The process itself inherently has every object corresponding to (1).

Every object in the world model whose presence is not required by one of the reasons above should be purged and put on the MsgRejectionQueue. This purging does not have to happen on every call to spWMUpdate, but should be checked whenever an spCom object or spBeaconMonitor is removed.

21.1 Audio

```
public boolean Audio; //* [spBoolean]

      spBoolean spObservingGetAudio(sp Object)
      spBoolean spObservingGetOldAudio(sp Object)
      void spObservingSetAudio(sp Object, spBoolean X)

      spBoolean sqObservingGetAudio(sp Object)
      spBoolean sqObservingGetOldAudio(sp Object)
      void sqObservingSetAudio(sp Object, spBoolean X)
```

When True, the *Audio* shared instance variable of an spObserving causes the system to receive streaming audio data. Otherwise, streaming audio data is not received.

The Audio bit of an spObserving can be freely turned on and off. By default, the Audio bit is False. (spAudioObserver objects change this default by setting the Audio bit True.) Information about the Audio bit is shared between processes.

21.2 IgnoreNearby (Fundamental)

```
public boolean IgnoreNearby; //* [spBoolean]

      spBoolean spObservingGetIgnoreNearby(sp Object)
      spBoolean spObservingGetOldIgnoreNearby(sp Object)
      void spObservingSetIgnoreNearby(sp Object, spBoolean X)

      spBoolean sqObservingGetIgnoreNearby(sp Object)
      spBoolean sqObservingGetOldIgnoreNearby(sp Object)
      void sqObservingSetIgnoreNearby(sp Object, spBoolean X)
```

When False, the *IgnoreNearby* shared instance variable of an spObserving causes the system to receive messages about objects in all the locales near (Section 26.1) the locale the spObserving object is in. If IgnoreNearby is True then the system attends only to the single locale the spObserving objects in.

The IgnoreNearby bit of an spObserving can be freely turned on and off. By default, the IgnoreNeighbors bit is False. Information about the IgnoreNearby bit is shared between processes.

22 spVisualParameters

```
public abstract class spVisualParameters extends sp
```

The class spVisualParameters groups together key pieces of data relevant to visual rendering. In an spSeeing object these parameters are used as requests. In an spVisualObserver object, these parameters are used to report on what the renderer is doing.

The shared class spVisualParameters inherits all the instance variables and functions of the class sp (Section 15). The class spVisualParameters defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spVisualParametersC - Class descriptor (Section 15.1).
shared **FarClip** - Far clipping distance (Section 22.1).
shared **NearClip** - Near clipping distance (Section 22.2).
shared **Field** - Field of view (Section 22.3).
shared **Interval** - Desired inter-frame interval (Section 22.4).

It is not possible to create instances of the class spVisualParameters. Rather, one can only create instances of particular subclasses of the class spVisualParameters.

22.1 FarClip

```
public float FarClip; //* [float]

      float spVisualParametersGetFarClip(sp Object)
      float spVisualParametersGetOldFarClip(sp Object)
      void spVisualParametersSetFarClip(sp Object, float X)

      float sqVisualParametersGetFarClip(sp Object)
      float sqVisualParametersGetOldFarClip(sp Object)
      void sqVisualParametersSetFarClip(sp Object, float X)
```

The *FarClip* shared instance variable of spVisualParameters specifies the distance to the far clipping plain that should (is) be(ing) used by the visual renderer. The FarClip is a float and is set to zero when an spVisualParameters is created. In an spSeeing, this indicates that the visual renderer should pick whatever value it feels is most appropriate. It can be changed later, but may take on the order of a second to take effect. Information about the FarClip is shared between processes.

22.2 NearClip

```
public float NearClip; //* [float]

      float spVisualParametersGetNearClip(sp Object)
      float spVisualParametersGetOldNearClip(sp Object)
      void spVisualParametersSetNearClip(sp Object, float X)

      float sqVisualParametersGetNearClip(sp Object)
      float sqVisualParametersGetOldNearClip(sp Object)
      void sqVisualParametersSetNearClip(sp Object, float X)
```

The *NearClip* shared instance variable of spVisualParameters specifies the distance to the Near clipping plain that should (is) be(ing) used by the visual renderer. The NearClip is a float and is set to zero when an spVisualParameters is created. In an spSeeing, this indicates that the visual renderer should pick whatever value it feels is most appropriate. It can be changed later, but may take on the order of a second to take effect. Information about the NearClip is shared between processes.

22.3 Field

```
public float Field; //* [float]

      float spVisualParametersGetField(sp Object)
      float spVisualParametersGetOldField(sp Object)
      void spVisualParametersSetField(sp Object, float X)

      float sqVisualParametersGetField(sp Object)
      float sqVisualParametersGetOldField(sp Object)
      void sqVisualParametersSetField(sp Object, float X)
```

The *Field* shared instance variable of spVisualParameters specifies the vertical viewing angle in radians that should (is) be(ing) used by the visual rendering process. The Field is a float and is set to zero when an spSeeing is created. In an spSeeing, this indicates that the visual renderer should pick whatever value it feels is most appropriate. It can be changed later, but may take on the order of a second to take effect. Information about the Field is shared between processes.

22.4 Interval

```
public int Interval; //* [spDuration]

      spDuration spVisualParametersGetInterval(sp Object)
      spDuration spVisualParametersGetOldInterval(sp Object)
      void spVisualParametersSetInterval(sp Object, spDuration X)

      spDuration sqVisualParametersGetInterval(sp Object)
      spDuration sqVisualParametersGetOldInterval(sp Object)
      void sqVisualParametersSetInterval(sp Object, spDuration X)
```

The *interval* shared instance variable of spVisualParameters specifies the time interval between frames in milliseconds. For instance, an interval of 33 specifies 30 frames per second. In an spSeeing, the Interval should be viewed as a target. It will not be possible for the visual renderer to achieve a very small Interval given a complex scene to render. The Interval is set to zero when an spPositioning is created. In an spSeeing, this indicates that the visual renderer should create frames as fast as possible. It can be changed later, but may take on the order of a second to take effect. Information about the Interval is shared between processes.

23 spAudioParameters

```
public abstract class spAudioParameters extends sp
```

The class spAudioParameters groups together key pieces of data relevant to visual rendering. In an spHearing object these parameters are used as requests. In an spAudioObserver object, these parameters are used to report on what the renderer is doing.

The shared class spAudioParameters inherits all the instance variables and functions of the class sp (Section 15). The class spAudioParameters defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spAudioParametersC - Class descriptor (Section 15.1).
shared **Focus** - Audio source to focus on (Section 23.1).
shared **Live** - Indicates whether live sources should be rendered (Section 23.2).
shared **Format** - Controls the format used for rendered sound (Section 23.3).

It is not possible to create instances of the class spAudioParameters. Rather, one can only create instances of particular subclasses of the class spAudioParameters.

23.1 Focus

```
public spAudioSource Focus; //* [sp]

      sp spAudioParametersGetFocus(sp Object)
      sp spAudioParametersGetOldFocus(sp Object)
      void spAudioParametersSetFocus(sp Object, sp X)

      sp sqAudioParametersGetFocus(sp Object)
      sp sqAudioParametersGetOldFocus(sp Object)
      void sqAudioParametersSetFocus(sp Object, sp X)
```

spAudioParameters objects have a shared instance variable called the *Focus* that specifies which surrounding sources should (are) be(ing) included as part of the rendered audio. If a particular spAudioSource is specified as the Focus, then only the sound from that source is received. Otherwise data from all nearby sources is mixed together. The Focus can be used both to zero in on a particular sound source so that it can be heard clearly amid many others and to limit the amount of processing that is required to support audio rendering.

The default value of the Focus is Null. In an spAudioSource, the Focus variable is ignored. In an spHearing, you can change it at will, but there may be a delay of a second or so before the change has any effect. (It is not expected that it will be changed often.) Information about the Focus is shared between processes.

23.2 Live

```
public boolean Live; //* [spBoolean]

      spBoolean spAudioParametersGetLive(sp Object)
      spBoolean spAudioParametersGetOldLive(sp Object)
      void spAudioParametersSetLive(sp Object, spBoolean X)

      spBoolean sqAudioParametersGetLive(sp Object)
      spBoolean sqAudioParametersGetOldLive(sp Object)
      void sqAudioParametersSetLive(sp Object, spBoolean X)
```

spAudioParameters objects have a shared instance variable called the *Live* bit, which specifies whether Live sources with the same owner should (are) be(ing) included in the rendered sound image. Specifically, if the Live bit is False for an spHearing beacon then spAudioSources with the Live bit True that have the same owner as the spHearing beacon triggering rendering are ignored. If the Live bit is True in an spHearing beacon, every source is treated the same. (This feature is used to suppress sound rendering of a user's spoken input in the headphones he is wearing.)

The default value of the Live bit is False. In an spHearing, you can change it at will, but there may be a delay of a second or so before the change has any effect. (It is not expected that it will be changed often.) Information about the Live bit is shared between processes.

23.3 Format

```
public long Format; //* [spFormat]

      spFormat spAudioParametersGetFormat(sp Object)
      spFormat spAudioParametersGetOldFormat(sp Object)
      void spAudioParametersSetFormat(sp Object, spFormat X)

      spFormat sqAudioParametersGetFormat(sp Object)
      spFormat sqAudioParametersGetOldFormat(sp Object)
      void sqAudioParametersSetFormat(sp Object, spFormat X)
```

spAudioParameters objects have a shared instance variable called the *Format*, which specifies the in-memory format for sound data. In an spHearing beacon, it can be set to zero with the meaning that the audio renderer should use whatever format it considers best. In an spAudioSource it specifies the format the application will use to write and observe data. In an spAudioObserver it specifies what format the renderer is using for localized sound.

This format is of type spFormat (Section 14). It is initialized to zero and can be changed later. If changed in an spHearing beacon, the change may take on the order of a second to take effect. Information about the Format is shared between processes.

24 spVisualDefinition

```
public class spVisualDefinition implements spMultilinking
```

An instance of the class spVisualDefinition is a multilink to one or more 3D graphical models specifying a visual appearance. Ordinary applications typically do not read in these 3D models. Rather, they are only read in by the visual renderer. When they are read in, they are converted into scene graphs in the internal format used by the renderer. The chunk of scene graph created is stored in the data variable of the spVisualDefinition link.

The shared class spVisualDefinition inherits all the instance variables and functions of the class spMultilinking (Section 19). The class spVisualDefinition defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spVisualDefinitionC - Class descriptor (Section 15.1).

The class spVisualDefinition defines the following functions:

spVisualDefinitionNew - Creates spVisualDefinition given a URL (Section 24.1).
spVisualDefinitionReadData - Reads visual model data (Section 24.2).
spVisualDefinitionSelect - Selects model to use (Section 24.3).

The basic approach for creating images of a virtual world centers around spDisplaying objects and their visual definitions. The appearances of all the spDisplaying objects in view at a given moment are combined together into a scene graph and rendered. Typically these objects will also inherit from spPositioning. In that case, the positions and orientations of the various appearances are taken from the Transform fields of the objects; otherwise, the identity spTransform is used.

Getting an appearance entered into the world model is a three step process.

(1) One first creates (or locates) a 3D model in VRML, or whatever other format the renderer you are using supports. This is done using standard modeling tools.

(2) Next one has to decide how the model should be positioned on an object in the world model. As discussed in detail below, this positioning is specified as part of the multilink index data.

(3) Finally, to get the appearance into a particular virtual world, you create an spVisualDefinition link whose URL refers to the intermediate description file and make it be the appearance of one or more spDisplaying objects. The following code shows an example of this.

```
look = spVisualDefinitionNew("http://www.models.com/demo/models/table");
object = spThingNew();
spDisplayingSetVisualDefinition(object, look);
spSetParent(look, object);
spPositioningSetTransform(object, Position);
```

The index files for an spVisualDefinition contain several pieces of information that are used as a basis for selecting which model to use. It also contains information that specifies how a given model should be used. This additional information is introduced by the keywords:

Format: - The Format of the corresponding model specified using standard MIME types. At the moment, VRML (I.e., "model/vrml") is the only supported format.

Polygons: - An integer specifying the number of polygons in the model. This is used to select level of detail based on the overall polygon budget of a given renderer. (This is optional and defaults to 0 indicating no information.

Importance: - An integer from 1 to 100 specifying the relative importance of this model. In general, a renderer will try to show more important models at higher detail. (This is optional and defaults to 0 indicating no information.)

Transform: - spTransform specifying how the model should be placed on an object. It can be used to move the origin, rotate the image, and/or change the scaling. (This is optional and defaults to the identity transform.)

The key element above is the positioning transform. It allows you to take an arbitrary model and use it without having to alter it in any way. For example, suppose that you are creating a virtual world that contains a virtual car that can move around. To conform with the API's object orientation standards (Section 8), the spThing for the car must be oriented with the Y axis up and with the front of the car facing down the negative Z axis. Further, to make it easy to move the car around, it is convenient to place the origin of the car on the ground under the center of the car's appearance. In addition, the car in the virtual world has some intended size. (These latter two needs would remain even if the API had no object orientation standards.) It is very unlikely that a given model for the car's appearance will meet these conditions. However, using a positioning transform it is trivial to adapt the model to its use in the virtual world without having to modify the model itself.

Continuing the example above, "http://www.models.com/demo/models/table" might contain:

```
http://www.models.com/demo/models/table.vrml 88347136
  Format: model/vrml
  Polygons: 36
  Importance: 40
  Transform: -1.0 8.0 0.2  0.0 0.0 1.0  1.523  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
http://www.models.com/demo/models/table-detailed.vrml 88347136
  Format: model/vrml
  Polygons: 420
  Importance: 20
  Transform: 0.0 4.0 0.2  0.0 1.0 0.0  2.345  2.0 2.0 2.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
```

The programming API considers graphical models to be completely opaque. All it cares about is that the visual renderer being used can understand the models created by the modeling tools being used. The system itself never looks at what is in a graphical model. In particular, it is not intended that an application dynamically modify models, but rather only effect the overall scene graph by affecting the aspects of spThings (such as their Transforms) that are visible in the world model. (However, through the definition of application specific spActions, one could get around this restriction.)

24.1 spVisualDefinitionNew

```
sp spVisualDefinitionNew(spFixedAscii URL)
```

URL - URL of the spVisualDefinition to be created.
Return value - The newly constructed object.

Creates a new object of the class spVisualDefinition, with the local process as its owner. In addition, sets the URL of the spVisualDefinition as specified.

24.2 spVisualDefinitionReadData (Internal)

```
void spVisualDefinitionReadData(sp Link)
```

Link - The spVisualDefinition link whose data is to be read in.
Return value - There is no return value.

spVisualDefinitionReadData does nothing, because there is no way to predict what the visual renderer in use needs to have done. The visual rendering process must specify a local ReadData operation via the spClassReadDataFn variable for spVisualDefinition.

24.3 spVisualDefinitionSelect

```
long spVisualDefinitionSelect(sp Link)
```

Link - The link for which a selection is to be made.
Return value - Returns zero.

spVisualDefinitionSelect merely selects the first model returning zero. If a visual rendering process wants to make any interesting kind of selection, it must specify a local Select operation via the spClassSelectFn variable for spVisualDefinition.

25 spSound

```
public class spSound implements spMultilinking
```

An important feature of the API is a facility for prerecorded sound. This is intended to be used for (usually short) things that get used over and over (like the sound of a door closing). Their advantage is that the data can be prestored at the destination processes instead of having to be transmitted every time it is used.

An instance of the class spSound is a link to a recorded block of sound data such as ambient background sound or a sound effect. The data can be created using a number of standard tools such as sound editors and written in files in a number of standard formats. They are then referred to using spSound links.

The shared class spSound inherits all the instance variables and functions of the class spMultilinking (Section 19). The class spSound defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spSoundC - Class descriptor (Section 15.1).
Duration - Time duration of data (Section 25.1).

The class spSound defines the following functions:

spSoundNew - Creates spSound given a URL (Section 25.2).
spSoundPlay - Plays stored sound (Section 25.3).
spSoundSelect - Selects sound to use (Section 25.4).
spSoundReadData - Reads stored sound data (Section 25.5).

The basic approach presented here for creating sound effects in a virtual world centers around spSound and spSoundSource objects. The spSound objects specify what the various sound effects sound like. The positions of the spSoundSources specify where sounds emanate. Getting a sound effect to happen in a virtual world is a three step process.

(1) One first creates (or locates) sound files in one of several common formats. This is done using standard sound capture and editing tools.

(2) As discussed in detail below, one then creates an index file that summarizes key features of one or more versions of the sound.

(3) Finally, to get the sound into a particular virtual world, you create an spSound link whose URL refers to the intermediate description file. You then create one or more spSoundSource objects and use spSoundPlay to play the sound at the right places and the right times. The following code shows an example of this.

```
bang = spSoundNew("http://www.models.com/sounds/bang23.wav");
source = spAudioSourceNew();
spSetParent(bang, source);
spSoundPlay(bang, source, FALSE, 1.0);
```

The index files for an spVisualDefinition contain several pieces of information that are used as a basis for selecting which model to use. It also contains information that specifies how a given model should be used. This additional information is introduced by the keywords:

Format: - The format of sound file specified using standard MIME types e.g., "audio/x-wav".
spFormat - An spFormat specifying the detailed format of the data in the file.
Duration - Length of sound data in milliseconds. This value is needed so that the Duration variable of an spSound can be set correctly even if the sound data itself is not read into memory.
Bytes - Length of sound data in bytes.
Importance: - An integer from 1 to 100 specifying the relative importance of this sound. In general, a renderer will try to play more important sounds at higher detail. (This is optional and defaults to 0 indicating no information.)

It is expected that the various sound files listed in an index file will have different encodings or different sample rates. For example, you might have 8 KHz file for the use of low powered machines and a 32 KHz high fidelity version for more powerful machines. The process loading a sound is free to choose whichever sound file is most appropriate for it.

Continuing the example above, `"http://www.models.com/sounds/bang23"` might contain:

```
http://www.models.com/sounds/bang23.wav 88347136
  Format: audio/x-wav
  spFormat: 6638759684996246
  Duration 123
  Bytes: 4500
  Importance: 75
http://www.models.com/sounds/bang23.aiff 72545754
  Format: audio/x-aiff
  spFormat: 5754345865388749
  Duration 123
  Bytes: 9000
  Importance: 25
```

The programming API considers stored sounds to be completely opaque. All it cares about is that the audio renderer being used can understand the sound files created by the modeling tools being used. The system itself never looks at what is in a sound file.

25.1 Duration

```
transient public int Duration; //* [spDuration] readonly

      spDuration spSoundGetDuration(sp Object)

      void spSoundiSetDuration(sp Object, spDuration X)

      spDuration sqSoundGetDuration(sp Object)
      void sqSoundSetDuration(sp Object, spDuration X)
```

spSound objects have a local instance variable call the *Duration* that records the period of time corresponding to the sound data referred to. This is recorded in a variable so that processes other than an audio renderer can reason about how much time is required to play back the data without having to load the entire sound file into memory. Among other things, this is needed for the proper operation of the actions created by spSoundPlay. In order for simulations to reason about sounds, they need to be able to observe spDoSoundPlay Actions and tell when they are done. For this to work, the spDoSoundPlay actions need to know how long they should remain in existance. They tell this by looking at the Duration of the spSound they are playing.

The Duration is a 32-bit integer of the type spDuration specifying the time in milliseconds. It should be set by the spSound ReadData function based on the URL data even when the full sound data is not read into memory. If the number of samples does not correspond to an exact integer number of milliseconds, then the time is calculated by rounding it to the nearest millisecond. The Duration should not be altered after that time. Information about the Duration is maintained separately in each process.

25.2 spSoundNew

```
sp spSoundNew(spFixedAscii URL)
```

URL - URL of the spSound to be created.
Return value - The newly constructed object.

Creates a new object of the class spSound, with the local process as its owner. In addition, sets the URL of the spSound as specified.

25.3 spSoundPlay

```
sp spSoundPlay(sp Sound, sp Source, spBoolean Loop, float Gain)
```

Sound - The spSound to be played.
Source - The spAudioSource to play the sound through.
Loop - If True, play the sound endlessly.
Gain - Gain multiplier, 1.0 means no change.
Return value - The action created.

Causes a stored sound to be played through an spAudioSource. The key feature of spSoundPlay is that it does not actually output the sound through the source, but rather creates an spDoSound-Play action that simulates the receipt of the sound by each process that can hear it, without having to send the sound data over the network.

The playing of sound continues until: the data runs out, the action object returned by spSoundPlay is removed, the source object is removed, the spSound object is removed, or some other sound is played through the same spAudioSource. (You can only be playing one sound at a time through a sound source.)

25.4 spSoundSelect

```
long spSoundSelect(sp Link)
```

Link - The link for which a selection is to be made.
Return value - Returns zero.

spSoundSelect merely selects no model returning -1, but setting the Duration to the duration of the first sound. This is useful in situations where a process merely needs to know how long sounds are, but is of no use to an audio renderer. An audio rendering process must specify a local Select operation via the spClassSelectFn variable for spSound.

25.5 spSoundReadData (Internal)

```
void spSoundReadData(sp Link)
```

Link - The spSound link whose data is to be read in.
Return value - There is no return value.

spSoundReadData does nothing, because there is no way to predict what the audio renderer in use needs to have done. The audio rendering process must specify a local ReadData operation via the spClassReadDataFn variable for spSound.

26 spLocale (Fundamental)

```
public class spLocale implements spLinking, spBeaconing, spDisplaying
```

Locales break the world model up into pieces that are handled separately. They are the basis for the scalability of the world model. They make it possible for both the size of the local world model copy and the number of incoming messages that have to be handled per second to be dependent only on the part of the virtual world that is 'near' to the local process, rather than dependent on the total size of the virtual world as a whole.

In addition, each locale has a separate coordinate system and arbitrarily defined neighbor relationships. This makes it easy to combine separately designed pieces into arbitrarily large virtual worlds and to support a variety of interesting effects.

The shared class spLocale inherits all the instance variables and functions of the classes: spLinking (Section 18), spBeaconing (Section 20) and spDisplaying (Section 17). The class spLocale defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spLocaleC - Class descriptor (Section 15.1).
shared **Boundary** - 3D extent of locale (Section 26.2).
NumNeighbors - Number of neighbors (Section 26.3).

The class spLocale defines the following functions:

spLocaleNew - Creates spLocale given URL (Section 26.4).
spLocaleChoose - Determines appropriate locale for position (Section 26.5).
spLocaleExportMatrix - Retrieves export matrix (Section 26.6).
spLocaleReadData - Reads locale data (Section 26.7).

An in-depth discussion of the concepts underlying Locales can be found in [Barrus J.W., Waters R.C., and Anderson D.B., "Locales: Supporting Large Multiuser Virtual Environments", *IEEE Computer Graphics and Applications*, 16(6):50–57, November 1996.] spLocales embody the residue of features described in the above paper that are not supported by spLocales.

Note locales are links (to data describing the relationship between locales), beacons (so that they can easily be found), and spDisplaying objects (so that they can have an associated background).

26.1 How Locales Work

The key feature of locales is that every shared object is in exactly one locale. The locale is determined by following the object's Parent link. In particular, every object other than an spLocale needs to have a Parent. If the Parent of an object is a locale, then the object is *in* that locale. Otherwise, the object is in whatever locale its Parent is in. Note that it is not possible to have an application without at least one locale, because without locales there is no communication.

A locale is a link; however, two locales that are as identical as possible describe locales that are nevertheless distinct. A locale has an owner that is the only process that can change it. If the owner of a locale leaves the session, the locale will go away, turning all the objects in it into orphans. All the information about the relationships between locales is stored in predefined data files and read in at run time.

The URL in an spLocale link identifies the data describing the locale. The Tag serves as a mnemonic name for the locale and identifies the process that must act as both a Content-Based Communication server for the locale object itself and a Locale-Based Communication server for objects in the locale. Both the Tag and the URL must be specified at the moment the locale is defined and not changed later.

Each locale is associated with a server process that keeps track of what is going on in the locale. The address of this process is specified as part of the locale object.

Each locale is also associated with a particular set of communication addresses. Information about objects in a locale is communicated only via those addresses. An orphaned object that is not in any locale because it has no Parent is not communicated to anybody. The exact addresses to use are chosen at run time from a range of suggested addresses that are specified when the locale is defined.

An important feature of a locale is its boundary. The boundary specifies the three dimensional extent of the locale. The boundary of an spLocale is typically specified when the locale is first created and not changed later.

It must be stressed that what locale an object is in for communication purposes is determined by its Parent as discussed above, not any geometric considerations. However, in general, an object should be placed in a locale only if its position is within the boundary of the locale. (A point is within a boundary only if the Inside function of the boundary returns True when applied to the boundary and the position of the object.)

In general, transferring an object form one locale to another involves nothing more complicated than changing its Parent. However, it can be somewhat complicated to figure out which locale an object should be in. Determining this is facilitated by the function spPositioningLocalize. If an object whose Parent is an spLocale is not within the boundary of the locale, and there is a nearby locale whose boundary does contain the object, then spPositioningLocalize transfers the object to the appropriate nearby locale.

A key concept above is what locales are *neighbors* of a given locale. This is statically specified when locales are designed. A set of locales that neighbor each other pairwise is called a scene. (One can move incrementally step by step from one locale to another in a scene using spThingLocalize; however, the only way to get from one scene to another is to teleport to the destination by explicitly setting the Parent of the object to be moved.)

In an spLocale, the data variable is a vector of structures describing the relationship between the locale and its neighbors. Each data structure for a neighbor N of locale L contains the following:

URL - URL of locale N. Critically, this is typically a relative hyperlink. Groups of locales can be defined in a single file. A full URL is only needed when referring to a neighbor in a different group of locales.

Locale - Locale object for N, if known.

Import matrix - The import matrix I is an spMatrix specifying how to convert a matrix T relative to the coordinate system of N into a matrix S relative to the coordinate system of L (i.e., S=IxT). This is used when one wants to know the position of something in N relative to L.

Export matrix - The export matrix E is an spMatrix specifying how to convert a matrix S relative to the coordinate system of L into a matrix T relative to the coordinate system of N (i.e., T=ExS). The export matrix is used when transferring an object from one locale to another. Typically E is the inverse of I; however, interesting special effects can be obtained when this is not the case.

The file pointed to by the URL in a locale object must contain one or more locale descriptions of the following form. Each description begins with a line of the form "NAME=*name*", where *name* is the identifying name of the locale. This is followed by a line beginning "TAG=" that specifies the Tag of the locale. The DNS name and port in the Tag specify the server for the locale. This is then optionally followed by a line beginning "MULTICASTRANGE=" which contains two Internet addresses separated by a space. If omitted this range takes on a system specified default. (ISTP servers assign multicast addresses dynamically, from specifed ranges if any. Therefore, it is fine not to specify any ranges or to specify the same range for many different locales.)

The above preamble lines are followed by three lines for each neighbor containing:

A) NEIGHBOR= followed by the (relative) hyperlink reference to the file entry containing data for the neighboring locale.

B) IMPORT= followed by the import spTransform (17 floats, X Y Z RX RY RZ RA SX SY SZ SOX SOY SOZ SOA CX CY CZ separated by blanks). This spTransform is converted to an spMatrix when the locale data file is read in.

C) EXPORT= followed by the export spTransform (17 floats separated by blanks) which is converted to an spMatrix when read in.

for example, you might have in the file "http://www.BigRetailer.com/cyberstore"

```
NAME=MainRoom
TAG=//www.cybermall.com/BigBookstore
MULTICASTRANGE=225.0.0.0 225.0.0.255
NEIGHBOR=http://www.cybermall.com/locales#MainHall
IMPORT=  1.0 0.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
EXPORT= -1.0 0.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
NEIGHBOR=#BackRoom
IMPORT=  0.0  1.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
EXPORT=  0.0 -1.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
NAME=BackRoom
TAG=//www.cybermall.com/BigBookstore-BackRoom
NEIGHBOR=#MainRoom
IMPORT=  0.0 -1.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
EXPORT=  0.0  1.0 0.0  0.0 0.0 1.0  0.0  1.0 1.0 1.0  0.0 0.0 1.0 0.0  0.0 0.0 0.0
```

Note the use of standard HTTP syntax when referring to individual locale entries. For example, the data pointed to by "http://www.cybermall.com/locales" assumedly contains a reference to "http://www.BigRetailer.com/cyberstore#MainRoom". Note also that while a file of locales will typically contain many links to locales in the same or nearby files, it must contain at least one absolute link to a locale in a far away file in order to be connected to the rest of a large virtual world.

Each locale defines a separate coordinate system. This coordinate system is the one used to interpret the boundary of the locale and the transforms of spThings in the locale. The matrices in the neighbor data structures relating locales are constants and cannot be rapidly changed. (Since they are link data, it is possible to change them, but not quickly.) The Parent of a locale is Null.

26.2 Boundary

```
public spBoundary Boundary; //* [sp]

      sp spLocaleGetBoundary(sp Object)
      sp spLocaleGetOldBoundary(sp Object)
      void spLocaleSetBoundary(sp Object, sp X)

      sp sqLocaleGetBoundary(sp Object)
      sp sqLocaleGetOldBoundary(sp Object)
      void sqLocaleSetBoundary(sp Object, sp X)
```

The *Boundary* shared instance variable of an spLocale object specifies the object describing the boundary of the locale. The Boundary must be an spBoundary (Section 27) object. It is used by spPositioningLocalize (Section 16.8) to determine when an object should be placed in a locale. Information about the Boundary is shared between processes.

26.3 NumNeighbors (Internal)

```
transient public int NumNeighbors; //* [long] internal

      long spLocaleiGetNumNeighbors(sp Object)
      void spLocaleiSetNumNeighbors(sp Object, long X)

      long sqLocaleGetNumNeighbors(sp Object)
      void sqLocaleSetNumNeighbors(sp Object, long X)
```

The *NumNeighbors* local instance variable of an spLocale specifies the number of neighboring locales. The neighbors themselves are described in the Data instance variable of the locale.

The NumNeighbors variable is a 16-bit integer. It is set by spLocaleReadData when a locale is loaded and cannot ever be changed. Information about the NumNeighbors is maintained separately in each process.

26.4 spLocaleNew

```
sp spLocaleNew(spFixedAscii URL, spFixedAscii Tag)
```

URL - URL of the spLocale to be created.
Tag - Tag of the spLocale to be created.
Return value - The newly constructed object.

Creates a new object of the class spLocale, with the local process as its owner. In addition, sets the URL and Tag of the spLocale as specified. Neither the URL nor the Tag can be altered after they are initially set. locales are the only built-in objects with two spFixedAscii variables.

To create a locale, you might write the following. To insure proper communication of the Boundary, the Parent of the Boundary must be set to be the locale. The same is true for the background appearance.

```
sp L,B,background;
L = spLocaleNew("http://www.BigRetailer.com/locales#BackRoom");
B = spBoundaryNew("http://www.BigRetailer.com/boundaries/BackRoom");
background = spVisualDefinitionNew("http://www.models.com/demo/models/bookstore");
spLocaleSetBoundary(L,B);
spSetParent(B,L);
spDisplayingSetVisualDefinition(L, background);
spSetParent(background, L);
```

26.5 spLocaleChoose (Internal)

```
sp spLocaleChoose(sp L, spMatrix P)
```

L - Locale specifying coordinate system.
P - Position in relation to L.
Return value - Locale containing position, if any.

Takes a position specified in a locale and determines whether it more naturally corresponds to a position in a neighboring locale, returning the most appropriate containing locale, if any. (This is a fundamental computation underlying spPositioningLocalize.)

Specifically, given a locale L and an spMatrix P specifying a position and orientation with respect to the coordinate system specified by L, spLocaleChoose checks L and all the locales neighboring L to see which ones have boundaries that encompass P. If any of these locales contain P, then the containing locale whose boundary has the smallest Volume is returned. Otherwise Null is returned.

26.6 spLocaleExportMatrix (Internal)

```
spMatrix spLocaleExportMatrix(sp L, sp Destination)
```

L - Source locale.
Destination - Destination Locale.
Return value - Export matrix from source to Destination. spMatrix to store export matrix in.

Determines the export matrix relating two neighboring locales. (This is a fundamental computation underlying spPositioningLocalize.)

Specifically, given a locale L and another locale N that must be a neighbor of L and cannot equal L, spLocaleExportMatrix returns the export matrix E that should be used to convert positions relative to L into positions relative to N. (The matrix Q relative to N that specifies the same position as a matrix P relative to L is Q=ExP.) The matrix returned shares memory with the Locale data for L and must not be modified by an application.

Note that since E is multiplied on the left, you might well use it as the left operand of spMatrixMult. However, this argument gets modified. Therefore, you must copy E before using it this way in order not to destroy the Matrix stored with the locale L.

26.7 spLocaleReadData (Internal)

```
void spLocaleReadData(sp Link)
```

Link - The spLocale link whose data is to be read in.
Return value - There is no return value.

Fills in the data in an spLocale object based on the information in the file specified by the URL. The system depends on this ReadData function always being used to load in locale data.

27 spBoundary

```
public class spBoundary implements spLinking
```

Boundaries encapsulate the basic concept of a machine manipulable description of a 3D volume. Their primary use is in conjunction with spLocale objects. However, they can also be used in other situations as well.

The shared class spBoundary inherits all the instance variables and functions of the class spLinking (Section 18). The class spBoundary defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spBoundaryC - Class descriptor (Section 15.1).
Volume - Approximate volume of boundary (Section 27.1).

The class spBoundary defines the following functions:

spBoundaryNew - Creates spBoundary given URL (Section 27.2).
spBoundaryBelow - Calculates boundary intercept (Section 27.3).
spBoundaryAbove - Calculates boundary intercept (Section 27.4).
spBoundaryInside - Determines whether point is within boundary (Section 27.5).
spBoundaryReadData - Reads spBoundary data (Section 27.6).

The URL in an spBoundary link identifies a file containing a detailed description of the boundary. In the case of spBoundary objects themselves, the description is an axis aligned bounding box. Specifically, the file identified by the URL must contain one line of the form "VOLUME=*float*" specifying the volume of the boundary followed by another line of text consisting of 6 floats separated by spaces. The six floats specify the minimum X value, minimum Y, minimum Z, maximum X, maximum Y, and maximum Z values of the spBoundaries bounding box respectively. (These numbers are stored in the Data variable of an spBoundary object.) For instance, the file might contain:

```
VOLUME=30.0
-1.0 2.5 0.0 9.0 5.5 1.0
```

The boundary information provided by an spBoundary object is very simple, and sufficient for some applications. Subclasses of spBoundary are free to use more detailed representations such as BSP trees.

spBoundary objects are separate objects from spLocales because they support a completely separate concept. spLocales need spBoundaries to work, but spBoundaries can be useful without spLocales. In particular, an spBoundary smaller than (or larger than) an spLocale can be used by itself as part of controlling a motion, without wanting to have any effect on communication.

27.1 Volume

```
transient public float Volume; //* [float] readonly

      float spBoundaryGetVolume(sp Object)

      void spBoundaryiSetVolume(sp Object, float X)

      float sqBoundaryGetVolume(sp Object)
      void sqBoundarySetVolume(sp Object, float X)
```

The *Volume* local instance variable of an spBoundary object specifies the approximate volume (in cubic meters) enclosed by the boundary. The Volume is specified by the boundary descriptor file. It is set by the ReadData function when a boundary is loaded. Information about the Volume is maintained separately in each process.

27.2 spBoundaryNew

```
sp spBoundaryNew(spFixedAscii URL)
```

URL - URL of the spBoundary to be created.
Return value - The newly constructed object.

Creates a new object of the class spBoundary, with the local process as its owner. In addition, sets the URL of the spBoundary as specified.

27.3 spBoundaryBelow

```
long spBoundaryBelow(sp Boundary, spVector P, spVector Q)
```

Boundary - The spBoundary used as the reference.
P - The point that may be above the boundary.
Q - Modified to reflect position below P on boundary.
Return value - Non-zero if there is a boundary point below the input position.

Given an (X,Y,Z) position P, this function determines whether a point P is above a boundary, and if it is, the position on the boundary that is immediately below P. Specifically, spBoundaryBelow casts a ray downward from P and determines whether and where this ray intersects the boundary. Since 'down' is defined to be along the negative Y axis, if there is a point below P on the boundary, then this point has the same X and Z values as P, but an equal or a lessor Y value.

If there is not a point on the boundary below P, then zero is returned. Otherwise, non-zero is returned. An integer is used as the return value so that subclasses of spBoundary can return more complex results as long as they adhere to the requirement that 0 means false.

In addition, if the Q argument is not Null, then the X, Y, and Z components of Q are altered so that they are equal to the coordinates of the boundary point below P. (It is permissible for the two vector arguments of spBoundaryBelow to be identical.)

For the class spBoundary itself, the definition of Below is trivial. To determine whether the ray case downward from P intersects the boundary, It merely has to determine if the X and Z coordinates of P are within the X-Z bounds of the locale. If so the intersection point has the X and Z coordinates of P and has a Y value that is largest of the Min or Max Y value of the boundary that is less than the Y value of P. However, it is expected that the definitions of Below for subclasses of spBoundary will do more detailed and complex calculations.

Since the X and Z coordinates of Q must be the same as P, it would have been possible for spBoundaryBelow to return just a single float. However, it is convenient to allow Q to be the same as P and therefore have P modified. Also, there is no way to return both a float and a logical value in the Java API.

27.4 spBoundaryAbove

```
long spBoundaryAbove(sp Boundary, spVector P, spVector Q)
```

Boundary - The spBoundary object used as a reference.
P - The point that may be below the boundary.
Q - Modified to reflect position above P on boundary.
Return value - Non-zero if there is a boundary point above the input point.

The Above function for a subclass of spBoundary is identical to the Below function except that it determines whether there is a point on the boundary that is above the test point, rather than below.

27.5 spBoundaryInside

```
long spBoundaryInside(sp Boundary, spVector P)
```

Boundary - The spBoundary used as the reference.
P - The point that may be within the boundary.
Return value - Non-zero if the input position is inside the boundary.

Given an (X,Y,Z) position P, this function determines whether a point P is on or inside the 3D volume specified by an spBoundary. If P is on or within the boundary, then non-zero is returned. Otherwise, zero is returned.

For the class spBoundary itself, the definition of Inside is trivial. However, it is expected that the definitions of Inside for subclasses of spBoundary will do more detailed and complex calculations.

27.6 spBoundaryReadData (Internal)

```
void spBoundaryReadData(sp Link)
```

Link - The spBoundary link whose data is to be read in.
Return value - There is no return value.

Reads in the data in an spBoundary data file and stores it in the Data variable as a vector of six floats. The other spBoundary methods depend on this ReadData function always being used to load in spBoundary data.

28 spTerrain

```
public class spTerrain extends spBoundary
```

spTerrain is a specialization of spBoundary that allows much more complex boundaries to be specified very efficiently. An spLocale can make use of either an spBoundary or an spTerrain.

The shared class spTerrain inherits all the instance variables and functions of the classes: spBoundary (Section 27) and spLinking (Section 18). The class spTerrain defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spTerrainC - Class descriptor (Section 15.1).

The class spTerrain defines the following functions:

spTerrainNew - Creates spTerrain given its URL (Section 28.1).
spTerrainBelow - Calculates boundary intercept (Section 28.2).
spTerrainAbove - Calculates boundary intercept (Section 28.3).
spTerrainInside - Determines whether point is within terrain (Section 28.4).
spTerrainReadData - Reads spTerrain data (Section 28.5).

spTerrain objects are implemented as a collection of triangles that define a *floor* and/or *ceiling*. A terrain model is quite similar to a 3D graphic model and can be computed from a graphic models. However, a terrain model is organized in such a way that the methods Inside, Below and Above can be implemented extremely efficiently.

The triangles in a terrain can overlap so as to represent a multi-level floor. In that case, the points returned by spTerrainBelow and spTerrainAbove will depend on the altitude of the input point.

The key virtue of spTerrains is that they allow very complex boundary outlines (including volumetric holes) to be defined and yet allow the very rapid determination of the intersection of vertical lines with the boundary. spTerrains are used for two quite different things: supporting terrain following and checking whether or not an object is within a boundary.

Another area where spTerrain goes beyond the basic spBoundary class is that each triangle in an spTerrain can be associated with a tag value. The methods Below, Above, and Inside return these tag values rather than just zero and one.

If a tag value is positive then it indicates that a triangle is part of the terrain. If a tag value is zero, this indicates that the associated triangle should be ignored as if it did not exist. If a tag value is negative then it indicates that a triangle acts as a barrier that hides the terrain from certain points of view. As illustrated below, this is useful for creating certain kinds of holes in terrains.

spTerrainBelow locates the highest (along the Y axis) triangle with a non-zero tag that is below the query point. If there is such a triangle and it has a positive tag, then this tag is returned. Otherwise, zero is returned.

spTerrainAbove is identical to spTerrainBelow except that it locates the lowest (along the Y axis) triangle with a non-zero tag that is above the query point and returns this tag if it is positive.

spTerrainInside determines whether a query point is in the interior of a terrain. If spTerrainAbove and spTerrainBelow both return positive values given the query point, then spTerrainInside returns what spTerrainBelow returned. Otherwise, zero is returned.

As an example, consider the two-dimensional terrain cross section illustrated below. spTerrainBelow applied to the point X returns 3, while spTerrainBelow applied to Y returns 0. Therefore, X is inside the terrain while Y is not.

```
 4  4  4  4  4  4  4
          X
 3  3  3  3  3  3  3
-1 -1 -1 -1 -1 -1 -1
          Y
-1 -1 -1 -1 -1 -1 -1
 2  2  2  2  2  2  2

 1  1  1  1  1  1  1
```

The file specified by the URL in an spTerrain has the following format. The first line specifies the volume of the terrain, which may be only approximate. The first part of the file is a list of points that are vertices of triangles. The second part of the file is a list of triangles using these points. The values X0, Y0, Z0, and so on are floats specifying the X/Y/Z coordinates of points.

```
VOLUME=V
<integerNumPoints>
X0 Y0 Z0
X1 Y1 Z1
...
<integerNumTriangles>
T0P1 T0P2 T0P3 Tag0
T1P1 T1P2 T1P3 Tag1
...
```

The values T0P1, T0P2, T0P3, and so on are integers identifying points from the first part of the file. (Note that the identification of points is zero based with the first point numbered zero.) The values Tag0, Tag1, and so on are integer tag values. If a tag is omitted it defaults to one. (The way negative tags are used was selected so that a terrain with no explicitly specified tags is useful as the boundary of a locale.

As an example, the following shows a trivial 2-triangle terrain.

```
VOLUME=0.25
4
0.0 0.0 0.0
0.0 0.0 1.0
1.0 0.0 0.0
0.0 1.0 0.0
2
0 1 2  1
4 1 2  4
```

28.1 spTerrainNew

```
sp spTerrainNew(spFixedAscii URL)
```

URL - URL of spTerrain to be created.
Return value - The newly constructed object.

Creates a new object of the class spTerrain, with the local process as its owner. In addition, sets the URL of the spTerrain as specified.

28.2 spTerrainBelow

```
long spTerrainBelow(sp Terrain, spVector P, spVector Q)
```

Terrain - The spTerrain used as the reference.
P - The point that may be above the terrain.
Q - Modified to reflect position below P on terrain.
Return value - Non-zero if there is a terrain point below the input position.

This is basically identical to spBoundaryBelow except that it operates on the more detailed and complex information specified by an spTerrain object. spTerrainBelow locates the highest (along the Y axis) triangle with a non-zero tag that is below the query point. If there is such a tag and it is positive, it is returned. Otherwise zero is returned.

28.3 spTerrainAbove

```
long spTerrainAbove(sp Terrain, spVector P, spVector Q)
```

Terrain - The spTerrain object used as a reference.
P - The point that may be below the terrain.
Q - Modified to reflect position above P on boundary.
Return value - Non-zero if there is a terrain point above the input point.

This is basically identical to spBoundaryAbove except that it operates on the more detailed and complex information specified by an spTerrain object. spTerrainAbove locates the lowest (along the Y axis) triangle with a non-zero tag that is above the query point. If there is such a tag and it is positive, it is returned. Otherwise zero is returned.

28.4 spTerrainInside

```
long spTerrainInside(sp Terrain, spVector P)
```

Terrain - The spTerrain used as the reference.
P - The point that may be within the terrain.
Return value - Non-zero if the input position is inside the boundary.

This is basically identical to spBoundaryInside except that it operates on the more detailed and complex information specified by an spTerrain object. spTerrainInside determines whether a query point is in the interior of a terrain. If spTerrainAbove and spTerrainBelow both return non-zero given the query point, then spTerrainInside returns what spTerrainBelow returned. Otherwise it returns zero.

28.5 spTerrainReadData (Internal)

```
void spTerrainReadData(sp Link)
```

Link - The spTerrain link whose data is to be read in.
Return value - There is no return value.

Reads in terrain data and stores it in memory in an internal format. The other spTerrain methods depend on this ReadData function always being used to load in spBoundary data.

29 spClass (Fundamental)

```
public class spClass implements spLinking
```

An spClass object describes the layout of the data in the memory representation of a shared object class. The URL of an spClass object identifies a file that contains this information. This file is created by the SPOT (Section 1.6) Java preprocessor supporting the definition of shared classes. The Data variable of an spClass object is a vector of instance variable descriptors that specify information about the shared and native instance variables defined for the class.

spClass objects are used extensively in the internal operation of the system. When writing applications they have important uses as well, but primarily only as opaque identifiers that are passed as arguments to functions like spClassExamine (Section 29.23).

In addition, knowing the identity of the class descriptor of an object is very useful for runtime dispatching in an application. For example, an application might perform the following test to determine whether an object X is a direct instance of the class spThing.

```
spClassEq(spGetClass(X), spThingC())
```

(The above expression returns False if X is an instance of a subclass of spThing.) An application can perform the following test to determine whether X is an instance of spThing or any subclass of spThing.

```
spClassLeq(spGetClass(X),spThingC())
```

The shared class spClass inherits all the instance variables and functions of the class spLinking (Section 18). The class spClass defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spClassC - Class descriptor (Section 15.1).
ClassName - Name string (Section 29.1).
LoadData - Triggers reading of link data (Section 29.2).
ReadDataFn - Data reading function (Section 29.3).
SelectFn - Action function (Section 29.4).
Superclasses - Containing class (Section 29.5).
NumSuperclasses - Number of superclasses (Section 29.6).
Size - Minimum memory size (Section 29.7).
Level - Depth of class in class hierarchy (Section 29.8).
LocalOffset - Local data offset (Section 29.9).
SharedOffset - Shared data offset (Section 29.10).
SharedBitNum - Highest shared bit used (Section 29.11).
LocalBitNum - Highest local bit used (Section 29.12).
TimeStampOffsets - Offsets of spTimeStamp variables in object (Section 29.13).
SendViaLocale - Indicates instances are communicate via locales (Section 29.14).
SendViaTCP - Indicates instances are communicate via TCP (Section 29.15).
NumVariables - Number of instance variables (Section 29.16).
MethodTable - Table of pointers to methods (Section 29.17).

The class spClass defines the following functions:

spClassNewObj - Creates object instance. (Section 29.18).
spClassNewLink - Creates link instance. (Section 29.19).
spClassNew - Creates spClass given URL (Section 29.20).
spClassEq - Tests if two classes are equal (Section 29.21).
spClassLeq - Tests if class is a subclass of another class (Section 29.22).
spClassExamine - Looks at current objects (Section 29.23).
spClassMonitor - Looks at current and future objects (Section 29.24).
spClassReadData - Reads class description (Section 29.25).

Note that the various instance variables above primarily specify information only about the instance variables of a shared class as opposed to about methods. The reason for this is that shared classes are primarily passive data structures. It is therefore better in many ways to view the world model as a database rather than a collection of objects. The key focus is on communicating this data, not on the complex interaction of methods. Users can define new classes with as many new data variables to be communicated as they like.

It is possible to define methods when defining a new shared class; however, except for a vary few methods that the system calls nothing will be done internally with these methods. (An application may take good advantage of them, however.) As in any extendable object oriented environment information is maintained at run time about which methods are associated with which classes so that the can be accesses appropriately.

This document describes the various built-in shared classes. These built-in classes are specified in a special way and are automatically loaded when the world model is initialized. Except for a few internal objects that are not visible to an application, the spClass objects corresponding to these classes are the only objects that exist in the world model immediately after a world model is created using spWMNew (Section 5.14).

New classes defined by an application are handled in exactly the same way as any other kind of link object. They must be created by some process and are then communicated to other processes. All the variables of the class spClass itself are local. Most of them are specified by the spClass data file. A few can be specified by the local process.

A peculiarity of the class hierarchy is that the class of the class spClass is spClass. This circularity is a logical paradox, but does not present any practical problems.

29.1 ClassName

```
transient public String ClassName; //* [spAscii32:32] readonly

    spAscii32 spClassGetClassName(sp Object)

    void spClassiSetClassName(sp Object, spAscii32 X)

    spAscii32 sqClassGetClassName(sp Object)
    void sqClassSetClassName(sp Object, spAscii32 X)
```

For debugging convenience, spClass objects have a local instance variable called the *ClassName* that contains an ASCII representation of the name of the class.

The ClassName is a string no more than 31 characters long. It is set by spClassReadData when a class is loaded from a class descriptor file and cannot be changed. Information about the ClassName is maintained separately in each process.

29.2 LoadData

```
transient public boolean LoadData; //* [spBoolean]

      spBoolean spClassGetLoadData(sp Object)
      void spClassSetLoadData(sp Object, spBoolean X)

      spBoolean sqClassGetLoadData(sp Object)
      void sqClassSetLoadData(sp Object, spBoolean X)
```

If an spClass object describes an spLinking class, then the *LoadData* local variable specifies whether the data corresponding to links in that class should be fetched and loaded into memory. By default the LoadData bit is False, except that the system sets it to True for the classes spClass, spLocale, spBoundary, and every subclass of these classes. LoadData must be True for these kinds of links because the system itself uses the data pointed to by these links. If a process wants to use the data in any other kind of link, it must first set LoadData True for that kind of link. Information about the LoadData is maintained separately in each process.

29.3 ReadDataFn (Fundamental)

```
transient public int ReadDataFn; //* [void *] internal

      void * spClassiGetReadDataFn(sp Object)
      void spClassiSetReadDataFn(sp Object, void * X)

      void * sqClassGetReadDataFn(sp Object)
      void sqClassSetReadDataFn(sp Object, void * X)
```

Each kind of spLinking class defines a ReadData function that can be used to read the data corresponding to a link. However, a local process may want to read the data in a special way. (For example, spVisual has to read spVisualDefinition link data into the scene graph representation appropriate for the render being used.) An application can override the ReadData function defined as part of a class by storing a pointer to a locally defined function in the ReadDataFn variable of the associated spClass descriptor. This local function must take the same arguments as any ReadData function and must obey the same restrictions.

The ReadDataFn is initially set to Null, indicating that the ReadData function defined by the class should be used. An application can set it to a different locally defined function. Information about the ReadDataFn is maintained separately in each process.

29.4 SelectFn (Fundamental)

```
transient public int SelectFn; //* [void *] internal

      void * spClassiGetSelectFn(sp Object)
      void spClassiSetSelectFn(sp Object, void * X)

      void * sqClassGetSelectFn(sp Object)
      void sqClassSetSelectFn(sp Object, void * X)
```

Each kind of spMultilinking class defines a Select function that can be used to which data corresponding to a multilink should be used. However, a local process may want to make a selection in a special way. (For example, spVisual has to make a selection based on the capabilities of the render being used.) An application can override the Select function defined as part of a class by storing a pointer to a locally defined function in the SelectFn variable of the associated spClass descriptor. This local function must take the same arguments as any Select function and must obey the same restrictions.

The SelectFn is initially set to Null, indicating that the Select function defined by the class should be used. An application can set it to a different locally defined function. Information about the SelectFn is maintained separately in each process.

29.5 Superclasses (Internal)

```
transient public int Superclasses; //* [void *] readonly internal

      void * spClassiGetSuperclasses(sp Object)
      void spClassiSetSuperclasses(sp Object, void * X)

      void * sqClassGetSuperclasses(sp Object)
      void sqClassSetSuperclasses(sp Object, void * X)
```

Except for the root class sp, every shared class has at least one superclass that is another shared class. The superclasses are stored in a local instance variable called the *Superclasses*.

The Superclasses of an spClass object point to a vector of pointers to other spClass objects. The only exception to this is that the Superclass of the class sp is Null. The Superclasses are set by spClassReadData when a class is loaded from a class descriptor file and cannot be changed. Information about the Superclasses is maintained separately in each process.

29.6 NumSuperclasses (Internal)

```
transient public short NumSuperclasses; //* [short] readonly internal

      short spClassiGetNumSuperclasses(sp Object)
      void spClassiSetNumSuperclasses(sp Object, short X)

      short sqClassGetNumSuperclasses(sp Object)
      void sqClassSetNumSuperclasses(sp Object, short X)
```

Except for the root class sp, every shared class has at least one superclass that is another shared class. The number of superclasses is stored in a local instance variable called the *NumSuperclasses*.

The NumSuperclasses of an spClass object is a non-negative integer. The NumSuperclasses is set by spClassReadData when a class is loaded from a class descriptor file and cannot be changed. Information about the NumSuperclasses is maintained separately in each process.

29.7 Size (Internal)

```
transient public short Size; //* [short] internal

      short spClassiGetSize(sp Object)
      void spClassiSetSize(sp Object, short X)

      short sqClassGetSize(sp Object)
      void sqClassSetSize(sp Object, short X)
```

The *Size* local instance variable of an spClass specifies the minimum amount of memory that must be allocated when creating an object that is an instance of the class. The memory allocated is divided into five parts as follows:

(1) Space for an old copy of the shared data.

(2) Space for the local data.

(3) Space for the Marker (Section 15.5), LocalPtr (Section 15.3) and NextPtr (Section 15.4). (The pointer that identifies a shared object points to this part of the memory for the object.)

(4) Space for the shared data. (The messages send between processes consist of this part of the data plus an appropriate portion of the beginning of part (5) below.)

(5) Extra space for various purposes. One use of this space is to store spFixedAscii values.

The Size of a class is a 16-bit integer that is the sum of the lengths of parts (1) through (4) above. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the Size is maintained separately in each process.

29.8 Level (Internal)

```
transient public short Level; //* [short] internal

      short spClassiGetLevel(sp Object)
      void spClassiSetLevel(sp Object, short X)

      short sqClassGetLevel(sp Object)
      void sqClassSetLevel(sp Object, short X)
```

The *Level* local instance variable of a class contains the minimum depth of the class in the class hierarchy. It is used to speed up testing of whether one class is a subclass of another. It must be a minimum because a given class can have several superclasses.

The Level instance variable of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the Level is maintained separately in each process.

29.9 LocalOffset (Internal)

```
transient public short LocalOffset; //* [short] internal

      short spClassiGetLocalOffset(sp Object)
      void spClassiSetLocalOffset(sp Object, short X)

      short sqClassGetLocalOffset(sp Object)
      void sqClassSetLocalOffset(sp Object, short X)
```

The *LocalOffset* local instance variable of a class contains the number of bytes between the beginning of the entire block of data corresponding to an instance of the class and the beginning of the local data.

The LocalOffset of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the LocalOffset is maintained separately in each process.

29.10 SharedOffset (Internal)

```
transient public short SharedOffset; //* [short] internal

      short spClassiGetSharedOffset(sp Object)
      void spClassiSetSharedOffset(sp Object, short X)

      short sqClassGetSharedOffset(sp Object)
      void sqClassSetSharedOffset(sp Object, short X)
```

The *SharedOffset* local instance variable of a class contains the number of bytes between the beginning of the entire block of data corresponding to an instance of the class and the beginning of the place where the LocalPtr and NextPtr reside.

The SharedOffset of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the SharedOffset is maintained separately in each process.

29.11 SharedBitNum (Internal)

```
transient public short SharedBitNum; //* [short] internal

      short spClassiGetSharedBitNum(sp Object)
      void spClassiSetSharedBitNum(sp Object, short X)

      short sqClassGetSharedBitNum(sp Object)
      void sqClassSetSharedBitNum(sp Object, short X)
```

The *SharedBitNum* local instance variable of a class is a bit mask that identifies the highest used bit in the SharedBits variable for the class. (This information is needed when defining subclasses of the class in question.)

The SharedBitNum of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the SharedBitNum is maintained separately in each process.

29.12 LocalBitNum (Internal)

```
transient public short LocalBitNum; //* [short] internal

      short spClassiGetLocalBitNum(sp Object)
      void spClassiSetLocalBitNum(sp Object, short X)

      short sqClassGetLocalBitNum(sp Object)
      void sqClassSetLocalBitNum(sp Object, short X)
```

The *LocalBitNum* local instance variable of a class is a bit mask that identifies the highest used bit in the LocalBits variable for the class. (This information is needed when defining subclasses of the class in question.)

The LocalBitNum of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the LocalBitNum is maintained separately in each process.

29.13 TimeStampOffsets (Fundamental and Internal)

```
transient public int TimeStampOffsets; //* [int *] internal

      int * spClassiGetTimeStampOffsets(sp Object)
      void spClassiSetTimeStampOffsets(sp Object, int * X)

      int * sqClassGetTimeStampOffsets(sp Object)
      void sqClassSetTimeStampOffsets(sp Object, int * X)
```

The *TimeStampOffsets* local instance variable of a class is a pointer to a zero terminated vector of offsets of the instance variables of the class that contain spTimeStamp values or Null if the class has no spTimeStamp instance variables. (This information is needed when adjusting time stamps in messages received to correct for clock differences.)

The TimeStampOffsets is of type a pointer to 32-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the TimeStampOffsets is maintained separately in each process.

29.14 SendViaLocale (Fundamental and Internal)

```
transient public boolean SendViaLocale; //* [spBoolean] internal

      spBoolean spClassiGetSendViaLocale(sp Object)
      void spClassiSetSendViaLocale(sp Object, spBoolean X)

      spBoolean sqClassGetSendViaLocale(sp Object)
      void sqClassSetSendViaLocale(sp Object, spBoolean X)
```

The *SendViaLocale* local instance variable of a class specifies whether instances of the class are communicated by the ordinary locale-based communication methods.

The SendViaLocale instance variable of a class is a boolean value. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the SendViaLocale is maintained separately in each process.

29.15 SendViaTCP (Fundamental and Internal)

```
transient public boolean SendViaTCP; //* [spBoolean] internal

      spBoolean spClassiGetSendViaTCP(sp Object)
      void spClassiSetSendViaTCP(sp Object, spBoolean X)

      spBoolean sqClassGetSendViaTCP(sp Object)
      void sqClassSetSendViaTCP(sp Object, spBoolean X)
```

The *SendViaTCP* local instance variable of a class specifies whether instances of the class are communicated by TCP to server processes. Exactly what kind of server a given class is sent to is built into the system core. For example beacons and beacon examines are sent to beacon servers. Note that some kinds of objects (e.g., interval callbacks) are not sent anywhere.

The SendViaTCP instance variable of a class is a boolean value. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the SendViaTCP is maintained separately in each process.

29.16 NumVariables (Internal)

```
transient public int NumVariables; //* [long] internal

      long spClassiGetNumVariables(sp Object)
      void spClassiSetNumVariables(sp Object, long X)

      long sqClassGetNumVariables(sp Object)
      void sqClassSetNumVariables(sp Object, long X)
```

The *NumVariables* local instance variable of a class contains the number of instance variables defined for the class, including those inherited from the Superclasses. The NumVariables of a class is a 16-bit integer. It is set by spClassReadData when a class is loaded and cannot be changed. Information about the NumVariables is maintained separately in each process.

29.17 MethodTable (Fundamental and Internal)

```
transient public int MethodTable; //* [void *] readonly internal

      void * spClassiGetMethodTable(sp Object)
      void spClassiSetMethodTable(sp Object, void * X)

      void * sqClassGetMethodTable(sp Object)
      void sqClassSetMethodTable(sp Object, void * X)
```

The *MethodTable* local instance variable of a class points to a table of pointers to functions implementing methods. This table includes those inherited from the Superclasses. The MethodTable of a class is a pointer to a table of pointers to subroutines implementing methods. It (and the table it points to) are initialized by spClassReadData when a class is loaded and cannot be changed. Information about the MethodTable is maintained separately in each process.

29.18 spClassNewObj (Fundamental)

```
sp spClassNewObj(sp ClassDescriptor, long Extra)
```

ClassDescriptor - spClass of object to create.
Extra - Bytes of spFixedAscii storage needed.
Return value - The newly constructed object.

This is a general operation for creating an instance of any shared object class. Given an spClass descriptor of a shared object class, spClassNewObj creates an instance of the class. If a shared object class has (either directly or by inheritance) a string variable with the C type spFixedAscii, storage for the string value must be allocated at the moment the object is created. The Extra argument specifies how many characters of storage are needed.

When the storage for an object has been allocated, the values of all instance variables are set to all bits zero. After this, the Initialization functions are called for the class and all its ancestors in order to properly initialize any variables that need to have non-zero values. The Initialization functions are called starting with spInitialization and then working down to the most specific class, so that the Initialization function for a class can override the Initialization function for its Superclasses.

29.19 spClassNewLink (Fundamental)

```
sp spClassNewLink(sp ClassDescriptor, spFixedAscii URL)
```

ClassDescriptor - spClass of link object to create.
URL - URL of link object.
Return value - The newly constructed object.

This is a general operation for creating an instance of a subclass of spLinking or spMultilinking. Given an spClass descriptor of a link class, spClassNewLink creates an instance of the class by calling spClassNewObj. The URL argument specifies the URL to use when creating the object. This is stored in the link created and then the ReadData function for the link is called. This is needed so that the Data in the link will be loaded immediately when the link is created. Just as the New functions for objects in general should start by calling spClassNewObj, the New functions for links should start by calling spClassNewLink.

29.20 spClassNew

```
sp spClassNew(spFixedAscii URL)
```

URL - URL of the spClass to be created.
Return value - The newly constructed object.

Creates a new object of the class spClass, with the local process as its owner. In addition, sets the URL of the spClass as specified.

29.21 spClassEq

```
spBoolean spClassEq(sp ClassA, sp ClassB)
```

ClassA - An spClass to compare.
ClassB - An spClass to compare.
Return value - True if the wo classes are equal.

This predicate tests whether two spClass objects represent the same class. True is returned if the two spClasses being compared represent the same class, and False otherwise. This is not the same as just using the operator == because two spClass objects with the same URL represent the same class even if they are distinct objects. It is very important that == never be used to compare spClass objects.

29.22 spClassLeq

```
spBoolean spClassLeq(sp Subclass, sp Superclass)
```

Subclass - An spClass that might be a subclass.
Superclass - An spClass that might be a superclass.
Return value - True if class is equal to (a sublcass of) Superclass.

This predicate tests for subclass relationships between spClass class descriptors. True is returned if the class being tested is either equal to Superclass or a descendent of Superclass in the shared class hierarchy. For example:

```
spClassLeq(spGetClass(spAvatorNew()), spThingGetC()) == TRUE
spClassLeq(spThingGetC(), spThingGetC()) == TRUE
spClassLeq(spLinkGetC(), spThingGetC()) == FALSE
```

29.23 spClassExamine (Fundamental)

```
spBoolean spClassExamine(sp C, spMask Mask, spFn F, void * Data)
```

C - The class whose objects are to be examined.
Mask - spMask (Section 7) limiting the objects considered.
F - Operation (Section 6) used to examine the objects.
Data - Passed to the operation each time it is called (modifiable).
Return value - True if examining stopped because F returned True.

Applies an operation F to all the objects in a class. This includes objects that are instances of subclasses of the class as well as instances of the class itself. However, it only includes objects that are current in the local world model copy and it only includes objects that are compatible with the specified Mask. There is no guarantee of the order in which F will be applied to objects. If F ever returns True, then examining ceases and spClassExamine returns True. Otherwise False is returned.

29.24 spClassMonitor

```
sp spClassMonitor(sp C, spMask Mask, spFn F, void * Data)
```

C - Monitored objects are instances of this class or its subclasses.
Mask - spMask (Section 7) limiting the objects considered.
F - Operation (Section 6) used to monitor the objects.
Data - Passed to the operation each time it is called (modifiable).
Return value - The interval callback produced, if any.

Applies an operation to present and future objects in the local world model copy that are in the indicated class (and its subclasses) and are compatible with Mask. If F ever returns True, then monitoring ceases.

This is done by first examining the objects that currently exist and then creating an alerter (Section 43) that applies F to objects that appear in the future. If a alerter is produced, it is returned. If an alerter is returned, monitoring can be terminated by removing the alerter object.

The following shows how spClassMonitor is implemented using spClassExamine and a alerter.

```
sp spClassMonitor(sp C, spMask mask, spFn F, void * Data) {
  sp A;
  if (spClassExamine(C, Mask, F, Data)) return NULL;
  A = spAlerterNew(C, -1, spJustNew, NULL, F, Data);
  spAlerterSetMask(A, Mask);
  return A;
}
```

29.25 spClassReadData (Fundamental and Internal)

```
void spClassReadData(sp Link)
```

Link - The spClass link whose Data is to be read in.
Return value - There is no return value.

Fills in the Data in an spClass object based on the information in the file specified by the URL. This includes filling in all the instance variables of an spClass (as distinct from a link in general). The Data variable of an spClass object is a vector of instance variable descriptors that specify information about the instance variables defined for the class. The system depends on this ReadData function always being used to load in class data.

30 spThing

```
public class spThing implements spPositioning, spDisplaying
```

This class is used to represent objects in the virtual world created by an application, as opposed to objects that are part of the way the software platform operates internally. The key defining characteristics of spThing objects are that they have positions and visual definitions and occupy volumes of space.

The shared class spThing inherits all the instance variables and functions of the classes: spPositioning (Section 16) and spDisplaying (Section 17). The class spThing defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spThingC - Class descriptor (Section 15.1).

The class spThing defines the following functions:

spThingNew - Create new spThing object. (Section 15.28).

31 spRoot

```
public class spRoot extends spThing
```

An spRoot is an spThing that corresponds to a logical whole. For example, if the torso was the highest level object in an articulated humanoid figure, then it should be created as an spRoot object rather than merely an spThing. In addition, it is expected that in general, only spRoot objects will have locales as their direct parents. (It is permissible to put an spRoot under another object as long as it is logically possible that the object might sometimes move on its own.)

The class spRoot is included because of its fundamental significance to computer simulations that want to interact with assemblages of objects in the virtual world.

The shared class spRoot inherits all the instance variables and functions of the classes: spThing (Section 30), spPositioning (Section 16) and spDisplaying (Section 17). The class spRoot defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spRootC - Class descriptor (Section 15.1).

The class spRoot defines the following functions:

spRootNew - Create new spRoot object. (Section 15.28).

For the convenience of the application writer, spPositioningLocalize is automatically called every cycle on every locally owned spRoot object. This is done with the Destination argument Null and the ChooseSmallest argument False. In general, application writers can control when spPositioningLocalize is called merely be deciding which objects should be spRoot objects. The typical application need not contain any calls on spPositioningLocalize.

32 spAvatar

```
public class spAvatar extends spRoot
```

The class spAvatar is a specialization of spRoot that corresponds to an active entity. That is to say, it is the avatar of a human user or the avatar of a computer simulated agent that wishes to interact in more-or-less the same way as if it were a human user.

Internally, the system does nothing special with instances of the class spAvatar. Rather, the class is included because of its fundamental significance to computer simulations that want to interact with human users and other computer simulated agents.

The shared class spAvatar inherits all the instance variables and functions of the classes: spRoot (Section 31), spThing (Section 30), spPositioning (Section 16) and spDisplaying (Section 17). The class spAvatar defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spAvatarC - Class descriptor (Section 15.1).
shared **IsBot** - True if avatar is not representing a person (Section 32.1).

The class spAvatar defines the following functions:

spAvatarNew - Create new spAvatar object. (Section 15.28).

32.1 IsBot

```
public boolean IsBot; //* [spBoolean]

      spBoolean spAvatarGetIsBot(sp Object)
      spBoolean spAvatarGetOldIsBot(sp Object)
      void spAvatarSetIsBot(sp Object, spBoolean X)

      spBoolean sqAvatarGetIsBot(sp Object)
      spBoolean sqAvatarGetOldIsBot(sp Object)
      void sqAvatarSetIsBot(sp Object, spBoolean X)
```

The *IsBot* shared instance variable of an spAvatar specifies whether the avatar represents a human being or a computer simulation. Specifically, if the IsBot bit is True, then the avatar represents a computer simulation of some kind. If the IsBot bit is False, then the avatar represents a human user.

The IsBot bit of an spAvatar is given a default value of False. Information about the IsBot bit is shared between processes.

33 spAudioSource (Fundamental)

```
public class spAudioSource implements spPositioning, spDisplaying, spAudioParameters
```

spAudioSource objects are used to represent sources of sound in the virtual world. They are used in two primary ways. Most simply, you can connect up with a processes that captures sound with a microphone by creating an spSpeaking object (Section 36). This indirectly leads to the creation of an appropriate spAudioSource object. Alternatively, you can create spAudioSource objects yourself and play sound effects through them using spSoundPlay (Section 25.3). Since spAudioSource is a subclass of spPositioning, sound sources can be moved around and oriented like any other spThing.

The shared class spAudioSource inherits all the instance variables and functions of the classes: spPositioning (Section 16), spDisplaying (Section 17) and spAudioParameters (Section 23). The class spAudioSource defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spAudioSourceC - Class descriptor (Section 15.1).
Duration - Duration of packets to be read (Section 33.1).
ExternalFormat - Format of sound data when communicated (Section 33.2).

The class spAudioSource defines the following functions:

spAudioSourceSetup - Prepares for writing (Section 33.3).
spAudioSourceWrite - Writes sound data through audio source (Section 33.4).
spAudioSourceRead - Reads sound data (Section 33.5).
spAudioSourceNew - Create new spAudioSource object. (Section 15.28).

spAudioSources can be localized point sources or diffuse sources. This is specified using the InRadius and OutRadius variables inherited from spDisplaying. If the InRadius is zero (its default value), then the source is considered to be a point source. Sound drops off in intensity as you move away and is localized to the direction of the source. If the InRadius is not zero, then the source is treated as diffuse—coming with equal intensity from every point within the InRadius of the source. Outside the InRadius, the sound is localized to the direction of the source and drops off to reach zero at the OutRadius.

33.1 Duration

```
transient public int Duration; //* [spDuration] readonly

      spDuration spAudioSourceGetDuration(sp Object)

      void spAudioSourceiSetDuration(sp Object, spDuration X)

      spDuration sqAudioSourceGetDuration(sp Object)
      void sqAudioSourceSetDuration(sp Object, spDuration X)
```

spAudioSource objects have a shared instance variable called the *Duration*, which specifies the lengths of the chunks of sound data that can be read using spAudioSourceRead. This is specified in milliseconds.

The Duration is initialized to zero when an spAudioSource is created, which has the meaning that audio is not being written or read via the spAudioSource. It is set by the system when spAudioSourceSetup is called. It should not be modified directly by application programs. Information about the Duration is maintained separately in each process.

33.2 ExternalFormat (Internal)

```
transient public long ExternalFormat; //* [spFormat] internal

      spFormat spAudioSourceiGetExternalFormat(sp Object)
      void spAudioSourceiSetExternalFormat(sp Object, spFormat X)

      spFormat sqAudioSourceGetExternalFormat(sp Object)
      void sqAudioSourceSetExternalFormat(sp Object, spFormat X)
```

spAudioSource objects have a local instance variable called the *ExternalFormat*, which specifies the format (i.e., encoding, samples per second, etc.) used when communicating the data between process.

This format is of type spFormat (Section 14). It is initialized to zero, which has the meaning that audio is not being written or read via the spAudioSource. The ExternalFormat is set by the system when spAudioSourceSetup is called and should not be modified directly by application programs. Information about the ExternalFormat is maintained separately in each process.

33.3 spAudioSourceSetup

```
void spAudioSourceSetup(sp S, spBoolean W, spBoolean R, spFormat F, spDuration D)
```

S - spAudioSource to be initialized.
W - True enables writing.
R - True enables reading.
F - Format to use for audio data in memory.
D - Audio chunk size in milliseconds.
Return value - There is no return value.

Sets up the buffering and communication that is needed to support audio I/O. If W is True, the source must be owned by spWMGetMe and things are set up so that spAudioSourceWrite can be called. If R is True, things are set up so that spAudioSourceRead can be called. If neither are True, audio buffering and communication are shut down, which saves significant system resources.

Things are set up so that the next chunk of data written and/or read will begin at the current moment. If you want to cease writing and/or reading sound at some point, you can simply stop calling spAudioSourceWrite and spAudioSourceRead. However, it is better to call spAudioSource-Setup with the W and R arguments both False. In any event, you must call spAudioSourceSetup again to resynchronize I/O with the current moment before resuming writing and/or reading.

The Format and Duration of the spAudioSource are set to F and D respectively. Either value can be specified as zero, in which case the system picks a reasonable value to use. For the Format, this is spFormat8LINEAR16MONO (Section 14), which specifies 8000, 16-bit, linearly encoded, mono samples per second. The default Duration is 40 milliseconds. The ExternalFormat is set to a reasonable value by the system, such as 8000, 8-bit, mu-law encoded, mono samples per second.

As discussed in ["Time Synchronization in Spline", MERL technical report 96-09, April 1996], a critical requirement is that for every chunk of sound read or written the duration in milliseconds must be a value that corresponds to an exact integer number of samples. If the rate is divisible by 1000, then this requirement is trivial to satisfy. However, when this is not the case, the restriction can be significant. For example, CD quality sound is at the rate 44100, which implies that the Duration must be a multiple of 10 milliseconds.

To make buffering work efficiently, it is further required that the Duration in an spAudioSource must be a divisor of a specially chosen internal number (a week in milliseconds). Since this number has many divisors, this places only a small additional constraint on durations (basically they cannot have any large prime divisors).

Lastly, to make communication easy, the Duration is required to be short enough so that the whole chunk of data will fit into a single UDP message.

If the Duration passed to spAudioSourceSetup does not meet the criteria above, it is changed by to the nearest duration (above or below) that does meet the criteria.

33.4 spAudioSourceWrite

```
void spAudioSourceWrite(sp Source, char * Data)
```

Source - spAudioSource to write data through.
Data - Sound data to be sent.
Return value - There is no return value.

Writes a chunk of data through an spAudioSource. The source must be an spAudioSource owned by spWMGetMe. The Format and Duration of the Data must be as specified when spAudioSourceSetup was called.

It is required that the data must be in the future i.e., begin now or later. However, due to limited buffer sizes, the data must in general end less than a second in the future.

Each time spAudioSourceWrite is called, it is assumed that the data being transmitted exactly follows the previous data transmitted. You are responsible for supplying the data before the time it is needed, (but not more than a second before) with a margin for error of not more than a few milliseconds outside these bounds. (Note you can only be writing one sequence of data chunks at a time through a source.)

33.5 spAudioSourceRead

```
char * spAudioSourceRead(sp Source)
```

Source - The spAudioSource to read data from.
Return value - The sound data read.

Reads a chunk of sound data from an spAudioSource. The data returned has the Format and Duration specified when spAudioSourceSetup was called. It is identical to the data that was written through the spAudioSource by the owner of the spAudioSource object, except that a format conversion may have been applied and periods of very quiet sound may have been forced to be exactly zero. (Significant bandwidth can be saved by omitting the communication of blocks of data corresponding to silence.)

It is required that the data read must refer to the past i.e., end now or earlier. (For instance, you must wait at least Duration milliseconds before reading the first chunk of data form an spAudioSource after calling spAudioSourceSetup.) However, due to limited buffer sizes, the data must in general begin less than a second in the past.

Each time spAudioSourceRead is called, it is assumed that the data being requested exactly follows the previous data requested. You are responsible for asking for the data after the time it is ready, (but not more than a second after) with a margin for error of not more than a few milliseconds beyond these bounds.

34 spBeacon

```
public class spBeacon implements spBeaconing
```

This class is the minimal embodiment of spBeaconing. It is needed because one cannot directly make an instance of spBeaconing itself. spBeacon is appropriate to use when one merely wants to label something without specifying an other information. Various subclasses of spBeacon specify additional information.

The shared class spBeacon inherits all the instance variables and functions of the class spBeaconing (Section 20). The class spBeacon defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spBeaconC - Class descriptor (Section 15.1).

The class spBeacon defines the following functions:

spBeaconNew - Creates beacon given URL (Section 34.1).

34.1 spBeaconNew

```
sp spBeaconNew(spFixedAscii Tag)
```

Tag - The tag of the spBeacon to be created.
Return value - The newly constructed object.

Creates a new object of the class spBeacon, with the local process as its owner. In addition, sets the tag of the spBeacon as specified.

Beacon tags must include a DNS name specification. To ensure this, the following default is applied to the Tag string provided. If the tag begins with just one / or a colon : then it is prefixed with //<HostName>. Where <HostName> is the DNS name of the machine the process is running on. If the tag does not begin with / or colon then it is prefixed with //<HostName>/. That is to say:

```
//string => //string
/string => //<HostName>/string
:string => //<HostName>:string
string => //<HostName>/string
=> //<HostName>/
```

Note that the defaulting to <HostName> is only useful for beacons that are used to communicate between (or within) processes on a single machine. If a beacon is to be used to communicate between machines, an explicit host name must be specified.

Note that in certain situations, e.g., when running as a high security Java plugin, it may not be possible for the system to determine the local host name. However, in this situation, the name "localhost" can be used and will suffice.

35 spPositionedBeacon

```
public class spPositionedBeacon implements spBeaconing, spPositioning
```

This class extends spBeacon by adding positional information. It allows one to create a beacon that specifies a particular location, e.g., in a locale. This is the kind of beacon that is typically used as the target of an spBeaconGoto.

The shared class spPositionedBeacon inherits all the instance variables and functions of the classes: spBeaconing (Section 20) and spPositioning (Section 16). The class spPositionedBeacon defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spPositionedBeaconC - Class descriptor (Section 15.1).

The class spPositionedBeacon defines the following functions:

spPositionedBeaconNew - Create new spPositionedBeacon object. (Section 15.28).

36 spSpeaking

```
public class spSpeaking extends spPositionedBeacon
```

The audio renderer has the built-in capability of creating a sound source corresponding to the sound captured by the microphone on a machine. To use this capability, you create an spSpeaking beacon. (The name is mnemonic for the fact that you use this kind of beacon as the 'mouth' of an avatar.) The Transform of an spSpeaking beacon specifies the position and orientation of the sound created by the user.

The shared class spSpeaking inherits all the instance variables and functions of the classes: spPositionedBeacon (Section 35), spBeaconing (Section 20) and spPositioning (Section 16). The class spSpeaking defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spSpeakingC - Class descriptor (Section 15.1).

The class spSpeaking defines the following functions:

spSpeakingNew - Creates microphone beacon (Section 36.1).

Upon seeing the creation of an spSpeaking beacon, the appropriate audio rendering process creates an spAudioSource sound source with the Live bit set to True and makes this source be a child of the beacon. The audio renderer then starts outputting all the sound captured by the microphone through this source. Outputting continues until the beacon is removed. To change the position of the source, one merely needs to change the position of the beacon that is its Parent.

36.1 spSpeakingNew

```
sp spSpeakingNew(spFixedAscii HostName)
```

HostName - DNS name of the audio rendering machine.
Return value - The newly constructed object.

Creates an spSpeaking beacon with a tag of the form `"//<HostName>/spAudio"` where the HostName is a DNS address string (e.g., `"server.cs.hoople.und.edu"`). If HostName is Null, the name of the machine you are on is used in the tag created.

When an audio rendering process running on machine X notices an spSpeaking beacon with the tag `"//X/spAudio"`, it creates an appropriate spAudioSource and starts to output sound. (This can be done for several spSpeaking beacons at once.)

37 spHearing

```
public class spHearing extends spPositionedBeacon implements spAudioParameters
```

The audio renderer has the built-in capability of creating a localized sound image and playing it through the headphones connected to a machine. To use this capability, you create an spHearing beacon. (The name is mnemonic for the fact that you use this kind of beacon as the 'ears' of an avatar.) The Transform of an spHearing beacon specifies the position and orientation from which sound is being observed.

The shared class spHearing inherits all the instance variables and functions of the classes: spPositionedBeacon (Section 35), spAudioParameters (Section 23), spBeaconing (Section 20) and spPositioning (Section 16). The class spHearing defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spHearingC - Class descriptor (Section 15.1).

The class spHearing defines the following functions:

spHearingNew - Creates headphones beacon (Section 37.1).

Upon seeing the creation of an spHearing beacon, the appropriate spAudio process creates an spAudioObserver and makes it a child of the beacon. It then starts routing localized sound from this observer to the headphones. This continues until the beacon is removed.

spHearing inherits from spAudio parameters so that an application can exercise control over various audio rendering parameters. As far as possible, the rendering process attempts to follow the suggests made by an application. The exact parameters it is using at any moment are reflected in the spAudioObserver it creates.

You can change the Focus, Live, and Format variables in an spHearing at any time, in which case the change will be followed (with up to a second or so delay) by the audio renderer. To change the position from which sound is being observed, you merely need to change the position of the spHearing beacon.

37.1 spHearingNew

```
sp spHearingNew(spFixedAscii HostName)
```

HostName - DNS name of the audio rendering machine.
Return value - The newly constructed object.

Creates an spHearing beacon with a tag of the form `"//<HostName>/spAudio"` where the string HostName is a DNS address string (e.g., `"server.cs.hoople.und.edu"`). If HostName is Null, the name of the machine you are on is used in the tag created.

When an audio rendering process running on machine X notices an spHearing beacon with the tag `"//X/spAudio"`, it starts routing localized sound to the headphones. (Unless it is already occupied supporting a previous spHearing beacon.)

38 spSeeing

```
public class spSeeing extends spPositionedBeacon implements spVisualParameters
```

The visual renderer has the built-in capability of rendered an image of the virtual world for the user to see. To use this capability, one creates an spSeeing beacon. (The name is mnemonic for the fact that you use this kind of beacon as the 'eyes' of an avatar.) The Transform of an spSeeing beacon specifies the position and orientation from which the scene is being observed.

The shared class spSeeing inherits all the instance variables and functions of the classes: spPositionedBeacon (Section 35), spVisualParameters (Section 22), spBeaconing (Section 20) and spPositioning (Section 16). The class spSeeing defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spSeeingC - Class descriptor (Section 15.1).

The class spSeeing defines the following functions:

spSeeingNew - Creates spSeeing beacon (Section 38.1).

Upon seeing the creation of an spSeeing beacon, the appropriate spVisual process creates an spVisualObserver and makes it a child of the beacon. It then starts creating rendered images. This continues until the beacon is removed.

spSeeing inherits from spVisualParameters so that an application can exercise control over various visual rendering parameters. As far as possible, the rendering process attempts to follow the suggests made by an application. The exact parameters it is using at any moment are reflected in the spVisualObserver it creates.

You can change the Focus, Live, and Format variables in an spHearing at any time, in which case the change will be followed (with up to a second or so delay) by the audio renderer.

You can change the control variables in the spSeeing object at any time, in which case the change will be followed as closely as possible (with up to a second or so delay) by the visual renderer. To change the viewpoint, you merely need to change the position and orientation of the spSeeing beacon.

38.1 spSeeingNew

```
sp spSeeingNew(spFixedAscii HostName)
```

HostName - DNS name of visual rendering machine.
Return value - The newly constructed object.

Creates an spSeeing beacon with a tag of the form `"//<HostName>/spVisual"` where the HostName is a DNS address string (e.g., `"server.cs.hoople.und.edu"`). If HostName is Null, the name of the machine you are on is used in the tag created.

When a visual rendering process running on machine X notices an spSeeing beacon with the tag `"//X/spVisual"`, it starts rendering. (Unless it is already occupied supporting a previous spSeeing beacon.)

Note that the above basically assumes that there will only be one visual rendering processes running on a given machine. Typically, this should be the case. However, at least for testing applications, it can be convenient to have more than one application process/visual renderer pair running on a single machine. To allow this to work, a given visual renderer will respond to an spSeeing beacon only if no other visual renderer has responded to it. This allows multiple application process/visual renderers to run correctly on a machine as long as one is completely started up before another is started.

39 spSimulationObserver

```
public class spSimulationObserver implements spObserving
```

An spSimulationObserver triggers the receipt of information that a process supporting a simulation needs to know about the world model. This includes information about all the objects in a locale. However, it does not include information that is required only for visual and audio rendering. In particular, it does not trigger the receipt of streaming audio. In addition, it does not have any parameters for summarizing what a render is doing since no renderer is involved.

The shared class spSimulationObserver inherits all the instance variables and functions of the class spObserving (Section 21). The class spSimulationObserver defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spSimulationObserverC - Class descriptor (Section 15.1).

The class spSimulationObserver defines the following functions:

spSimulationObserverNew - Create new spSimulationObserver object. (Section 15.28).

40 spVisualObserver

```
public class spVisualObserver implements spObserving, spVisualParameters
```

spVisualObserver objects have two purposes, one internal in nature and one external. First, spVisualObserver objects are created by the visual renderer spVisual in order to trigger the receipt of appropriate information as the basis for rendering. No process other than a visual renderer should ever create an spVisualObserver object. The spVisualObserver created by a renderer is made the child of the spSeeing beacon the renderer is following, so that it will echo the position and orientation of this beacon.

Second, the spVisualObserver object created can by spVisual can be observed by a user process in order to determine exactly what the visual render is doing. That is to say, to observe rendering parameters being used. Particularly, with regard to the Interval, this may different from what was requested using an spSeeing beacon.

The shared class spVisualObserver inherits all the instance variables and functions of the classes: spObserving (Section 21) and spVisualParameters (Section 22). The class spVisualObserver defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spVisualObserverC - Class descriptor (Section 15.1).

The class spVisualObserver defines the following functions:

spVisualObserverNew - Create new spVisualObserver object. (Section 15.28).

41 spAudioObserver (Fundamental)

```
public class spAudioObserver implements spObserving, spAudioParameters
```

spAudioObserver objects have two purposes, one internal in nature and one external. First, spAudioObserver objects are created by the audio renderer spAudio in order to trigger the receipt of appropriate information as the basis for rendering. Typically, no process other than an audio renderer will ever create an spAudioObserver object. The spAudioObserver created by a renderer is made the child of the spHearing beacon the renderer is following, so that it will echo the position and orientation of this beacon.

Second, the spAudioObserver object created by spAudio can be observed by a user process in order to determine exactly what the audio render is doing. That is to say, to observe rendering parameters being used. With regard to the Format, this may be different from what was requested using an spHearing beacon.

The shared class spAudioObserver inherits all the instance variables and functions of the classes: spObserving (Section 21) and spAudioParameters (Section 23). The class spAudioObserver defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spAudioObserverC - Class descriptor (Section 15.1).

The class spAudioObserver defines the following functions:

spAudioObserverInitialization - Sets Audio Bit to True. (Section 41.1).
spAudioObserverNew - Create new spAudioObserver object. (Section 15.28).

A key feature of spAudioObservers as opposed to spObserving objects in general is that spAudioObserver objects set the Audio bit to True by default.

41.1 spAudioObserverInitialization (Internal)

```
void spAudioObserverInitialization(sp Object)
```

Object - spAudioObserver object to initialize.
Return value - There is no return value.

Initializes the Audio bit of an spAudioObserver to True.

42 spIntervalCallback

```
public class spIntervalCallback extends sp
```

The class spIntervalCallback makes it easy to trigger a locally defined operation on the basis of time. Unlike other world model objects, spIntervalCallbacks are not communicated between processes by Locale-Based Communication. As a result, they operate purely in the local process. If you want an operation that is communicated between processes then you should define an action class (Section 46).

The shared class spIntervalCallback inherits all the instance variables and functions of the class sp (Section 15). The class spIntervalCallback defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spIntervalCallbackC - Class descriptor (Section 15.1).
shared **Interval** - Triggering interval (Section 42.2).
F - Operation to apply (Section 42.3).
FState - State used with operation (Section 42.4).
NextTriggerTime - Time action will be considered for application (Section 42.5).
IntNext - Forward link in list of interval triggered actions (Section 42.6).
IntPrev - Back link in list of interval triggered actions (Section 42.7).

The class spIntervalCallback defines the following functions:

spIntervalCallbackNew - Creates an interval callback (Section 42.8).

Interval callbacks get triggered based on time under the control of the Interval variable. An Interval value of N causes an action to be run every N milliseconds. A negative Interval value means that time having elapsed should never cause the callback to run. When an interval callback is triggered, it calls a user specified operation F.

Interval callbacks inherit from the class sp because several of the variables of sp (e.g., Class) and several operations of sp (e.g., Remove) are useful for callbacks as well. However, since callbacks are not subject to Locale-Based Communication, it would otherwise have made sense for callbacks to not inherit from sp.

42.1 Details of Callback processing

Interval callbacks are only called during calls on spWMUpdate. Specifically, in spWMUpdate, after all actions have been called and just after all alerters that are triggered by changes in external objects have been called, the world model is scanned looking for interval callbacks that should be applied.

It is important to note that since alerters are also interval callbacks, the following applies to them as well. In fact, during a single call on spWMUpdate, an alerter can get called both because it is an alerter and then also because it is an interval callback.

The following pseudo-code shows the logic of the operation of interval callbacks during spWMUpdate processing.

```
for each spIntervalCallback C in the local world model copy {
  if (spIntervalCallbackGetNextTriggerTime(C) <= UdateWorldModelStartTime()) {
    spFn F = spIntervalCallbackGetF(C);
    spDuration T = spIntervalCallbackGetNextTriggerTime(C) ;
    if (spGetParent(C) && spIsRemoved(spGetParent(C))) spRemove(C);
    else {
      spWMSetMe(spGetOwner(C));
      if (F(spGetParent(C), spIntervalCallbackGetFState(C))) spRemove(C);
      else {
        T = max(UdateWorldModelStartTime(),
                T + spIntervalCallbackGetInterval(C));
        spIntervalCallbackSetNextTriggerTime(C, T);
      }
      spWMSetMe(spWMGetSystemOwner());
    }
  }
}
```

There is no guarantee of the order in which interval callbacks will be considered. A list of interval callbacks ordered in terms of their NextTriggerTimes is used to allow efficient computation of which interval callbacks are applicable when.

The value of spWMGetMe is altered during the application of an interval callback so that F is prevented from changing objects owned by other processes.

An interval callback F operation is always applied to the Parent of the callback. This happens even if the Parent is an spClass object. This is different from the way alerters are handled. The reason why this is done is to guarantee that F will absolutely be called when the Interval is up. For example, it might seem more sensible to call F on every appropriate object in the class when the Parent is a class object. However, it might be the case that there were no objects in the class, in which case F would not be called at all. If the Parent is a class, F can call spClassExamine if it wants to inspect individual objects of the class.

When an object X is removed, any interval callbacks that have X as their Parents are also removed.

Since all interval callbacks are locally owned, it is not possible to find out about a callback before finding out about its Parent. It is conceivable for the Parent to later become unknown, but this is considered unlikely and no special actions are taken.

There can be considerable sloppiness with regard to when an interval callback is called after its Interval expires. However, the way the next trigger times are set based on the last desired trigger time instead of on the current time leads to good average timing behavior as long as the requested Intervals are relatively large.

The pseudo-code above shows the general way that interval callbacks are processed. The pseudo-code does not illustrate the fact that special actions are taken so that interval callbacks created by spVisual and spAudio are run at a later time, after all application code has been run. This is supported by separating interval callbacks into two groups based on their owners.

42.2 Interval (Fundamental)

```
public int Interval; //* [spDuration]

      spDuration spIntervalCallbackGetInterval(sp Object)
      spDuration spIntervalCallbackGetOldInterval(sp Object)
      void spIntervalCallbackSetInterval(sp Object, spDuration X)

      spDuration sqIntervalCallbackGetInterval(sp Object)
      spDuration sqIntervalCallbackGetOldInterval(sp Object)
      void sqIntervalCallbackSetInterval(sp Object, spDuration X)
```

spIntervalCallbacks have a shared instance variable called the *Interval* that specifies a time in milliseconds after which the spIntervalCallback should be triggered. When an spIntervalCallback is created, it is not immediately triggered, but rather only after the Interval has expired. If an spIntervalCallback that was triggered due to its Interval expiring does not remove itself from the world model, then it is requeued to trigger again after the Interval expires again. For example, an spIntervalCallback in the world model with an Interval of 100 will be triggered 10 times per second. An spIntervalCallback with an Interval of zero will be called every time spWMUpdate is called. If the Interval for an spIntervalCallback is negative, then the spIntervalCallback is never triggered based on time. (This is not useful for spIntervalCallback themselves, but is useful for spAlerters.)

The Interval for an spIntervalCallback is set when an action is created and cannot be altered later. (This is essential for efficiency. If you want to alter the value, you can do so by removing the spIntervalCallback in question and creating a corresponding new spIntervalCallback with a new value.) Information about the Interval is shared between processes.

(Actually, it is possible to alter the Interval for an spIntervalCallback, but only inside the F operation and then only when the spIntervalCallback has been triggered due to the Interval. This will effect the Interval used when requeuing the action for subsequent processing.)

An spIntervalCallback cannot be triggered by its Interval more often than once each time spWMUpdate is called. As a result, if the Interval specified for an spIntervalCallback is less than the Interval between calls on spWMUpdate, it cannot possibly be satisfied. Even if the Interval specified for an spIntervalCallback is greater than the Interval between calls on spWMUpdate, the Interval can at best only be approximated, because the actual time the spIntervalCallback is called must coincide with a call on spWMUpdate and there is considerably sloppiness in the underlying timing mechanisms that the system relies on. However, the system insures that for sufficiently large spIntervalCallback Intervals, the average timing error experienced by a repetitively applied spIntervalCallback tends toward zero.

42.3 F

```
transient public spFn F; //* [spFn] readonly

      spFn spIntervalCallbackGetF(sp Object)

      void spIntervalCallbackiSetF(sp Object, spFn X)

      spFn sqIntervalCallbackGetF(sp Object)
      void sqIntervalCallbackSetF(sp Object, spFn X)
```

spIntervalCallback objects have a local instance variable called *F* that holds the locally defined operation (Section 6) to be called when the spIntervalCallback is applied. It must be set when an spIntervalCallback is created and cannot be changed later. Information about the F is maintained separately in each process.

Each time an interval callback C is run, F is applied to the Parent of C (which in simple interval callbacks is Null) and the FState of C. F performs some computation possibly modifying the Parent and possibly accumulating a value in the FState. If F ever returns True, the interval callback is removed. Otherwise, it is requeued for execution the next time the interval expires.

The F operation for an spIntervalCallback should run quickly—i.e., in a few microseconds at most—because it is being called from inside spWMUpdate and not in a separate thread.

42.4 FState

```
transient public int FState; //* [void *] readonly

      void * spIntervalCallbackGetFState(sp Object)

      void spIntervalCallbackiSetFState(sp Object, void * X)

      void * sqIntervalCallbackGetFState(sp Object)
      void sqIntervalCallbackSetFState(sp Object, void * X)
```

spIntervalCallback objects have a local instance variable called *FState* that holds state information that is passed to the F operation when it is called. This must be set when an spIntervalCallback is created and cannot be changed later. Information about the FState is maintained separately in each process.

42.5 NextTriggerTime (Fundamental and Internal)

```
transient public int NextTriggerTime; //* [spTimeStamp] internal

      spTimeStamp spIntervalCallbackiGetNextTriggerTime(sp Object)
      void spIntervalCallbackiSetNextTriggerTime(sp Object, spTimeStamp X)

      spTimeStamp sqIntervalCallbackGetNextTriggerTime(sp Object)
      void sqIntervalCallbackSetNextTriggerTime(sp Object, spTimeStamp X)
```

spIntervalCallbacks have a local instance variable called the *NextTriggerTime* that specifies the absolute time at which the spIntervalCallback will next be triggered due to the Interval. This time is reset after each time the spIntervalCallback is triggered due to the Interval. For efficiency, spIntervalCallbacks with non-negative Intervals are maintained in a queue ordered by their NextTriggerTimes.

The NextTriggerTime of an spIntervalCallback is maintained by the system and must not be altered in any other way. Information about the NextTriggerTime is maintained separately in each process.

42.6 IntNext (Fundamental and Internal)

```
transient public int IntNext; //* [void *] internal

      void * spIntervalCallbackiGetIntNext(sp Object)
      void spIntervalCallbackiSetIntNext(sp Object, void * X)

      void * sqIntervalCallbackGetIntNext(sp Object)
      void sqIntervalCallbackSetIntNext(sp Object, void * X)
```

For efficiency, a time ordered queue is maintained containing all the spIntervalCallbacks that can be triggered based on the passage of time. This queue makes it very fast to determine which spIntervalCallbacks, if any, should be triggered at a given moment. The *IntNext* local instance variable of an spIntervalCallback is used as a forward pointer when constructing this doubly linked queue.

The IntNext variable is maintained by the system and cannot be manipulated by an application. Information about the IntNext is maintained separately in each process.

42.7 IntPrev (Fundamental and Internal)

```
transient public int IntPrev; //* [void *] internal

      void * spIntervalCallbackiGetIntPrev(sp Object)
      void spIntervalCallbackiSetIntPrev(sp Object, void * X)

      void * sqIntervalCallbackGetIntPrev(sp Object)
      void sqIntervalCallbackSetIntPrev(sp Object, void * X)
```

The *IntPrev* local instance variable of an spIntervalCallback is used as a backward pointer when constructing the doubly linked, time ordered queue of interval callbacks that can be triggered due to the passage of time. The IntPrev variable is maintained by the system and cannot be manipulated by an application. Information about the IntPrev is maintained separately in each process.

42.8 spIntervalCallbackNew

```
sp spIntervalCallbackNew(spDuration Interval, spFn F, void * FState)
```

Interval - The desired interval.
F - Desired operation (Section 6).
FState - Passed to the operation each time it is called (modifiable).
Return value - The newly constructed object.

Creates an interval callback with the specified Interval, F, and FState. The Parent of the callback is set to Null. (Therefore, F will be called with Null as its first argument.) None of the above can subsequently be changed. The Mask is set to spMaskNORMAL. You can change it to a different value. An spIntervalCallback can be prematurely terminated by using spRemove to get rid of the spIntervalCallback object.

If the spIntervalCallback operation F returns False when it is called, then the spIntervalCallback is automatically reinstalled for interval milliseconds in the future. Otherwise, it is removed.

To have an operation myFn called every ten seconds, you might do the following.

```
C = spIntervalCallbackNew(10000, myFn, NULL)
```

43 spAlerter

```
public class spAlerter extends spIntervalCallback
```

The class spAlerter lets you easily define operations and tests locally and use them as event-triggered operations that run only in the local process. In particular, an spAlerter object causes an operation to be applied to the object(s) specified by its Parent when an event occurs in the future. The primary purpose of spAlerters is to efficiently detect events that the main body of the application can then act upon.

Since they are spIntervalCallbacks, spAlerters are not communicated between processes by Locale-Based Communication. As a result, they operate purely in the local process. If you want an operation that is communicated between processes then you should define an action class (Section 46).

The shared class spAlerter inherits all the instance variables and functions of the classes: spIntervalCallback (Section 42) and sp (Section 15). The class spAlerter defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spAlerterC - Class descriptor (Section 15.1).
P - Predicate determining applicability (Section 43.2).
PState - State used in conjunction with predicate (Section 43.3).
shared **Mask** - Visibility mask of objects to act on (Section 43.4).
ChgNext - Forward link in list of change triggered actions (Section 43.5).
ChgPrev - Back link in list of change triggered actions (Section 43.6).

The class spAlerter defines the following functions:

spAlerterNew - Creates an alerter (Section 43.7).
spAlerterInitialization - Initialize object (Section 43.8).

spAlerters are triggered based on intervals (because they are spIntervalCallbacks) and by events. Whenever the Interval expires, F is applied. In addition, Whenever there has been a change in the shared instance variables of a relevant object, the spAlerter predicate P is applied to the changed object to determine whether an event of interest has occurred.

Triggering based on events is controlled by the Parent and Mask variables as well as the predicate P. Basically, the alerter is triggered when an object specified by the Parent matches the Mask and satisfies the test. If the Parent is Null, then the alerter is never triggered based on any events. If an alerter has both a non-negative Interval and a Parent, then it is triggered whenever either the Interval is satisfied or the event has occurred.

spAlerters attach special meaning to the Parent variable. If the Parent is a class descriptor, then the alerter is potentially applied to every object in that class (or its subclasses). If the Parent is not a class descriptor, then the alerter is only applied to the one object that is the Parent. The above notwithstanding, if an alerter is triggered by an Interval, then the action is applied to the Parent itself, be it a class, an object, or Null.

If the spAlerter predicate P returns True, then F is applied to the object in question as well. If F returns True, then the spAlerter is removed. If not, the spAlerter continues its monitoring.

For example, to be notified whenever any spSeeing object in the local world model moves, you might do the following.

```
A = spAlerterNew(spSeeingC(), -1, spChangedTransform, NULL, myFn, NULL)
```

You could turn this off as follows.

```
spRemove(A);
```

To be notified when any spSeeing object in the local world model moves, but in any event after ten seconds, you might do the following.

```
A = spAlerterNew(spSeeingC(), 10000, spChangedTransform, NULL, myFn, NULL)
```

To be notified purely based on an interval, use an spIntervalCallback.

43.1 Details of alerter processing

Alerters are only called during calls on spWMUpdate. Specifically, in spWMUpdate, just after all actions have been called and just before interval callbacks are applied, alerters that are triggered by changes in external objects are called. Later, after all application code has been run, alerters that are triggered by changes in local objects are called.

The following pseudo-code shows the logic of the operation of alerters during spWMUpdate processing. It is important to note that since alerters are also interval callbacks, they are also processed like any other interval callback (Section 42.1). In fact, during a single call on spWMUpdate, an alerter can get called both because it is an alerter and then also because it is an interval callback.

```
loop as long as any object has the Change bit set {
  sp X = an object with the Change bit set;
  sp * (A1 A2 ... An) = all alerters relevant to X;
  void * Old = X.Old (the old values associated with X);
  ProcessAlerters(X, Old, (C1 ... Cn));
  spSetIsNew(X, FALSE);
  X.Old = X.Current;
}


boolean RelevantToObject(sp A, sp X) {
  return CompatibleWithMask(X, spAlerterGetMask(A)) &&
         (X == spGetParent(A) ||
          (spGetParent(A) != NULL &&
           spClassEq(spGetClass(spGetParent(A)), spClassC()) &&
           spClassLeq(spGetClass(X), spGetParent(A))));
}
```

```
ProcessAlerters(X, Old, (A1 ... An)) {
  Current = X.Current;
  spSetChanged(X, FALSE);
  For i = 1 to n {
    if (!spGetIsRemoved(Ai)) {
      spFn P = spAlerterGetP(Ai);
      spFn F = spAlerterGetF(Ai);
      X.Old = Old
      spWMSetMe(spGetOwner(Ai));
      if (P(X, spAlerterGetPState(Ai))) {
         if (F(X, spAlerterGetFState(Ai))) spRemove(Ai);
      }
      spWMSetMe(spWMGetSystemOwner());
      if (spGetChanged(X)) ProcessAlerters(X, Current, (A1 ... Ai));
      if (spGetIsRemoved(X) && X == spGetParent(Ai)) spRemove(Ai);
    }
  }
}
```

In addition to specifying when alerters are triggered, the pseudo-code specifies exactly when the Change and IsNew bits are reset and exactly when the old values of the shared variables are set. These things only have clear values during the evaluation of alerter predicates and functions. In ordinary application code, they are of little or not benefit.

A central reason why the code above has the structure shown is to insure that the old values have an easily understood meaning. Consider a particular object X and a particular alerter A. When the predicate P or function F of A are applied to X, the Change bit is False (but must have just been true) and the old values associated with X record what the shared values were the last time P was applied to X. If A was created since the last time alerters had an opportunity to be applied to X, then the old values record the final shared values after the last time alerters had an opportunity to be applied to X. If X was created since the last time alerters had an opportunity to be applied to objects like X, then the old values record the initial shared values associated with X and the IsNew bit is True.

Looked at naively, the above suggests that each object has to have separate old value storage for each alerter relevant to it. The trick of the above code is that it arranges to have one set of stored values for each object suffice by saving these values only when a single set of values is correct for every alerter relevant to the object.

The main loop just keeps running alerters on objects until every object has the Change bit off. This allows all changes to propagate from one alerter to another completely in a single update time. It is advisable that there be no such propagation. If there isn't, then each changed object only has to be processed once. If there is propagation, then the programmer must take care to ensure that the propagation will terminate. Fortunately, since every alerter must be locally created, all alerter interaction is directly under the control of the programmer.

As noted above, alerters are actually called on external and local objects separately. This is achieved by restricting the main loop above to one group of objects at a time. When the main loop terminates, the old and current shared values are equal for every object considered.

The subroutine RelevantToObject specifies which alerters are relevant to which objects. The test CompatibleWithMask is presented in the discussion of spMask values (Section 7). In order to reduce processing time, the actual implementation does not call a subroutine like RelevantToObject , but rather uses highly efficient indexing based on the Parents of alerters.

Given an object X, a set of old values, and a set of alerters A1...An, ProcessAlerters runs these alerters until a situation is reached where spGetChanged(X) is False and every alerter has been given a chance to run on the final state of the object. Note that each time an alerter is tested for running, the old values are the shared values that existed the last time that particular alerter was tested. Critically, the old values are never later values nor earlier values. As a result, alerters don't fail to fire and don't fire twice when they should only fire once. (Handling the old values right requires a stack of saved old value sets associated with the recursive invocations of ProcessAlerters.)

Note that if the alerters A1...An refrain from modifying shared values of X, then each alerter will only be tested once and (at most) run once. (Note also that if none of the Ai have the same owner as X, then no modification of X is possible and copying to temporarily save old values is not necessary.)

As is the case in the main loop, the termination of ProcessAlerters is not assured. The programmer must take care. In general, it is best for alerters to refrain from making any modification that could trigger another alerter. If there are no such modifications then each alerter is only processed once and only one copy of current to old values is required.

43.2 P

```
transient public spFn P; //* [spFn] readonly

      spFn spAlerterGetP(sp Object)

      void spAlerteriSetP(sp Object, spFn X)

      spFn sqAlerterGetP(sp Object)
      void sqAlerterSetP(sp Object, spFn X)
```

spAlerter objects have a local instance variable called *P* that holds the locally defined predicate (Section 6.1) to be called when testing whether the spAlerter should be applied. It must be set when an spAlerter is created and cannot be changed later. Information about the P is maintained separately in each process.

Alerter event processing proceeds as follows. Each time spWMUpdate is called, each alerter is considered for possible event-driven triggering. For a given alerter A, the system first determines the set of objects compatible with the Parent of A. If the Parent is an spClass object, this is every object that is in that class or any of its subclasses. If the Parent is an ordinary object then this is the only object that needs to be considered. If the Parent is Null then there are no compatible objects and no events ever get triggered.

The set of compatible objects is then refined as follows. Every object that does not have some shared variable that has changed since the last call on spWMUpdate (i.e., every object that does not have the Change bit set to True) is discarded. Then, every object that is not compatible with the alerters mask is discarded. The predicate P is then applied to each of the remaining objects, and every object for which P returns False is also discarded. The function F is then applied to every object that remains. (For example, the predicate P might specify that F be applied only when the position of the object under consideration has changed.)

The predicate P for an alerter should be purely a function, not having any side-effects anywhere. In addition, it is very important that P run quickly—i.e., in a few microseconds at most—because it is being called from inside spWMUpdate and not in a separate thread.

43.3 PState

```
transient public int PState; //* [void *] readonly

      void * spAlerterGetPState(sp Object)

      void spAlerteriSetPState(sp Object, void * X)

      void * sqAlerterGetPState(sp Object)
      void sqAlerterSetPState(sp Object, void * X)
```

spAlerter objects have a local instance variable called *PState* that holds state information that is passed to the P operation when it is called. This must be set when an spAlerter is created and cannot be changed later. Information about the PState is maintained separately in each process.

43.4 Mask (Fundamental)

```
public int Mask; //* [spMask]

      spMask spAlerterGetMask(sp Object)
      spMask spAlerterGetOldMask(sp Object)
      void spAlerterSetMask(sp Object, spMask X)

      spMask sqAlerterGetMask(sp Object)
      spMask sqAlerterGetOldMask(sp Object)
      void sqAlerterSetMask(sp Object, spMask X)
```

spAlerters have a shared instance variable called the *Mask* that controls which objects the alerter can be applied to. The Mask is a world model visibility mask (Section 7). Events that trigger the alerter must involve an object that matches the Mask of the alerter.

The default value of the Mask is spMaskNORMAL, which is appropriate for most situations. It can be changed to any other spMask value. Information about the Mask is shared between processes.

43.5 ChgNext (Fundamental and Internal)

```
transient public int ChgNext; //* [void *] internal

      void * spAlerteriGetChgNext(sp Object)
      void spAlerteriSetChgNext(sp Object, void * X)

      void * sqAlerterGetChgNext(sp Object)
      void sqAlerterSetChgNext(sp Object, void * X)
```

For efficiency, an index is created of spAlerters. This index makes if very fast to determine which spAlerters could possibly be triggered due to a change in a given object. The *ChgNext* local instance variable of an spAlerter is used as a forward pointer when constructing doubly linked lists of alerters in this index.

The ChgNext variable is maintained by the system and cannot be manipulated by an application. Information about the ChgNext is maintained separately in each process.

43.6 ChgPrev (Fundamental and Internal)

```
transient public int ChgPrev; //* [void *] internal

      void * spAlerteriGetChgPrev(sp Object)
      void spAlerteriSetChgPrev(sp Object, void * X)

      void * sqAlerterGetChgPrev(sp Object)
      void sqAlerterSetChgPrev(sp Object, void * X)
```

The *ChgPrev* local instance variable of an spAlerter is used as a back pointer in the doubly linked lists of alerters that can be triggered by changes in a given object. The ChgPrev variable is maintained by the system and cannot be manipulated by an application. Information about the ChgPrev is maintained separately in each process.

43.7 spAlerterNew (Fundamental)

```
sp spAlerterNew(sp X, spDuration I, spFn P, void * PState, spFn F, void * FState)
```

X - Desired parent.
I - The desired interval.
P - Desired test (Section 6.1).
PState - Desired test state.
F - Desired operation (Section 6).
FState - Desired operation state.
Return value - The newly constructed object.

Creates as spAlerter with the specified Parent X, Interval I, P, PState, F, and FState. None of the above can subsequently be changed. The Mask is set to spMaskNORMAL. You can change it to a different value. The alerter can be terminated by removing the alerter object.

43.8 spAlerterInitialization (Internal)

```
void spAlerterInitialization(sp Action)
```

Action - spAction to initialize.
Return value - There is no return value.

Initializes the Mask in an alerter to spMaskNORMAL.

44 spBeaconMonitor (Fundamental)

```
public class spBeaconMonitor extends spAlerter
```

Applies an operation to every beacon (Section 20.1) whose Tag matches an indicated Pattern. This is done by first applying the specified operation to every matching beacon currently in the local world model copy. The spBeaconMonitor alerter then applies the operation in the future to every matching beacon that appears, changes, or is removed. In addition, a request is sent to the appropriate Content-Based Communication server. This request is initially answered by the server sending descriptions of every currently existing beacon matching the Pattern so that they can be entered in the local world model copy. Modifications to these beacons (including removal) and information about newly created beacons is communicated at later times by the server as long as the spBeaconMonitor alerter remains in effect. This process terminates only when the spBeaconMonitor object is removed.

The shared class spBeaconMonitor inherits all the instance variables and functions of the classes: spAlerter (Section 43), spIntervalCallback (Section 42) and sp (Section 15). The class spBeaconMonitor defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spBeaconMonitorC - Class descriptor (Section 15.1).
shared **Pattern** - Query pattern (Section 44.1).

The class spBeaconMonitor defines the following functions:

spBeaconMonitorNew - Creates beacon monitor (Section 44.2).

The Pattern variable of spBeaconMonitor object specifies which beacons are to be monitored. The Pattern of an spBeaconMonitor is a URL and is exactly the same as the Pattern of an spBeaconing, except that it can contain instances of the wild card character asterisk (*). There must be a DNS/port part of the Pattern present and this part of the Pattern cannot contain a wild card character. The rest of the Pattern can contain one or more wild card characters. A beacon is deemed to match an spBeaconMonitor if (and only if) the Tag of the beacon matches the pattern.

Wild card matching operates in the standard way. For example, you might have beacons with the Tags `"//myhost/bicycle/EBF"`, `"//myhost/bicycle/RCW"`, and `"//myhost/unicycle/RCW"`. In that case, an spBeaconMonitor with Pattern `"//myhost/bicycle/"` matches the first two tags while `"//myhost/*RCW"` matches the last two Tags. (When matching a Tag and a Pattern, things are simple as long as the Pattern contains only one wild card. However, if there are multiple wild cards, then things become more complex and greater computation is required to decided which Tags match which Patterns.)

44.1 Pattern

```
public String Pattern; //* [spFixedAscii]

      spFixedAscii spBeaconMonitorGetPattern(sp Object)
      void spBeaconMonitorSetPattern(sp Object, spFixedAscii X)

      spFixedAscii sqBeaconMonitorGetPattern(sp Object)
      void sqBeaconMonitorSetPattern(sp Object, spFixedAscii X)
```

The key feature of beacons is that a global name service is maintained that maps from Tags to the beacon objects so that beacons can be accessed based on their Tags no matter where they are located. To allow many different people to create Tags that do not conflict with each other, beacon Tags are URLs. It is generally not a good idea to have lots of beacons with the same Tag, but there can be several beacons with the same Tag.

The central instance variable of an spBeaconMonitor is a shared variable called the *Pattern*. This is identical to a beacon Tag, except that it can contain asterisks as wild card characters and acts as a query instead of a label. The DNS/port part of a Pattern URL (which must be present and must not contain a wild card character) selects the process that provides beacon service for the beacon.

An spBeaconMonitor Pattern is a string of type spFixedAscii. It must be less than 500 characters in length, so that the containing object can fit into a single UDP message. It must be specified when a beacon monitor is initially created and cannot be changed after that time. Information about the Pattern is shared between processes.

44.2 spBeaconMonitorNew

```
sp spBeaconMonitorNew(spFixedAscii Pattern, spFn F, void * S, spDuration I)
```

Pattern - Pattern for Tag of beacons to be monitored.
F - Operation (Section 6) to be applied to every beacon with tag.
S - Passed to the operation each time it is called (modifiable).
I - Time after which the monitoring should stop.
Return value - The newly constructed object.

Creates a beacon monitor object with the indicated Pattern, F, Fstate (S), and Interval (I). It is an error for the DNS name part of an spBeaconMonitor Pattern to contain a `*`. If the operation F ever returns True, the spBeaconMonitor is removed and monitoring ceases. If a non-negative Interval is specified, then monitoring will cease after that time. You can terminate monitoring at any time by removing the spBeaconMonitor object. In the spBeaconMonitor created, the Parent is the class spBeaconing.

An spBeaconMonitor alerter uses a predicate P that tests whether an spBeaconing object has a matching Tag and is not an spBeaconMonitor itself. If the Interval expires, F is called with spBeaconingC as its argument.

spBeaconMonitor Patterns must include a DNS name specification. To ensure this, the following default is applied to the Pattern string provided. If the Pattern begins with just one `/` or a colon `:` then it is prefixed with `//<HostName>`. Where `<HostName>` is the DNS name of the machine the process is running on. If the Pattern does not begin with `/` or colon then it is prefixed with `//<HostName>/`. That is to say:

```
//string => //string
/string => //<HostName>/string
:string => //<HostName>:string
string => //<HostName>/string
=> //<HostName>/
```

Note that the defaulting to `<HostName>` is only useful for spBeaconMonitors that are used to communicate between (or within) processes on a single machine. If an spBeaconMonitor is to be used to communicate between machines, an explicit host name must be specified.

Monitoring happens in three stages. First, at the moment the spBeaconMonitor object is created, an spClassExamine is done to locate any beacons in the local world model copy with matching Tags. If F returns True during this process, then examining is terminated and the spBeaconMonitor object created is removed before it is returned.

Second, if the examination of the objects currently in the world model does not cause F to return True, then F is used as the F value of the spBeaconMonitor alerter created. The appropriate Content-Based Communication server is contacted to obtain information about all beacons that currently exist that match the Pattern in the global world model. Monitoring continues on the beacons that come from the Content-Based Communication server.

Since the server has no knowledge about what beacons the local process does and does not have in its world model, it sends complete information about all beacons that match the Pattern to the local process. This includes duplicate information about the beacons currently in the world model. However, this duplicate information is filtered out because the messages do not have Counter values larger than those used on the objects in the world model already. Therefore, the spBeaconMonitor alerter will not be triggered a second time on the objects it was initially triggered on in the first phase above. (The key reason that the first phase is needed, is that even though this duplicate data arrives, it does not cause world model changes that can trigger the spBeaconMonitor alerter.)

Third, as long as the spBeaconMonitor continues to exist, the Content-Based Communication server will inform the local process about relevant beacon information. The interval value in an spBeaconMonitor can be used to remove the spBeaconMonitor after a specified amount of time.

45 spBeaconGoto

```
public class spBeaconGoto extends spBeaconMonitor
```

Places an spPositioning object either under, or beside a beacon with a particular Tag. It does this by finding a beacon with the tag and then appropriately positioning the object. It is expected that, the object must be an spObserving or have an spObserving as one of its descendants. The spObserving descendant is necessary in order to trigger the receipt of information about the Parent of the beacon.

As with spBeaconMonitor, if the beacon exists, it is located no matter what locale it is in. The Object is made a brother of the beacon with the same Transform as the beacon. This is the proper way to teleport an avatar to a place specified by a beacon. It is necessary if you wish the avatar to move around the locale and into neighboring locales under its own control.

An operation is called after the Object has been properly positioned or an interval has timed out. In general, spBeaconGoto is best used when there is only one beacon in existence with the tag. If there are more, then the first one encountered will be used.

The shared class spBeaconGoto inherits all the instance variables and functions of the classes: spBeaconMonitor (Section 44), spAlerter (Section 43), spIntervalCallback (Section 42) and sp (Section 15). The class spBeaconGoto defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spBeaconGotoC - Class descriptor (Section 15.1).
Object - The object to be placed beside the beacon found (Section 45.1).

The class spBeaconGoto defines the following functions:

spBeaconGotoNew - Creates spBeaconGoto given a pattern (Section 45.2).

spBeaconGoto operates in essentially the same way as spBeaconMonitor. However, once having found a beacon, it copies the Parent, Locale, and Transform of the beacon into the Object. Because it uses internal operations, it could do this without needing to wait until the Parent object appears. However, it waits until the Parent object appears before calling the specified operation so that the application can inspect the Parent if it wants to.

45.1 Object

```
transient public spObserving Object; //* [sp] readonly

      sp spBeaconGotoGetObject(sp Object)

      void spBeaconGotoiSetObject(sp Object, sp X)

      sp sqBeaconGotoGetObject(sp Object)
      void sqBeaconGotoSetObject(sp Object, sp X)
```

The *Object* local instance variable of an spBeaconGoto object records the spPositioning object that is to become a brother of the beacon to be located.

45.2 spBeaconGotoNew

```
sp spBeaconGotoNew(spFixedAscii P, sp Object, spFn F, void * S, spDuration I)
```

P - Pattern of beacon to be found.
Object - Object to be positioned beside the beacon.
F - Operation (Section 6) to be applied after spObserving positioned.
S - Passed to the operation each time it is called (modifiable).
I - Time after which searching should stop.
Return value - The newly constructed object.

Creates an spBeaconGoto object with the indicated Object, Pattern (P), and Interval (I). The Pattern of an spBeaconGoto defaults in exactly the same way as any other spBeaconMonitor (Section 44.2).

The specified F and FState (S) do not directly become the F and FState values of the spBeaconGoto alerter created. Rather, a special operation is used that calls the user specified F once the object has been properly placed beside a beacon. A special predicate operation is used that locates a beacon matching the specified Pattern.

The special predicate and function position the object in two steps. The predicate first looks for a beacon matching the Pattern. When the beacon is found, the Parent, Locale, and Transforms are copied from the beacon to the Object. (This can be done even when the Parent is not in the locale world model via the use of internal operations.) In order to avoid ugly transitions, the Object is not modified until it can all Parent, Locale and Transform can all be given correct final values. (The beacon is saved in the PState for future reference, and to indicate that the second step of processes should commence.)

If the Object is (or has a descendant that is) an spObserving, this triggers the subsequent acquisition of information about the Parent of the beacon. The spBeaconGoto alerter continues in operation until the predicate observes that the Parent of the beacon has appeared. Once this happens, F is called with the beacon as its argument. If the Interval expires before all this can be accomplished, F is called with spBeaconingC as its argument.

46 spAction (Fundamental)

```
public abstract class spAction extends sp
```

Action objects contain functions that are executed during calls on spWMUpdate. spAction objects are communicated to other processes using Locale-Based Communication just like any other object. The key use of actions is to reduce communication costs by making it possible to infrequently send small programs that make it unnecessary to frequently send data updates. For example, this is used to support the smooth motion (Section 16.13) of objects.

The class spAction is in the external API so that application writers can create new subclasses of spAction that support new kinds of activities. In order to lay the groundwork for devising new actions, the following subsection (Section 46.1) describes how actions operate in detail. There are several predefined subclasses of spActions, but only one is part of the external API.

The shared class spAction inherits all the instance variables and functions of the class sp (Section 15). The class spAction defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spActionC - Class descriptor (Section 15.1).

The class spAction defines the following functions:

spActionFunction - Performs remote action (Section 46.2).

It is not possible to create instances of the class spAction. Rather, one can only create instances of particular subclasses of the class spAction.

Every time spWMUpdate is called, every action in the world model is run. This is done by applying the Function method to the action object and its Parent.

As in any object, the Parent variable of an action also controls what locale the action is in and therefore how it will be communicated to other processes. In particular, if the Parent is Null then an action will not be communicated to any other processes. In addition, if the Parent is an spClass object, then the action will not be communicated anywhere except in the unlikely event that the spClass has a Parent that puts it in a locale.

If the object that is the Parent of an action is removed, then the action itself will also be removed the next time spWMUpdate is called.

46.1 Details of Action Processing

Actions are only called during calls on spWMUpdate. Specifically, in spWMUpdate, just after all messages describing changes in the world model have been processed, the world model is scanned looking for actions that should be applied.

The following pseudo-code shows the logic of the operation of actions during spWMUpdate processing.

```
for each action A in the local world model copy {
  method Function = Function method corresponding to A;
  spWMSetMe(spGetOwner(A));
  inhibitMessagingExcept(A));
  if (spGetParent(A) not present in world model) do nothing;
  elseif (spGetIsRemoved(spGetParent(A))) remove A from the locale world model copy;
  elseif (Function(A, spGetParent(A))) remove A from the locale world model copy;
  allowNormalMessaging();
  spWMSetMe(spWMGetSystemOwner());
}
```

There is no guarantee of the order in which actions will be considered. An index of actions is used to make retrieving them faster.

The value of spWMGetMe is altered during the application of an action so that the Function is prevented from changing objects owned by other processes. The operation inhibitMessagingExcept sets flags so that altering fields of any object other than A itself will not cause the MessageNeeded bit to be set.

An action Function is always applied to the Parent of the action.

When an object X is removed, any actions that have X as their Parents are also removed.

It is possible for an action created by another process that has a Parent to be communicated to the local world model before its Parent is. In this situation, the Parent will appear to an application program to be Null. The action Function is not applied until after the Parent appears. (Note that while an application cannot tell the difference between the Parent having not arrived yet and having been removed, the system core can discriminate between these two cases.)

46.2 spActionFunction

```
spBoolean spActionFunction(sp Action, sp Parent)
```

Action - Action being applied.
Parent - Parent of the action.
Return value - True causes action to be removed from the local world model.

Each action class is associated with a *Function* that performs the action in question. In particular, the Function is called whenever the action is triggered. The Function does not exist in the class spAction itself. It is described here to show how the Function methods for subclasses of spAction must be defined. In particular, they must take the arguments shown and return a boolean value.

On each call to spWMUpdate, every action is run once. When an action is run, the Function is called on its Parent. The purpose of the Function is to perform some operation, typically modifying the Parent. (For example, an action supporting smooth motion might interpolate between positions at particular times to determine intermediate positions.) Subclasses of spAction typically contain additional instance variables for the purpose of representing state information.

The Function may well call a Set accessor on the target object (or some other). Two things are important to note. First, if an object is modified, it must have the same owner as the action. This is the only situation where a program running in a process can modify an object that is not owned by that process. The action can be looked at as a proxy of the actual owner of the object, acting on the owner's behalf.

Second, Set accessors typically have two special side effects. They tell the system core that an object has been changed so that appropriate alerters will be run on the object and they tell the system core that a message should be sent out describing the change in the object. When used inside the Function of an spAction, Set accessors have the former effect, but they do not have the latter effect except when the action itself is being modified. To understand this, consider the following. Suppose that an spAction is controlling the position of its target object. A key virtue of actions in that situation is that communicating one action can compactly specify a long sequence of positions. This benefit would be canceled if an additional message was sent whenever the action changed the position of the object. However, it is occasionally beneficial for an spAction to modify itself causing the updated action to be communicated. (Note that Set accessors operate normally in interval callbacks and alerters.)

As a general matter, action Functions should never delete or create objects. (Note that if an object were created it would have a name from the name space of one owner, but an the owner id from another owner.)

The return value of Function specifies what should happen to the action after it is applied to an object. If Function returns True, then the action is removed, but only from the local world model copy. In particular, even if the action is being applied in the process that owns it, the removal occurs only in the local world model copy with no effect on the world model in any other process. This is important so that the copies of the action in other processes can run to normal completion without having their existence terminated prematurely.

It is very important that Function run quickly—i.e., in much less than 1 Millisecond—because it is being called from inside spWMUpdate and not in a separate thread. If you have a computation intensive thing to do, the application should perform the within itself, exporting the results, rather than exporting the computation to other processes.

47 spOwnershipRequest

```
public class spOwnershipRequest implements spAction
```

Each shared object has only one owner and only the owner can alter any of the shared instance variables of an object. This is an important restriction that is essential for achieving consistent real-time interaction. However, it can be constricting when groups of users wish to exercise joint control over an object. To facilitate joint control of objects, the ownership of an object can be transferred easily from one process to another by merely setting the Owner variable. However, this can only be done by the present owner.

To further facilitate the transfer of ownership, a handshaking protocol centered around a special shared object called an spOwnershipRequest is provided to make it easy for a process to ask for and obtain ownership of an object. The basic protocol (Section 47.1) is that someone who wishes to get ownership of an object creates an spOwnershipRequest whose Parent is the object. The current owner of the object can choose to grant the request, in which case ownership is transferred.

The shared class spOwnershipRequest inherits all the instance variables and functions of the class spAction (Section 46). The class spOwnershipRequest defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spOwnershipRequestC - Class descriptor (Section 15.1).
F - Operation to apply (Section 47.2).
FState - State used with operation (Section 47.3).
shared **Timeout** - Timeout interval (Section 47.4).
TimeAlive - Time request has been in existance (Section 47.5).

The class spOwnershipRequest defines the following functions:

spOwnershipRequestNew - Creates ownership request (Section 47.6).
spOwnershipRequestFunction - Applies F to newly owned object (Section 47.7).
spOwnershipRequestGrant - Satisfies ownership request (Section 47.8).

The object that is the target of the ownership request is indicated by making it be the Parent of the ownership request object. The Parent is set when the ownership request is first made and should not be changed thereafter.

spOwnershipRequest objects are communicated between processes, but the specified operation is only called in the creating process.

47.1 Ownership Transfer

The owner of an object can force the object on another process. However, the transfer of ownership of an object typically involves two processes: a requesting process that decides it wants to obtain ownership and the current owner who is the only one that can change who the owner is. Moreover, once the owner has been changed, the old owner can no longer do anything with the object. Therefore, the new owner must realize that it has become the owner and take over.

In recognition of the above, the ownership transfer protocol operates in three basic steps.

(1) A process that wants to gain ownership creates a request. The request must be left in existence for a substantial time (seconds) because otherwise, the request might come and go in the world model so quickly that a slowly reacting owner might never notice its existence.

(2) A process that owns an object it might be willing to give up monitors the world model looking for the appearance of appropriate ownership requests. To grant a request, the owner calls spOwnershipRequestGrant, which effects the change. After that time, the old owner need take no further action.

Being on the lookout for appropriate ownership requests is best done with an alerter. In particular, the alerter predicate spRelevantOwnershipRequest (Section 6.1) exists to make this easy. Using this alerter predicate causes the alerter operation to be applied whenever a new alerter request appears that is asking for an object owned by the local process. For example you might write:

```
spAlerterNew(spOwnershipRequestC(),-1,spRelevantOwnershipRequest, NULL, F, NULL);
```

If it is possible that there are preexisting spOwnershipRequests that are relevant, then you should do an spClassExamine to find them before setting up the alerter above.

(3) The process that wants to gain ownership must monitor the world model to determine whether the request is granted. This is done by the spOwnershipRequest action itself.

(It is worthy of note that joint control over an object can also be obtained by having a central neutral broker that controls the object in response to requests from other processes. This might be a convenient way to handle the ball in a virtual soccer game, especially in situations where the ball is far from any player. However, it has the problem that it increases the latency of interaction with the object in comparison to having the processes that is effecting the object most at any given moment be the owner of the object and effect it directly.)

Since a reliable communication protocol is used, the ownership transfer protocol can rely on the fact that the original owner of an object will be informed of any ownership requests on the object and that if a request is granted, the requester (and every other process) will be informed of the change. This simplifies the implementation of the ownership transfer protocol.

However, the underlying communication protocol puts no limits on the length of time that a message can take in transit. As a result, there is a race condition in the ownership transfer protocol.

A process may observe and grant an ownership request, but due to delays, the notification of the change may not reach the requester until a time when the request is no longer in effect. This could happen either because the request timed out shortly after (or before) it was granted or because the request was removed by the requester shortly after (or before) it was granted.

In either case, the situation could arise that the requester was no longer expecting to get ownership but did anyway. Which in turn could lead to the situation that no application thought that it was responsible for the object. (Due to the reliability of the underlying communication, the system in the requesting process will realize that it had ownership of the object, but the application might not.)

To ameliorate this race condition, the ownership transfer protocol operates as follows. Care is taken to make sure that the spOwnershipRequest remains in existence in the creating process substantially longer than in other processes. This is done by having the spOwnershipRequest be in effect for a second longer in the process that creates it than in remote processes. (To allow for this the timeout interval is required to be at least 2 seconds.) This makes it unlikely in practice that the request can be granted without the spOwnershipRequest alerter being triggered. However, the probability is not zero.

As a result, applications that wish to share an object between processes should be written so that they can deal with unexpected ownership changes. In particular, they should be written so that each participating process always considers ownership requests for objects they own even if they do not realize that they own the objects in question. That is to say, they should have a blanket alerter on spOwnershipRequests always in force, rather than only putting alerters on spOwnershipRequests for particular objects they think they are responsible for.

Note that it is impossible for the situation to arise where two processes both think they own a given object. The reason for this is that no process can think that ownership has changed until after the original owner no longer thinks that it owns the object. (It can happen that two processes disagree on who the owner of an object is, because one hears of a change before the other; however, at most one of the processes can think that it is the owner.)

Actually changing the ownership of an object is trivial. The owner merely changes the Owner variable and sends a message describing the change to other processes. (This is the only situation where a process sends a message about an object it does not currently own.)

The above discusses how ownership transfer is achieved. It must also be considered, when ownership transfer is sensible. To start with transferring ownership of an object A from X to Y only makes sense when X and Y both know about A. In particular, it does not make sense for alerters because only the original creator even knows a given alerter exists. In addition, if X and Y both know about A, they do not necessarily know exactly the same things about A. In particular, while X and Y agree on the shared variables of A, they do not agree on the local variables of A. Fortunately, this often does not matter. For example, spThings have a GraphicsNode variable that will have different pointer values on different machines, but nevertheless have equivalent values on different machines. In particular, what the value is has no need to change when ownership changes. There are however, situations where this is not true and therefore ownership change is complex.

A good example of the problems that can be involved with ownership transfer is spMover objects. These objects have a local variable Queue, which holds information about future motions. However, this information is represented only on the owning machine. On other machines the Queue is always empty and the only information is about the present (and immediate future) motion of the object. If an spMover object has its owner changed, the information about far future motions will be lost. Any change of ownership of spMover objects must take this into account. In general, it is better to change the ownership of a thing and then make a new spMover object, rather than attempt to change ownership of the spMover object.

47.2 F

```
transient public spFn F; //* [spFn] readonly

      spFn spOwnershipRequestGetF(sp Object)

      void spOwnershipRequestiSetF(sp Object, spFn X)

      spFn sqOwnershipRequestGetF(sp Object)
      void sqOwnershipRequestSetF(sp Object, spFn X)
```

SpOwnershipRequest objects have a local instance variable called *F* that holds the locally defined operation (Section 6) to be called when the ownership request succeeds or times out. It must be set when an SpOwnershipRequest is created and cannot be changed later. Information about the F is maintained separately in each process.

47.3 FState

```
transient public int FState; //* [void *] readonly

      void * spOwnershipRequestGetFState(sp Object)

      void spOwnershipRequestiSetFState(sp Object, void * X)

      void * sqOwnershipRequestGetFState(sp Object)
      void sqOwnershipRequestSetFState(sp Object, void * X)
```

spOwnershipRequest objects have a local instance variable called *FState* that holds state information that is passed to the F operation when it is called. This must be set when an spOwnershipRequest is created and cannot be changed later. Information about the FState is maintained separately in each process.

47.4 Timeout

```
public int Timeout; //* [spDuration]

      spDuration spOwnershipRequestGetTimeout(sp Object)
      spDuration spOwnershipRequestGetOldTimeout(sp Object)
      void spOwnershipRequestSetTimeout(sp Object, spDuration X)

      spDuration sqOwnershipRequestGetTimeout(sp Object)
      spDuration sqOwnershipRequestGetOldTimeout(sp Object)
      void sqOwnershipRequestSetTimeout(sp Object, spDuration X)
```

The *Timeout* shared instance variable of an spOwnershipRequest specifies how long the request should be in effect before it automatically times out. A negative value indicates that the request will never time out.

The Timeout value is a time in milliseconds. It should be set when an spOwnershipRequest is initially created and should not be changed later. Information about the Timeout is shared between processes.

47.5 TimeAlive (Internal)

```
transient public int TimeAlive; //* [spDuration]

      spDuration spOwnershipRequestGetTimeAlive(sp Object)
      void spOwnershipRequestSetTimeAlive(sp Object, spDuration X)

      spDuration sqOwnershipRequestGetTimeAlive(sp Object)
      void sqOwnershipRequestSetTimeAlive(sp Object, spDuration X)
```

The *TimeAlive* local instance variable of an spOwnershipRequest records how long the request has been in existance. This is used to control when the request terminates itself. If the Timeout is positive then: on a remote machine the request terminates when TimeAlive becomes greater than 1 second less than the Timeout; on the local machine the request times out when the TimeAlive becomes greater than the Timeout.

The TimeAlive value is a time in milliseconds. It is maintained by the action Function and should not be modified by an application. Information about the Timeout is maintained separately in each process.

47.6 spOwnershipRequestNew

```
sp spOwnershipRequestNew(sp Object, spFn F, void * FState, spDuration Timeout)
```

Object - Object you want to own.
F - Op (Section 6) applied when request is satisfied or times out.
FState - Passed to the operation each time it is called (modifiable).
Timeout - Interval in milliseconds to wait for ownership to be granted.
Return value - The newly constructed object.

Creates an spOwnershipRequest object seeking ownership of an object. It is up to the current owner of the object to decide whether to grant a given ownership request.

The spOwnershipRequest action created applies F to the target object either when the timeout interval is reached or when the owner of the object changes to spWMGetMe. It is up to F to detect which has occurred (by looking at the owner variable of Object) and decide what to do next.

In either case, the spOwnershipRequest removes itself as soon as it is triggered. A negative Timeout causes the request to never Timeout. It the Timeout is not negative, it should be at least several seconds. The request is removes itself on remote machines 1 second before the timeout interval is reached. On the local machine, the request times out, calling F when the Timeout interval is reached.

In the spOwnershipRequest created, the Parent is the target object. Note that ownership request objects are communicated between processes. However, the operation F is only called in the process that created the spOwnershipRequest.

(It is possible to cancel an ownership request by removing the spOwnershipRequest object. However, this is unwise because you might then fail to notice that another process granted the request just before you removed it.)

If a process creates an ownership request whose target is an object it already owns, then everything proceeds in the same way as above. How long it takes before F is applied to the object depends on whether the process grants ownership of the object to itself or not.

47.7 spOwnershipRequestFunction (Internal)

```
spBoolean spOwnershipRequestFunction(sp Action, sp Object)
```

Action - Action being applied.
Object - Object whose ownership you want.
Return value - Always returns True.

If running in a process other than the one that owns the spOwnershipRequest action, the Function merely monitors the passage of time and returns True, terminating the action, one second before the Timeout interval expires.

If running in the process that owns the spOwnershipRequest action, the Function monitors the passage of time and whether the ownership of the target object. If the owner of the target object changes to be the same as the owner of the request or the Timeout interval expires, then F is applied to the target object and True is returned, terminating the request.

47.8 spOwnershipRequestGrant

```
void spOwnershipRequestGrant(sp Request)
```

Request - The ownership request to be granted.
Return value - There is no return value.

If the process owning an object decides to grant an ownership request, it does so by calling spOwnershipRequestGrant on the spOwnershipRequest object in question. If the process does not wish to grant the request, then it need not take any action.

spOwnershipRequestGrant operates by merely changing the Owner of the object to be the same as the owner of the ownership request.

48 spDoSoundPlay (Internal)

```
public class spDoSoundPlay implements spAction
```

This specialization of spAction supports the remote playback of a recorded sound. It does this by simulating the receipt of sound as if it had been played through the spAudioSource that is the Parent of the spDoSoundPlay object. spDoSoundPlay actions are created by the function spSoundPlay.

The shared class spDoSoundPlay inherits all the instance variables and functions of the class spAction (Section 46). The class spDoSoundPlay defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spDoSoundPlayC - Class descriptor (Section 15.1).
shared Sound - The sound to be played (Section 48.1).
shared Loop - Causes the sound to loop (Section 48.2).
shared Gain - Gain applied to sound (Section 48.3).

The class spDoSoundPlay defines the following functions:

spDoSoundPlayFunction - Plays recorded sound effect (Section 48.4).
spDoSoundPlayNew - Create new spDoSoundPlay object. (Section 15.28).

48.1 Sound (Internal)

```
public spSound Sound; //* [sp] internal

      sp spDoSoundPlayiGetSound(sp Object)
      sp spDoSoundPlayiGetOldSound(sp Object)
      void spDoSoundPlayiSetSound(sp Object, sp X)

      sp sqDoSoundPlayGetSound(sp Object)
      sp sqDoSoundPlayGetOldSound(sp Object)
      void sqDoSoundPlaySetSound(sp Object, sp X)
```

The *Sound* shared instance variable of an spDoSoundPlay is the spSound to be played. It is set when the spDoSoundPlay is created and cannot be changed later. Information about the Sound is shared between processes.

48.2 Loop (Internal)

```
public boolean Loop; //* [spBoolean] internal

      spBoolean spDoSoundPlayiGetLoop(sp Object)
      spBoolean spDoSoundPlayiGetOldLoop(sp Object)
      void spDoSoundPlayiSetLoop(sp Object, spBoolean X)

      spBoolean sqDoSoundPlayGetLoop(sp Object)
      spBoolean sqDoSoundPlayGetOldLoop(sp Object)
      void sqDoSoundPlaySetLoop(sp Object, spBoolean X)
```

The *Loop* bit shared instance variable of an spDoSoundPlay specifies whether the sound should be played just once, or continuously. It is set when the spDoSoundPlay is created and cannot be changed later. Information about the Loop bit is shared between processes.

48.3 Gain (Internal)

```
public float Gain; //* [float] internal

      float spDoSoundPlayiGetGain(sp Object)
      float spDoSoundPlayiGetOldGain(sp Object)
      void spDoSoundPlayiSetGain(sp Object, float X)

      float sqDoSoundPlayGetGain(sp Object)
      float sqDoSoundPlayGetOldGain(sp Object)
      void sqDoSoundPlaySetGain(sp Object, float X)
```

The *Gain* shared instance variable of an spDoSoundPlay is the gain to be used when playing back the sound. A value of 1.0 means no gain. The Gain value is set when the spDoSoundPlay is created and cannot be changed later. Information about the Gain is shared between processes.

48.4 spDoSoundPlayFunction (Fundamental and Internal)

```
spBoolean spDoSoundPlayFunction(sp Action, sp Object)
```

Action - Action being applied.
Object - Source to play sound through.
Return value - Returns True when sound finished playing.

Plays a recorded sound as coming from an spAudioSource.

49 spMover (Internal)

```
public class spMover implements spAction
```

This specialization of spAction supports the smooth motion of objects. It does this by interpolating between spTransform values using Catmul Rom interpolation. The Parent of the spMover is the object being moved. spMover actions are created by the smooth motion operations (Section 16.13).

The shared class spMover inherits all the instance variables and functions of the class spAction (Section 46). The class spMover defines the following instance variables, with the variables in the external API in bold and the variables that are fundamental in the sense that they could not be properly supported by an application programmer underlined:

spMoverC - Class descriptor (Section 15.1).
shared X - spTransform interpolation points (Section 49.1).
shared T - Times of corresponding interpolation points (Section 49.2).
Queue - Queue of interpolation points (Section 49.3).

The class spMover defines the following functions:

spMoverFunction - Calculates spTransform via interpolation (Section 49.4).
spMoverNew - Create new spMover object. (Section 15.28).

49.1 X (Internal)

```
public float[] X; //* [spTransform6:102] internal

      spTransform6 spMoveriGetX(sp Object)
      spTransform6 spMoveriGetOldX(sp Object)
      void spMoveriSetX(sp Object, spTransform6 X)

      spTransform6 sqMoverGetX(sp Object)
      spTransform6 sqMoverGetOldX(sp Object)
      void sqMoverSetX(sp Object, spTransform6 X)
```

The *X* shared instance variable of an spMover records six spTransform values that are the basis of calculating an interpolated object position. It is manipulated by the smooth motion software and should not be directly modified by an application. Information about the X is shared between processes.

49.2 T (Internal)

```
public int [] T; //* [spTimeStamp6:6] internal

      spTimeStamp6 spMoveriGetT(sp Object)
      spTimeStamp6 spMoveriGetOldT(sp Object)
      void spMoveriSetT(sp Object, spTimeStamp6 X)

      spTimeStamp6 sqMoverGetT(sp Object)
      spTimeStamp6 sqMoverGetOldT(sp Object)
      void sqMoverSetT(sp Object, spTimeStamp6 X)
```

The *T* shared instance variable of an spMover record six time values that correspond to the six spTransform values in the X variable. Together, the X and T variables operate as a ring buffer. The T variable is manipulated by the smooth motion software and should not be directly modified by an application. Information about the T is shared between processes.

49.3 Queue (Internal)

```
transient public int Queue; //* [spPath] internal

      spPath spMoveriGetQueue(sp Object)
      void spMoveriSetQueue(sp Object, spPath X)

      spPath sqMoverGetQueue(sp Object)
      void sqMoverSetQueue(sp Object, spPath X)
```

The *queue* local instance variable of an spMover contains a queue of future motions for an object. It is manipulated by the smooth motion software and should not be directly modified by an application. Information about the queue is maintained separately in each process.

49.4 spMoverFunction (Internal)

```
spBoolean spMoverFunction(sp Action, sp Object)
```

Action - The spMover action being applied.
Object - Object to move.
Return value - Returns True when motion over.

Uses interpolation to calculate the Transform of an object at the current moment.

A Java Declaration File

The following shows a Java definition appropriate for input to SPOT (Section 1.6) that corresponds to the (Internal) Spline Version 3.0 API presented in this document. Since all of the functions are part of the system core, they are all native. The functions spWMRegister and spWMDeregister are not included in the Java below, because they are not part of the Java API and cannot be expressed in Java.

The Java code below is included both as an extended example of SPOT input and as a concise summary of what is available in the (Internal) Spline Version 3.0 API. Discussions of the variables and functions shown can be found by looking them up in the table of contents of this documentation. They appear in the same order below as in the table of contents.

```
public class spApp {
  native public static String ChooseServer();
   //* [char * spAppChooseServer()]
  native public static void Init();
   //* [void spAppInit()]
  native public static boolean Body();
   //* [spBoolean spAppBody()]
  native public static void Finish();
   //* [void spAppFinish()]
}


public class spVisual extends spApp {
  native public final static void Init();
   //* [void spVisualInit()]
  native public final static void Finish();
   //* [void spVisualFinish()]
}


public class spAudio extends spApp {
  native public final static void Init();
   //* [void spAudioInit()]
  native public final static void Finish();
   //* [void spAudioFinish()]
}
```

```
public class spWM {
  transient int CPtr;
  public static int Me;                              //* [spName]
  public static int MainOwner;                       //* [spName] readonly
  public static int SystemOwner;                     //* [spName] internal
  public static String Error;                        //* [char *:500]
  public static String LastError;                    //* [char *:500] readonly
  public static int Interval;                        //* [spDuration] readonly
  public static int DesiredInterval;                 //* [spDuration]
  public static int Week;                            //* [long] readonly
  public static int Msec;                            //* [spDuration] readonly
  public static int Window;                          //* [spWindow]
  public static String DNSName;                      //* [char *:500] readonly
  public static short Port;                          //* [short] readonly
  public static int MsgRejectionQueue;               //* [void *] internal
  native private static int New(String Server, int V);
   //* [spWM spWMNew(char * Server, spTransferVector V)]
  native public final static void Remove();
   //* [void spWMRemove()]
  native public final static void Update();
   //* [void spWMUpdate()]
  native public final static int GenerateOwner();
   //* [spName spWMGenerateOwner()]
  native public final static void ReportError(sp Object, int Code, String Description);
   //* [void spWMReportError(sp Object, long Code, char *:500 Description)]
}


public abstract class spFn {
  public abstract boolean F(sp object);
}


public class spMask {
  public static final int MINE = 1;                  //* [spMask]
  public static final int OTHERS = 2;                //* [spMask]
  public static final int NORMAL = 3;                //* [spMask]
  public static final int SYSTEM = 8;                //* [spMask] internal
  public static final int CALLBACKS = 16;            //* [spMask] internal
  public static final int ALL = -1;                  //* [spMask] internal
}
```

```
public class spTransform {
  public static final int X = 0;                    //* [long]
  public static final int Y = 1;                    //* [long]
  public static final int Z = 2;                    //* [long]
  public static final int RX = 3;                   //* [long]
  public static final int RY = 4;                   //* [long]
  public static final int RZ = 5;                   //* [long]
  public static final int RA = 6;                   //* [long]
  public static final int SX = 7;                   //* [long]
  public static final int SY = 8;                   //* [long]
  public static final int SZ = 9;                   //* [long]
  public static final int SOX = 10;                 //* [long]
  public static final int SOY = 11;                 //* [long]
  public static final int SOZ = 12;                 //* [long]
  public static final int SOA = 13;                 //* [long]
  public static final int CX = 14;                  //* [long]
  public static final int CY = 15;                  //* [long]
  public static final int CZ = 16;                  //* [long]
  native public final static float[] Copy(float[] Destination, float[] Source);
   //* [spTransform:17 spTransformCopy(spTransform:17 Destination, spTransform:17 Source)]
  native public final static float[] FromIdent(float[] Transform);
   //* [spTransform:17 spTransformFromIdent(spTransform:17 Transform)]
  native public final static float[] GetTranslation(float[] Transform);
   //* [spVector:3 spTransformGetTranslation(spTransform:17 Transform)]
  native public final static float[] SetTranslation(float[] Transform, float[] Translation);
   //* [spTransform:17 spTransformSetTranslation(spTransform:17 Transform, spVector:3 Translation)]
  native public final static float[] GetRotation(float[] Transform);
   //* [spRotation:4 spTransformGetRotation(spTransform:17 Transform)]
  native public final static float[] SetRotation(float[] Transform, float[] Rotation);
   //* [spTransform:17 spTransformSetRotation(spTransform:17 Transform, spRotation:4 Rotation)]
  native public final static float[] GetScale(float[] Transform);
   //* [spVector:3 spTransformGetScale(spTransform:17 Transform)]
  native public final static float[] SetScale(float[] Transform, float[] Vector);
   //* [spTransform:17 spTransformSetScale(spTransform:17 Transform, spVector:3 Vector)]
  native public final static float[] GetScaleOrientation(float[] Transform);
   //* [spRotation:4 spTransformGetScaleOrientation(spTransform:17 Transform)]
  native public final static float[] SetScaleOrientation(float[] Transform, float[] R);
   //* [spTransform:17 spTransformSetScaleOrientation(spTransform:17 Transform, spRotation:4 R)]
  native public final static float[] GetCenter(float[] Transform);
   //* [spVector:3 spTransformGetCenter(spTransform:17 Transform)]
  native public final static float[] SetCenter(float[] Transform, float[] Center);
   //* [spTransform:17 spTransformSetCenter(spTransform:17 Transform, spVector:3 Center)]
}
```

```
public class spVector {
  public static final int X = 0;                    //* [long]
  public static final int Y = 1;                    //* [long]
  public static final int Z = 2;                    //* [long]
  public static final float[] ZERO = {0.0, 0.0, 0.0};
         //* [spVector:3=((spVector)spVectorDataZero)]
  public static final float[] AXISX = {1.0, 0.0, 0.0};
         //* [spVector:3=((spVector)spVectorDataAxisX)]
  public static final float[] AXISY = {10.0, 1.0, 0.0};
         //* [spVector:3=((spVector)spVectorDataAxisY)]
  public static final float[] AXISZ = {10.0, 0.0, 1.0};
         //* [spVector:3=((spVector)spVectorDataAxisZ)]
  native public final static float[] Copy(float[] Destination, float[] Source);
   //* [spVector:3 spVectorCopy(spVector:3 Destination, spVector:3 Source)]
  native public final static float[] SetFromScalar(float[] Vector, float Scalar);
   //* [spVector:3 spVectorSetFromScalar(spVector:3 Vector, float Scalar)]
  native public final static boolean Equals(float[] A, float[] B);
   //* [spBoolean spVectorEquals(spVector:3 A, spVector:3 B)]
  native public final static boolean EqualsDelta(float[] A, float[] B, float Tolerance);
   //* [spBoolean spVectorEqualsDelta(spVector:3 A, spVector:3 B, float Tolerance)]
  native public final static float[] Add(float[] A, float[] B);
   //* [spVector:3 spVectorAdd(spVector:3 A, spVector:3 B)]
  native public final static float[] Subtract(float[] A, float[] B);
   //* [spVector:3 spVectorSubtract(spVector:3 A, spVector:3 B)]
  native public final static float[] MultiplyByScalar(float[] Vector, float Scalar);
   //* [spVector:3 spVectorMultiplyByScalar(spVector:3 Vector, float Scalar)]
  native public final static float[] DivideByScalar(float[] Vector, float Scalar);
   //* [spVector:3 spVectorDivideByScalar(spVector:3 Vector, float Scalar)]
  native public final static float[] CrossProduct(float[] A, float[] B);
   //* [spVector:3 spVectorCrossProduct(spVector:3 A, spVector:3 B)]
  native public final static float DotProduct(float[] A, float[] B);
   //* [float spVectorDotProduct(spVector:3 A, spVector:3 B)]
  native public final static float[] ComposeScales(float[] A, float[] B);
   //* [spVector:3 spVectorComposeScales(spVector:3 A, spVector:3 B)]
  native public final static float Length(float[] Vector);
   //* [float spVectorLength(spVector:3 Vector)]
  native public final static float[] Normalize(float[] Vector);
   //* [spVector:3 spVectorNormalize(spVector:3 Vector)]
}
```

```
public class spRotation {
  public static final int X = 0;                    //* [long]
  public static final int Y = 1;                    //* [long]
  public static final int Z = 2;                    //* [long]
  public static final int A = 3;                    //* [long]
  native public final static float[] Copy(float[] Destination, float[] Source);
   //* [spRotation:4 spRotationCopy(spRotation:4 Destination, spRotation:4 Source)]
  native public final static float[] FromIdent(float[] Rotation);
   //* [spRotation:4 spRotationFromIdent(spRotation:4 Rotation)]
  native public final static float[] GetAxis(float[] Rotation);
   //* [spVector:3 spRotationGetAxis(spRotation:4 Rotation)]
  native public final static float[] SetAxis(float[] Rotation, float[] Axis);
   //* [spRotation:4 spRotationSetAxis(spRotation:4 Rotation, spVector:3 Axis)]
  native public final static float GetAngle(float[] Rotation);
   //* [float spRotationGetAngle(spRotation:4 Rotation)]
  native public final static float[] SetAngle(float[] Rotation, float Angle);
   //* [spRotation:4 spRotationSetAngle(spRotation:4 Rotation, float Angle)]
  native public final static float[] ToQuat(float[] Rotation, float[] Quat);
   //* [spQuaternion:4 spRotationToQuat(spRotation:4 Rotation, spQuaternion:4 Quat)]
  native public final static float[] FromQuat(float[] Rotation, float[] Quat);
   //* [spRotation:4 spRotationFromQuat(spRotation:4 Rotation, spQuaternion:4 Quat)]
  native public final static float[] ToAngles(float[] Rotation, float[] Vector);
   //* [spVector:3 spRotationToAngles(spRotation:4 Rotation, spVector:3 Vector)]
  native public final static float[] FromAngles(float[] Rotation, float[] Angles);
   //* [spRotation:4 spRotationFromAngles(spRotation:4 Rotation, spVector:3 Angles)]
  native public final static float[] Mult(float[] A, float[] B);
   //* [spRotation:4 spRotationMult(spRotation:4 A, spRotation:4 B)]
  native public final static float[] LookAt(float[] R, float[] From, float[] To, float[] Up);
   //* [spRotation:4 spRotationLookAt(spRotation:4 R, spVector:3 From, spVector:3 To, spVector:3 Up)]
}


public class spQuaternion {
  native public final static float[] Copy(float[] Destination, float[] Source);
   //* [spQuaternion:4 spQuaternionCopy(spQuaternion:4 Destination, spQuaternion:4 Source)]
  native public final static float[] FromIdent(float[] Quaternion);
   //* [spQuaternion:4 spQuaternionFromIdent(spQuaternion:4 Quaternion)]
  native public final static float[] Mult(float[] A, float[] B);
   //* [spQuaternion:4 spQuaternionMult(spQuaternion:4 A, spQuaternion:4 B)]
}
```

```
public class spMatrix {
  native public final static float[] Copy(float[] Destination, float[] Source);
   //* [spMatrix:16 spMatrixCopy(spMatrix:16 Destination, spMatrix:16 Source)]
  native public final static float[] FromIdent(float[] Matrix);
   //* [spMatrix:16 spMatrixFromIdent(spMatrix:16 Matrix)]
  native public final static float[] GetTranslation(float[] Matrix);
   //* [spVector:3 spMatrixGetTranslation(spMatrix:16 Matrix)]
  native public final static float[] SetTranslation(float[] Matrix, float[] Translation);
   //* [spMatrix:16 spMatrixSetTranslation(spMatrix:16 Matrix, spVector:3 Translation)]
  native public final static float[] FromTransform(float[] Matrix, float[] Transform);
   //* [spMatrix:16 spMatrixFromTransform(spMatrix:16 Matrix, spTransform:17 Transform)]
  native public final static float[] ToTransform(float[] Matrix, float[] Transform);
   //* [spTransform:17 spMatrixToTransform(spMatrix:16 Matrix, spTransform:17 Transform)]
  native public final static float[] Inverse(float[] spMatrix);
   //* [spMatrix:16 spMatrixInverse(spMatrix:16 spMatrix)]
  native public final static float[] Mult(float[] A, float[] B);
   //* [spMatrix:16 spMatrixMult(spMatrix:16 A, spMatrix:16 B)]
  native public final static float[] MultVector(float[] Matrix, float[] Vector);
   //* [spVector:3 spMatrixMultVector(spMatrix:16 Matrix, spVector:3 Vector)]
}


public class spPath {
  transient int CPtr;
  native private static int New();
   //* [spPath spPathNew()]
  native public final static int AppendTransform(int Path, float[] Point, int Duration);
   //* [spPath spPathAppendTransform(spPath Path, spTransform:17 Point, spDuration Duration)]
  native public final static int GetTransform(int Path, int Index, float[] Transform);
   //* [spDuration spPathGetTransform(spPath Path, long Index, spTransform:17 Transform)]
  native public final static int Copy(int Path);
   //* [spPath spPathCopy(spPath Path)]
  native public final static void Save(int Path, String Name, String File);
   //* [void spPathSave(spPath Path, char * Name, char * File)]
  native public final static int Load(int Path, String Name, String File);
   //* [spPath spPathLoad(spPath Path, char * Name, char * File)]
  native public final static int ChangeStartPoint(int Path, float[] Transform);
   //* [spPath spPathChangeStartPoint(spPath Path, spTransform:17 Transform)]
  native public final static int Thin(int Path, float Tolerance);
   //* [spPath spPathThin(spPath Path, float Tolerance)]
}


public class spFormat {
  public static final long LINEAR8MONO16 = 0x1F40000000100001L; //* [spFormat]
  public static final long LINEAR16MONO16 = 0x3E80000000100001L; //* [spFormat]
  public static final long LINEAR32MONO16 = 0x7D00000000100001L; //* [spFormat]
  native public final static int DurationFromLength(long Format, int Bytes);
   //* [spDuration spFormatDurationFromLength(spFormat Format, long Bytes)]
  native public final static int LengthFromDuration(long Format, int Duration);
   //* [long spFormatLengthFromDuration(spFormat Format, spDuration Duration)]
}
```

```
public interface sp {
  transient int CPtr;
  public static final float DEGREES = 0.0174532925199; //* [float]
  transient public int LocalPtr;                    //* [void *] internal
  transient public int NextPtr;                     //* [void *] internal
  transient public int Marker;                      //* [long] internal
  public short DescriptionLength;                   //* [short] internal
  public short Counter;                             //* [short] internal
  public int Name;                                  //* [spName] internal
  public spClass Class;                             //* [sp] readonly
  public int Owner;                                 //* [spName]
  public spLocale Locale;                           //* [sp] internal
  public int SharedBits;                            //* [long] internal
  public sp Parent;                                 //* [sp]
  public boolean IsRemoved;                         //* [spBoolean] readonly
  public boolean ForceReliable;                     //* [spBoolean] internal
  public boolean InhibitReliable;                   //* [spBoolean] internal
  transient public short LocalBits;                 //* [short] internal
  transient public boolean IsNew;                   //* [spBoolean] readonly
  transient public int AppData;                     //* [void *]
  transient public boolean MessageNeeded;           //* [spBoolean] internal
  transient public boolean Change;                  //* [spBoolean] internal
  transient public int OldPtr;                      //* [void *] internal
  transient public int JavaPtr;                     //* [void *] internal
  transient public int Referrers;                   //* [spNameInfoPtr] internal
  transient public int Alerters;                    //* [void *] internal
  transient public int Msgs;                        //* [void *] internal
  transient public int LastUpdateTime;              //* [spTimeStamp] internal
  native public void Initialization();
   //* [void spInitialization(sp Object)]
  native public final void Remove();
   //* [void spRemove(sp Object)]
  native public final void ExamineChildren(int Mask, spFn F);
   //* [void spExamineChildren(sp Object, spMask Mask, spFn F, void * Data)]
  native public final void ExamineDescendants(int Mask, spFn F);
   //* [void spExamineDescendants(sp Object, spMask Mask, spFn F, void * Data)]
  native public final sp Topmost();
   //* [sp spTopmost(sp Object)]
  native public final void Print();
   //* [void spPrint(sp Object)]
  native public final boolean LocallyOwned();
   //* [spBoolean spLocallyOwned(sp Object)]
  native public final void SetParent(sp Parent);
   //* [void spSetParent(sp Object, sp Parent)]
}
```

```
public interface spPositioning extends sp {
  public float[] Transform;                          //* [spTransform:17]
  transient public float[] Matrix;                   //* [spMatrix:16] internal
  transient public boolean MatrixOK;                 //* [spBoolean] internal
  transient public float[] MatrixInverse;            //* [spMatrix:16] internal
  transient public boolean MatrixInverseOK;          //* [spBoolean] internal
  native public final float[] Matrix();
   //* [spMatrix:16 spPositioningMatrix(sp Object)]
  native public final float[] MatrixInverse();
   //* [spMatrix:16 spPositioningMatrixInverse(sp Object)]
  native public final boolean Localize(spLocale Destination, boolean ChooseSmallest);
   //* [spBoolean spPositioningLocalize(sp Object, sp Destination, spBoolean ChooseSmallest)]
  native public final float[] RelativeMatrix(spPositioning Object, float[] Matrix);
   //* [spMatrix:16 spPositioningRelativeMatrix(sp X, sp Object, spMatrix:16 Matrix)]
  native public final float[] RelativeVector(spPositioning X, float[] Vector);
   //* [spVector:3 spPositioningRelativeVector(sp Object, sp X, spVector:3 Vector)]
  native public final float Distance(spPositioning X);
   //* [float spPositioningDistance(sp Object, sp X)]
  native public final void LookAt(spPositioning Target);
   //* [void spPositioningLookAt(sp Object, sp Target)]
  native public final void GoThru(float[] Transform, int Time);
   //* [void spPositioningGoThru(sp Object, spTransform:17 Transform, spDuration Time)]
  native public final void StopAt(float[] Transform, int Time);
   //* [void spPositioningStopAt(sp Object, spTransform:17 Transform, spDuration Time)]
  native public final void Stop();
   //* [void spPositioningStop(sp Object)]
  native public final void FollowPath(int Path);
   //* [void spPositioningFollowPath(sp Object, spPath Path)]
  native public final int MotionTimeLeft();
   //* [spDuration spPositioningMotionTimeLeft(sp Object)]
  native public final int GetMotionQueue();
   //* [spPath spPositioningGetMotionQueue(sp Object)]
  native public final void FlushMotionQueue();
   //* [void spPositioningFlushMotionQueue(sp Object)]
  native public final void SetTransform(float[] Transform);
   //* [void spPositioningSetTransform(sp Object, spTransform:17 Transform)]
  native public void Initialization();
   //* [void spPositioningInitialization(sp Object)]
}


public interface spDisplaying extends sp {
  public spVisualDefinition VisualDefinition;        //* [sp]
  public float InRadius;                             //* [float]
  public float OutRadius;                            //* [float]
  transient public int GraphicsNode;                 //* [void *] internal
}


public interface spLinking extends sp {
  public String URL;                                 //* [spFixedAscii] readonly
  public int Checksum;                               //* [long] internal
  transient public String FileName;                  //* [char *]
  transient public int Data;                         //* [void *] readonly
  native public final void URLAltered();
   //* [void spLinkingURLAltered(sp Link)]
  native public void ReadData();
   //* [void spLinkingReadData(sp Link)]
}
```

```
public interface spMultilinking extends spLinking {
  transient public int Multipart;                    //* [long] internal
  transient public String IndexName;                 //* [char *]
  native public int New(v URL);
   //* [sp spMultilinkingNew(v URL)]
  native public int Select();
   //* [long spMultilinkingSelect(sp Link)]
}


public interface spBeaconing extends sp {
  public String Tag;                                 //* [spFixedAscii] readonly
  public boolean Suppress;                           //* [spBoolean]
}


public interface spObserving extends sp {
  public boolean Audio;                              //* [spBoolean]
  public boolean IgnoreNearby;                       //* [spBoolean]
}


public interface spVisualParameters extends sp {
  public float FarClip;                              //* [float]
  public float NearClip;                             //* [float]
  public float Field;                                //* [float]
  public int Interval;                               //* [spDuration]
}


public interface spAudioParameters extends sp {
  public spAudioSource Focus;                        //* [sp]
  public boolean Live;                               //* [spBoolean]
  public long Format;                                //* [spFormat]
}


public class spVisualDefinition implements spMultilinking {
  native private static int New(String URL);
   //* [sp spVisualDefinitionNew(spFixedAscii URL)]
  native public void ReadData();
   //* [void spVisualDefinitionReadData(sp Link)]
  native public int Select();
   //* [long spVisualDefinitionSelect(sp Link)]
}


public class spSound implements spMultilinking {
  transient public int Duration;                     //* [spDuration] readonly
  native private static int New(String URL);
   //* [sp spSoundNew(spFixedAscii URL)]
  native public final spAction Play(spAudioSource Source, boolean Loop, float Gain);
   //* [sp spSoundPlay(sp Sound, sp Source, spBoolean Loop, float Gain)]
  native public int Select();
   //* [long spSoundSelect(sp Link)]
  native public void ReadData();
   //* [void spSoundReadData(sp Link)]
}
```

```
public class spLocale implements spLinking, spBeaconing, spDisplaying {
  public spBoundary Boundary;                        //* [sp]
  transient public int NumNeighbors;                 //* [long] internal
  native private static int New(String URL, String Tag);
   //* [sp spLocaleNew(spFixedAscii URL, spFixedAscii Tag)]
  native public final spLocale Choose(float[] P);
   //* [sp spLocaleChoose(sp L, spMatrix:16 P)]
  native public final float[] ExportMatrix(spLocale Destination);
   //* [spMatrix:16 spLocaleExportMatrix(sp L, sp Destination)]
  native public void ReadData();
   //* [void spLocaleReadData(sp Link)]
}


public class spBoundary implements spLinking {
  transient public float Volume;                     //* [float] readonly
  native private static int New(String URL);
   //* [sp spBoundaryNew(spFixedAscii URL)]
  native public int Below(float[] P, float[] Q);
   //* [long spBoundaryBelow(sp Boundary, spVector:3 P, spVector:3 Q)]
  native public int Above(float[] P, float[] Q);
   //* [long spBoundaryAbove(sp Boundary, spVector:3 P, spVector:3 Q)]
  native public int Inside(float[] P);
   //* [long spBoundaryInside(sp Boundary, spVector:3 P)]
  native public void ReadData();
   //* [void spBoundaryReadData(sp Link)]
}


public class spTerrain extends spBoundary {
  native private static int New(String URL);
   //* [sp spTerrainNew(spFixedAscii URL)]
  native public final int Below(spTerrain Terrain, float[] P, float[] Q);
   //* [long spTerrainBelow(sp Terrain, spVector:3 P, spVector:3 Q)]
  native public final int Above(float[] P, float[] Q);
   //* [long spTerrainAbove(sp Terrain, spVector:3 P, spVector:3 Q)]
  native public int Inside(float[] P);
   //* [long spTerrainInside(sp Terrain, spVector:3 P)]
  native public void ReadData();
   //* [void spTerrainReadData(sp Link)]
}
```

```
public class spClass implements spLinking {
  transient public String ClassName;                //* [spAscii32:32] readonly
  transient public boolean LoadData;                //* [spBoolean]
  transient public int ReadDataFn;                  //* [void *] internal
  transient public int SelectFn;                    //* [void *] internal
  transient public int Superclasses;                //* [void *] readonly internal
  transient public short NumSuperclasses;           //* [short] readonly internal
  transient public short Size;                      //* [short] internal
  transient public short Level;                     //* [short] internal
  transient public short LocalOffset;               //* [short] internal
  transient public short SharedOffset;              //* [short] internal
  transient public short SharedBitNum;              //* [short] internal
  transient public short LocalBitNum;               //* [short] internal
  transient public int TimeStampOffsets;            //* [int *] internal
  transient public boolean SendViaLocale;           //* [spBoolean] internal
  transient public boolean SendViaTCP;              //* [spBoolean] internal
  transient public int NumVariables;                //* [long] internal
  transient public int MethodTable;                 //* [void *] readonly internal
  native public int NewObj(int Extra);
   //* [sp spClassNewObj(sp ClassDescriptor, long Extra)]
  native public int NewLink(String URL);
   //* [sp spClassNewLink(sp ClassDescriptor, spFixedAscii URL)]
  native private static int New(String URL);
   //* [sp spClassNew(spFixedAscii URL)]
  native public final boolean Eq(spClass ClassB);
   //* [spBoolean spClassEq(sp ClassA, sp ClassB)]
  native public final boolean Leq(spClass Superclass);
   //* [spBoolean spClassLeq(sp Subclass, sp Superclass)]
  native public final boolean Examine(int Mask, spFn F);
   //* [spBoolean spClassExamine(sp C, spMask Mask, spFn F, void * Data)]
  native public final spIntervalCallback Monitor(int Mask, spFn F);
   //* [sp spClassMonitor(sp C, spMask Mask, spFn F, void * Data)]
  native public void ReadData();
   //* [void spClassReadData(sp Link)]
}


public class spThing implements spPositioning, spDisplaying {
}


public class spRoot extends spThing {
}


public class spAvatar extends spRoot {
  public boolean IsBot;                             //* [spBoolean]
}
```

```
public class spAudioSource implements spPositioning, spDisplaying, spAudioParameters {
  transient public int Duration;                    //* [spDuration] readonly
  transient public long ExternalFormat;             //* [spFormat] internal
  native public final void Setup(boolean W, boolean R, long F, int D);
   //* [void spAudioSourceSetup(sp S, spBoolean W, spBoolean R, spFormat F, spDuration D)]
  native public final void Write(String Data);
   //* [void spAudioSourceWrite(sp Source, char * Data)]
  native public final String Read();
   //* [char * spAudioSourceRead(sp Source)]
}


public class spBeacon implements spBeaconing {
  native private static int New(String Tag);
   //* [sp spBeaconNew(spFixedAscii Tag)]
}


public class spPositionedBeacon implements spBeaconing, spPositioning {
}


public class spSpeaking extends spPositionedBeacon {
  native private static int New(String HostName);
   //* [sp spSpeakingNew(spFixedAscii HostName)]
}


public class spHearing extends spPositionedBeacon implements spAudioParameters {
  native private static int New(String HostName);
   //* [sp spHearingNew(spFixedAscii HostName)]
}


public class spSeeing extends spPositionedBeacon implements spVisualParameters {
  native private static int New(String HostName);
   //* [sp spSeeingNew(spFixedAscii HostName)]
}


public class spSimulationObserver implements spObserving {
}


public class spVisualObserver implements spObserving, spVisualParameters {
}


public class spAudioObserver implements spObserving, spAudioParameters {
  native public void Initialization(spAudioObserver Object);
   //* [void spAudioObserverInitialization(sp Object)]
}
```

```
public class spIntervalCallback extends sp {
  public int Interval;                              //* [spDuration]
  transient public spFn F;                          //* [spFn] readonly
  transient public int FState;                      //* [void *] readonly
  transient public int NextTriggerTime;             //* [spTimeStamp] internal
  transient public int IntNext;                     //* [void *] internal
  transient public int IntPrev;                     //* [void *] internal
  native private static int New(int Interval, spFn F);
   //* [sp spIntervalCallbackNew(spDuration Interval, spFn F, void * FState)]
}


public class spAlerter extends spIntervalCallback {
  transient public spFn P;                          //* [spFn] readonly
  transient public int PState;                      //* [void *] readonly
  public int Mask;                                  //* [spMask]
  transient public int ChgNext;                     //* [void *] internal
  transient public int ChgPrev;                     //* [void *] internal
  native private static int New(sp X, int I, spFn P, spFn F);
   //* [sp spAlerterNew(sp X, spDuration I, spFn P, void * PState, spFn F, void * FState)]
  native public void Initialization();
   //* [void spAlerterInitialization(sp Action)]
}


public class spBeaconMonitor extends spAlerter {
  public String Pattern;                            //* [spFixedAscii]
  native private static int New(String Pattern, spFn F, int I);
   //* [sp spBeaconMonitorNew(spFixedAscii Pattern, spFn F, void * S, spDuration I)]
}


public class spBeaconGoto extends spBeaconMonitor {
  transient public spObserving Object;              //* [sp] readonly
  native private static int New(String P, spObserving Object, spFn F, int I);
   //* [sp spBeaconGotoNew(spFixedAscii P, sp Object, spFn F, void * S, spDuration I)]
}


public interface spAction extends sp {
  native public boolean Function(sp Parent);
   //* [spBoolean spActionFunction(sp Action, sp Parent)]
}


public class spOwnershipRequest implements spAction {
  transient public spFn F;                          //* [spFn] readonly
  transient public int FState;                      //* [void *] readonly
  public int Timeout;                               //* [spDuration]
  transient public int TimeAlive;                   //* [spDuration]
  native private static int New(sp Object, spFn F, int Timeout);
   //* [sp spOwnershipRequestNew(sp Object, spFn F, void * FState, spDuration Timeout)]
  native public final boolean Function(sp Object);
   //* [spBoolean spOwnershipRequestFunction(sp Action, sp Object)]
  native public final void Grant();
   //* [void spOwnershipRequestGrant(sp Request)]
}
```

```
public class spDoSoundPlay implements spAction {
  public spSound Sound;                              //* [sp] internal
  public boolean Loop;                               //* [spBoolean] internal
  public float Gain;                                 //* [float] internal
  native public final boolean Function(spAudioSource Object);
   //* [spBoolean spDoSoundPlayFunction(sp Action, sp Object)]
}


public class spMover implements spAction {
  public float[] X;                                  //* [spTransform6:102] internal
  public int [] T;                                   //* [spTimeStamp6:6] internal
  transient public int Queue;                        //* [spPath] internal
  native public final boolean Function(spThing Object);
   //* [spBoolean spMoverFunction(sp Action, sp Object)]
}
```

B Quick Function Reference

The following is a quick reference and index for the complex functions in the (Internal) Spline Version 3.0 API. In the interest of brevity, it does not include: variable access functions, functions of the form spKC() for obtaining class descriptors, and zero-argument creation functions of the form spKNew().

Each entry consists of two lines. The first line indicates which function in which class is being summarized and the page on which the function is described in detail. The second line shows the complete signature of the function.

For greater convenience in looking up the functions, the entries are ordered alphabetically based on the first line of each entry. That is to say, the entries are sorted primarily on the name of the method itself and secondarily on the name of the class that contains the method. For Example, spClassExamine is ordered primarily by "Examine" and only secondarily by "spClass".

ToQuat in spRotation on page 72
```
spQuaternion spRotationToQuat(spRotation Rotation, spQuaternion Quat)
```
ToTransform in spMatrix on page 78
```
spTransform spMatrixToTransform(spMatrix Matrix, spTransform Transform)
```
Topmost in sp on page 103
```
sp spTopmost(sp Object)
```
URLAltered in spLinking on page 122
```
void spLinkingURLAltered(sp Link)
```
Update in spWM on page 46
```
void spWMUpdate()
```
Write in spAudioSource on page 172
```
void spAudioSourceWrite(sp Source, char * Data)
```