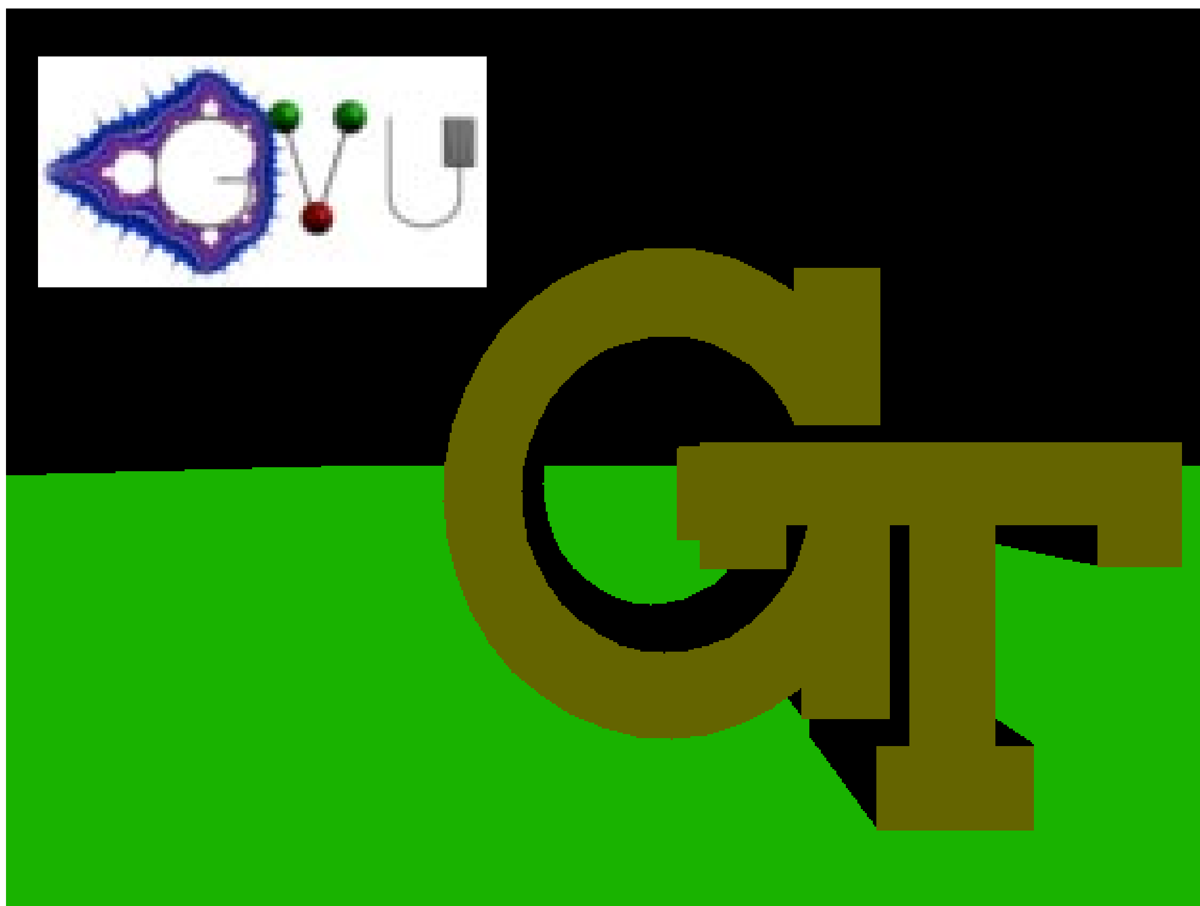
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Scholarly Materials And Research @ Georgia Tech

THE SIMPLE VIRTUAL ENVIRONMENT LIBRARY

Version 2.0
User's Guide



Drew Kessler, Rob Kooper,
Larry F. Hodges

{drew, kooper, hodges}@cc.gatech.edu

Graphics, Visualization and Usability Center,
Georgia Institute of Technology, USA.

April 16, 1997

THE SIMPLE VIRTUAL ENVIRONMENT LIBRARY

User's Guide
Version 2.0

Drew Kessler, Rob Kooper,
Larry F. Hodges,

{drew, kooper, hodges}@cc.gatech.edu
Graphics, Visualization and Usability Center,
Georgia Institute of Technology, USA.

ABSTRACT

The *Simple Virtual Environments* (SVE) C library provides a framework for the development of virtual environment (VE) applications. The library provides the default components of simple VE applications (such as fly-throughs), allowing these applications to be quickly implemented, and allows applications to selectively alter, enhance, or replace components such as user interactions, animations, rendering, and input device polling. The library also allows the hardware and software configuration (devices used and placement in the workspace, location of remote servers, directories, etc.) to be given at run-time using an initialization file. Therefore, SVE provides support for rapid prototyping as well as complete implementation of simple and complex VE applications.

Differences between version 2.0 and 1.5

Although some differences exist between version 1.5 and 2.0 (see "Converting from v1.5" on page 1), most of the changes are enhancements to the library that allow it to more completely fulfill the task of supporting VE applications. New configurations flags have been added to separate lighting from gouraud shading, and allow for an application with no display, or with no audio. Level of detail switching is supported at the object level, where an object may contain a list of geometric descriptions which are used for a particular range of distances from the user. Many new texture types are supported, including images with 1, 2, 3, or 4 components per pixel, and two methods of blending the texture with the geometry it is mapped onto. These features support transparent textures, among other things. In addition, a series of textures can be given to be mapped onto a geometry, with the actual texture used for a particular frame chosen automatically or by the application. The initialization file can now contain commands to move, rotate, and scale objects, to alter their initial position. New functions are provided to move, rotate, and scale objects in the coordinate system of objects other than the parent object. New functions allow the application to alter, extend, or override object culling from the rendered scene (usually done using the viewing volume of the user). Finally, a new function, `SVE_pickObject()`, allows for precise ray-object intersections. See the files ENHANCEMENTS.FROM.V1.5 and CONVERTING.FROM.V1.5.

Acknowledgments

The following people have contributed to this library: Doug Bowman, Elizabeth Bright, Eric Brittain, Jim Durbin, Kevin Hamilton, Drew Kessler, David Koller, Rob Kooper, E. J. Lee, Peter Lindstrom, Tom Meyer, Greg Newton, Jouke Verlinden, Zach Wartell, and Ben Watson.

Table of Contents

1. Introduction 7
1.1. Example Application 8
1.2. On-line Help 11
2. SVE Basics 13
2.1. Initialization of the Application. 13
2.1.1. Loading an Environment. 14
2.2. The Interaction Loop. 15
2.2.1. Input Handlers 15
2.2.2. Animation Routines 16
2.2.3. Frame Drawing Routines 17
2.3. Shutting Down. 18
2.4. Summary and Another Example 18
2.5. Tracker Devices and Other Configurations 20
3. Programming Details 23
3.1. Application configuration 23
3.1.1. Configuration Flags 23
3.1.2. Initialization File 28
3.1.3. Display Configuration Files 31
3.1.4. Directories 32
3.2. The SVE System 33
3.2.1. World State 33
3.2.2. System Overview 34
3.3. Events 36
3.3.1. Responding to Events 36
3.3.2. Generating Events 39
3.4. Animation and Rendering 41
3.4.1. Animation Routines 41
3.4.2. Animation Callbacks 43
3.4.3. Rendering Callbacks 45
3.4.4. Object Rendering 46
3.4.5. Navigation 49
3.5. Objects 51
3.5.1. Creating Objects 52
3.5.2. Loading and Saving Objects 52
3.5.3. Finding Objects 52
3.6. Object trees 53
3.6.1. Object tree manipulation 53
3.6.2. Rendered Object Tree (SVE_WORLD) 58
3.6.3. Tracker Objects 61
3.6.4. Loading and Saving Object Trees 61
3.7. Object Appearance 63
3.7.1. Object Position 63
3.7.2. Object Geometry 64
3.7.3. Object Boundaries 72

3.8. Colors and materials 74
3.9. Lighting and Shading 76
3.10. Sound 79
3.10.1. Audio support 79
3.10.2. Spatial audio support 80
3.11. Polling Devices 82
3.11.1. Tracking Devices 82
3.11.2. Hand Input Devices 83
3.11.3. User-Defined Polling Devices 89
3.12. 3D interactors 91
3.12.1. Registering a Widget Type 91
3.12.2. Creating a Widget 92
3.12.3. Creating a Widget From a File 94
3.12.4. Widget Event Function 95
3.12.5. Widget Instantiation 96
3.12.6. Widget Deletion 97
3.12.7. Retrieving Widget Data 97
3.13. Servers 99
3.14. Porting Version 1.5 Applications to Version 2.0 100

4. Future Directions 103

A Starter Kit 105
1. Introduction 105
2. General Overview 105
3. An Example 105
4. Second Example 107
5. Using Tracking Devices 110
6. Hardware Set Up 110
7. Where to Find Additional Help 110

B File Formats 111
1. Introduction 111
1.1. Environment Description Files - World and Objects 111
1.2. Initialization File 113
1.3. Display Configuration File 114
1.4. Gesture Description File 114
2. Formal Definition of the World Description File 117
3. Formal Definition of the Object Description File 120
4. Formal Definition of the Initialization File 125
5. Formal Definition of the Display Configuration File 129
6. Formal Definition of the Glove Gesture File 131

C Reference Manual 133
1. SVE Function Reference 133
1.1. Main SVE loop 133
1.2. SVE Configuration Routines 134
1.3. World/object utilities 134
1.3.1. Load/Save 134
1.3.2. Information 136

1.3.3. Object Creation and Deletion 137
1.3.4. Object Manipulation 138
1.3.5. Object Geometry 142
1.3.6. Geometry Routines 143
1.3.7. Low Level Geometry Routines 148
1.3.8. Material Routines 149
1.3.9. Texture swapping 151
1.3.10. Automatic object animation 152
1.3.11. Animation Callbacks 152
1.3.12. Object Tree Manipulation 153
1.3.13. Object Boundaries 155
1.3.14. Widgets 159
1.4. Callback utilities 160
1.4.1. Event callbacks 160
1.4.2. Frame callback 161
1.4.3. Object Frame Callbacks 162
1.4.4. Frame End Callbacks 162
1.4.5. Object Culling Callbacks 163
1.5. User Oriented Utilities 164
1.6. Rendering Functions 165
1.7. General utilities 167
1.7.1. State functions 167
1.7.2. Matrix functions 169
1.8. Default User Tree Information 170
1.8.1. Cursor Information 170
1.8.2. HMD Information 170
1.9. Polling Device Routines 170
1.9.1. Tracker Device Routines 171
1.9.2. Glove Utilities 172
1.10. Sound utilities 173
1.10.1. Audio commands 174
1.10.2. Spatial sound utilities 175
1.11. Text Output Routines 175
2. Data Structure Utilities 177
2.1. Linked List 177
2.2. Dynamic Array 181
2.3. File Parser Utility 183
2.4. Vector Routines 185
2.5. Matrix Routines 187
3. SVE Global Variables. 188
3.1. Information 188
3.2. Setup 190
4. SVE Data Structures 193

D Tracker Library 215
1. Using a tracker receiver 215
2. Tracker function definitions 216
3. Tracker data type 217

E Inset Utility 219

Index 225

1. Introduction

When the Graphics, Visualization and Usability center of Georgia Tech received the necessary equipment for a Virtual Environment set up, there was a need for common software to bring all of the hardware together in a way in which students could quickly begin exploring the research issues within Virtual Environments. Thus, the Simple Virtual Environment toolkit was born on November 4th, 1992 with the purpose of providing a software toolkit that a student with limited experience in Virtual Environments or advanced computer graphics could quickly develop a simple VE application, and have the resources to begin developing applications and study Virtual Environments and the applications of VE to real-world problems.

The SVE toolkit was begun from a mixture of previous efforts and experiences in developing Virtual Environment support software (at Delft University of Technology in Holland and at the University of Virginia in the USA), which revealed that most VE applications share a common set of functions such as loading/saving worlds and rendering objects. Recognizing this, SVE was developed to provide the most basic set of these functions and the more involved needs of various applications to maintain a "Virtual World", position tracking and hand input mechanisms. Current equipment includes a Silicon Graphics Indigo Elan, a Silicon Graphics Crimson and Onyx machines with Reality Engine graphics subsystems, Hewlett Packard workstations equipped with Freedom graphics from Evans and Sutherland, three Head Mounted Displays (HMD's) from Virtual Research (the Flight Helmet, the EYEGEN3, and the VR4), a Virtual I/O Glasses HMD, a dual-receiver Ascension Bird 3D, a Polhemus ISOTRAK II, and a Polhemus FASTRAK tracking system, as well as a Virtual Research Cyberglove. A diagram demonstrating the equipment handled by the SVE system is shown in Figure 1.

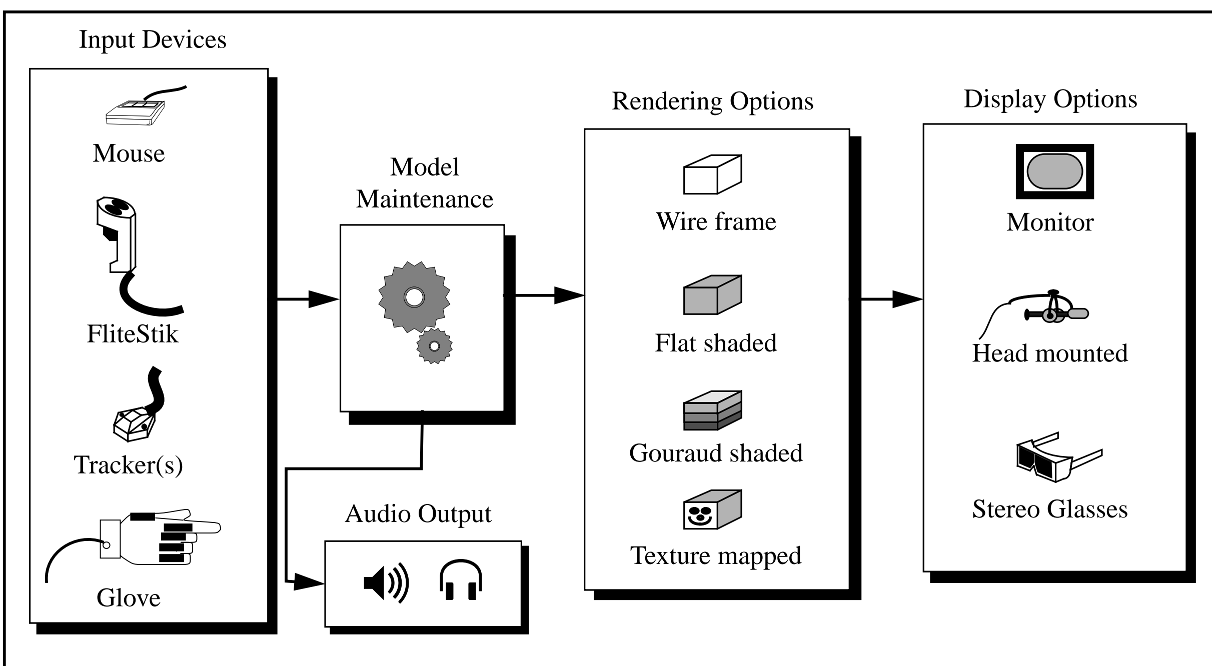


Figure 1. Configurations Provided by the SVE System

The library produces an image to be shown in a HMD which is continuously updated. The library uses either the SGI GL graphics library, or the OpenGL graphics library. It can use Wavefront objects or objects described in a format designed for the SVE library. A virtual world can be described using a world file format designed for the SVE library. The library is highly extensible. Code to read other geometric object formats can be written and easily incorporated. Code to interface to tracking and other input devices not already supported can be written and also easily incorporated. The library is designed to provide for easy

VE application development. For example, tracking devices are not needed during development, as the keyboard and mouse can be used to interact with the environment. Therefore the tedious job of setting up the physical devices required for VE applications need only be done for critical evaluation stages

Since this is an ongoing project and changes are constantly being made, we have felt it necessary to provide a mailing list where people can send questions to or remarks. To become a member of this list simply send E-mail to *majordomo@cc.gatech.edu*, with *subscribe sve* as the body of the message. You will then be added to the SVE mailing list. To send comments or bug report you can than send E-mail to *sve@cc.gatech.edu*. Mail will in turn then be sent to everyone else on the list. Please note, however, that SVE is currently an in-house system and only Georgia Tech and invited people will be allowed to join the mailing list. Although we make the documentation and manuals available to the general public, the source code is still only available within Georgia Tech. We hope to make it available outside of Georgia Tech in the future, but that involves quite a few legal issues that we would like to avoid as long as possible.

This document is intended to be a comprehensive description and reference for the SVE system. If you wish to quickly get your hands "dirty", we suggest that you begin with the SVE Starter Kit, which can be found as an independent file, Starter.fm, or in Appendix A (page 105) of this document. The Starter Kit provides a couple illustrative examples and instructions on how to interact with SVE applications.

1.1. Example Application

Before discussing the SVE-basics, we present to you an application that demonstrates how the SVE library allows for "simple" VE application implementations.

Source i: "Hello World" Example, My First SVE Program.

```
/************************************************************************
 * Example1 (sve module)
 *
 * This example shows a simple Virtual Environment using SVE. The default
 * key's as described in the SVE manual will work with this example.
 *
 ************************************************************************/

#include "sve.h"


main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

/************************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the SVE BASICS section of
 * the manual for a description of the different configurations you can use.
 ************************************************************************/
   printf("Starting application\n");
   SVE_init("Example1 (sve)", config, &argc, argv);

/************************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 ************************************************************************/
   if(!SVE_loadWorld("example1.world"))
   {
      printf("Error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/************************************************************************
 * SVE will take over control of the program until it is finnished.
 ************************************************************************/
   printf("Beginning event loop.\n");
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
```

```
}
```

This program uses only four SVE-functions: `SVE_init()`, `SVE_loadWorld()`, `SVE_beginEventLoop()` and `SVE_done()`. It allows you to "walk around" a cube, with "hello world" written the side. Neither the trackers nor the glove are used in this example (which would have to be specified by setting the "config" variable to a different configuration from "SVE_NORMAL"). The standard SVE key commands are recognized by this application. The mouse cursor must be over the SVE window to execute any of the commands given in the table below.

Table 1: Standard SVE Key/Mouse Commands

Key/Mouse Input	Command
q or Esc	Quit application.
t	Toggle between NTSC (Head mounted display for systems that can't get the video straight from the frame buffer) and display on the monitor.
x and X	Move up and down the x-axis of the world
y and Y	Move up and down the y-axis of the world.
z and Z	Move up and down the z-axis of the world.
up and down arrow	Move viewpoint backword and forward.
left and right arrow	Move viewpoint right and left.
Left mouse button and drag.	Rotate the viewpoint
o	Toggle the view to be/not be fixed.
Print Screen key	Writes current view to disk as out_???.rgb.

When you run this application (by typing "`example1`"), you should see this in the bottom left corner of your screen.

Figure 2. snapshot of the first example.

The file `example1.world` defines the world by summing up all the objects that are used in the example:

Source ii: `example1.world`**, the World Description File**

```
Simple Virtual Object File Format  version 1.0
number of objects: 2

object name: meadow
primitives file: plane.object
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
0 -1 -1 1
other attributes: 0
number of children: 0

object name: cube_text
primitives file: hello_world.object
transformation matrix:
1 0 0 0
0 1.1 0 0
0 0 1 0
0 0.5 -3 1
other attributes: 0
number of children: 0
```

The geometry of the objects is defined in separate files, so they can be used to describe more than one object in the environment:

Source iii: `hello_world.object`, **the Object Geometry File**

```
Simple Virtual Primitive File Format  version 1.1
bounding sphere 1.1 at 0.8 0.5 0.17
number of components: 3

component 1 type:  text
Data of component 1:
transformation matrix:
0.4 0 0 0
0 0.4 0 0
0 0 0.3 0
0.1 0.6 0.36 1
no of lines:
1
Hello World!

component 2 type: polyhedron
Data of component 2:
no of vertices:
8
vertices: x y z
0 0 0
0 1.0 0
1.6 1.0 0
1.6 0 0
0 0 0.35
0 1.0 0.35
1.6 1.0 0.35
1.6 0 0.35

no of faces:
6
faces: R G B #_of_vertices v1 v2 v3 ...
30 0 0 4 0 1 2 3
30 0 0 4 0 4 5 1
30 0 0 4 5 6 2 1
30 0 0 4 6 7 3 2
30 0 0 4 7 4 0 3
30 0 0 4 7 6 5 4
```

```
component 3 type: polyhedron
Data of component 3:
no of vertices:
10
vertices: x y z
0.4  0.4 0.37
0.5  0.5 0.37
0.8  0.6 0.37
1.1  0.5 0.37
1.2  0.4 0.37
1.2  0.2 0.37
1.1  0.1 0.37
0.8  0.0 0.37
0.5  0.1 0.37
0.4  0.2 0.37
no of faces:
1
faces: R G B #_of_vertices v1 v2 v3 ...
0 20 70 10 9 8 7 6 5 4 3 2 1 0
```

Certain configuration details, such as which directory or directories contain the object and world files of the application, where the window should be places, and with what dimensions, and the details about any hardware devices used by the application (such as tracking devices) are specified in the `.sve.init` file. This file is either contained in the directory from which the application is run, or, if none is there, in the home directory of the user. Here is an example `.sve.init` file which indicates where the object files should be found:

Source iv: `.sve.init`, the initial configuration file.

```
# This file contains some default variables. The read routine is not
# case sensitive.

DefaultObjectDirectory   ../../objects
DefaultMaterialDirectory ../../objects
```

1.2. On-line Help

The SVE manual (this document) is available on the World Wide Web, along with hypertext links for every function in the library. The URL for the SVE homepage, which contains a link to the latest version of the documentation, is "`http://www.cc.gatech.edu/gvu/virtual/SVE/SVE.html`".

2. SVE Basics

The control flow in a virtual environment and in any interactive application can be characterized by the following diagram:

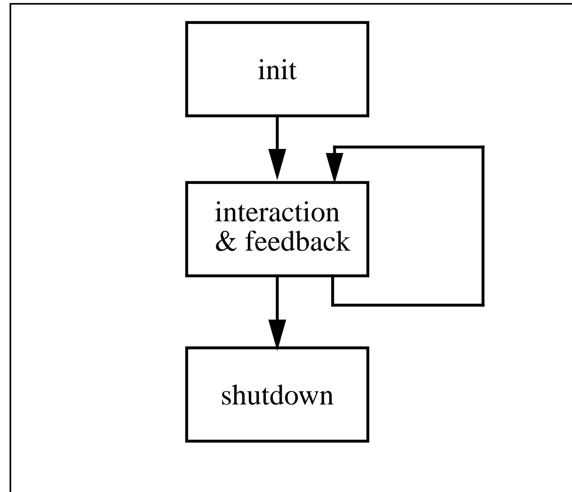


Figure 3. Global control flow.

This diagram is used to form the skeleton of the SVE library: the library provides functions for initializing the VR application, processing the input, giving feedback, and shutting down the application. We'll now discuss each of these parts.

2.1. Initialization of the Application.

```
void SVE_init(char * programname, SVE_config configuration,
              int *argc, char *argv[])
```

The `SVE_init()` function should be the first SVE function call used in an application, it allocates and initializes the various data structures SVE requires and opens a graphics window in the lower left corner of the screen[1]. The program name will appear in the title bar of the window. The `configuration` parameter specifies which graphics modes and hardware devices are used (lighting, texture mapping, using the HMD, etc.). Each additional option is represented by an integer (defined in *sve.h*), the combination of multiple devices and preferences is defined by combining the integers with a binary "or" operation. `SVE_init()` will then call the initialization routines of the specified hardware devices, and define callback-routines that define the default set of interactions for use with that configuration. The many available options are detailed in 3.1.1 "Configuration Flags" on page 23. A few of the more common options are briefly described below.

- *SVE_NORMAL*: The standard configuration (represented by 0). It does not make use of any special hardware (such as the glove or a tracker system), just the standard input devices (keyboard, mouse, etc.).
- *SVE_HMD*: Uses tracking devices to determine the location and orientation of objects in the environment, including the object representing the user's view. The particulars of which tracker devices to use to control which objects can be determined in an initialization file (see 3.1.2 "Initialization File" on page 28), or (less often) by the application explicitly. This configuration is actually a combination of *SVE_TRACKER*, which enables tracking devices, and *SVE_HMDMONO*, which produces a monoscopic view of the environment where the viewpoint moves as if rigidly attached to the tracker.

1. The window placement is standard for NTSC-conversion with help of the VIDI/O box. Other window placements are possible.

- *SVE_GLOVE*: Makes use of the CyberGlove. A graphical object will show the movements of the glove in the virtual screen and simple gesture recognizing can be enabled. As is the case with the tracking devices, the details of the glove device are given in the initialization file.
- *SVE_LIT_GOURAUD*: The world will be rendered with gouraud shading (flat shading is used by default) using lights defined in the world to shade the objects. This configuration is actually a combination of *SVE_LIGHTING* and *SVE_GOURAUD*.
- *SVE_STEREO*: The library will switch the monitor to stereo mode (if available), and render the left and right eye views, which produce a 3D view that can be seen using Crystal Eyes glasses.
- *SVE_TEXTURES*: Textured polygons will be rendered with their specified textures (these textured polygons are rendered as ordinary polyhedrons when this option is not used).

The configuration options can be changed by an application at any time. In addition, certain configurations come with a default behavior (event callback functions, default objects, etc.). The default behaviors can be overridden or augmented by application specific behaviors. This is discussed in more detail in the section, 3.1. "Application configuration" on page 23.

2.1.1 Loading an Environment.

`boolean SVE_loadWorld(filename)`

The `SVE_loadWorld()` function loads objects specified in an SVE world file into an internal data structure which is rendered automatically during the interaction loop. The function returns `FALSE` when an error occurs during loading. The file formats are described in detail in appendix B, page 111. The world description file describes a rooted tree of geometric objects, each described by an object file. For example, figure 4 shows a world consisting of a cube with text, a "meadow" ground plane, and two trees on the meadow. The two tree objects are linked to the meadow object, as they are "attached" to the ground. The

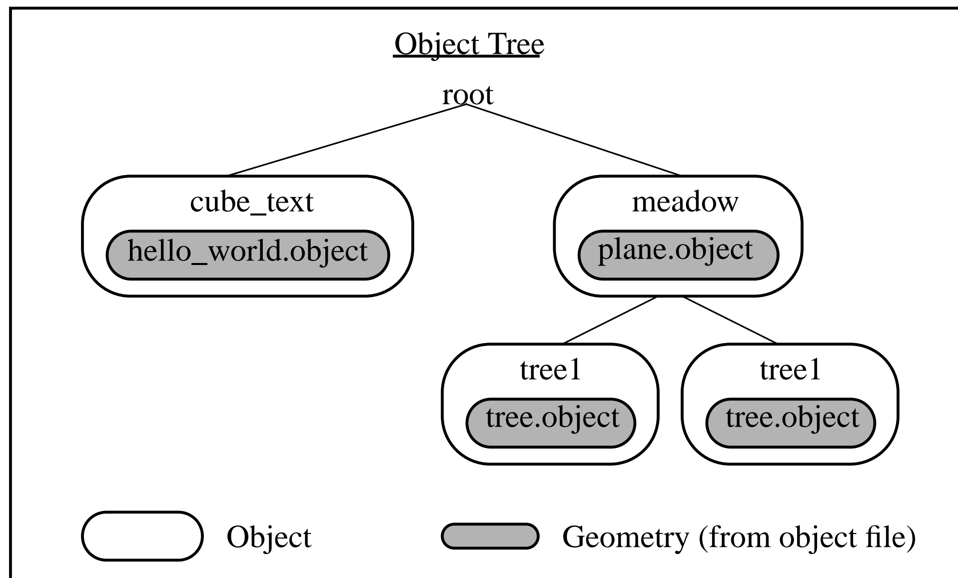


Figure 4. Example Object Tree

world and object files that describe this environment are shown in figure 5. The world file describes the object hierarchy (which objects are children of which objects), each object's position in terms of their parent's coordinate system (or world coordinate system for top level objects), and which object file to use for each object. The object file defines object geometry, which consists of many different types of primitives (polygons, lines, text, textured polygons, etc.).

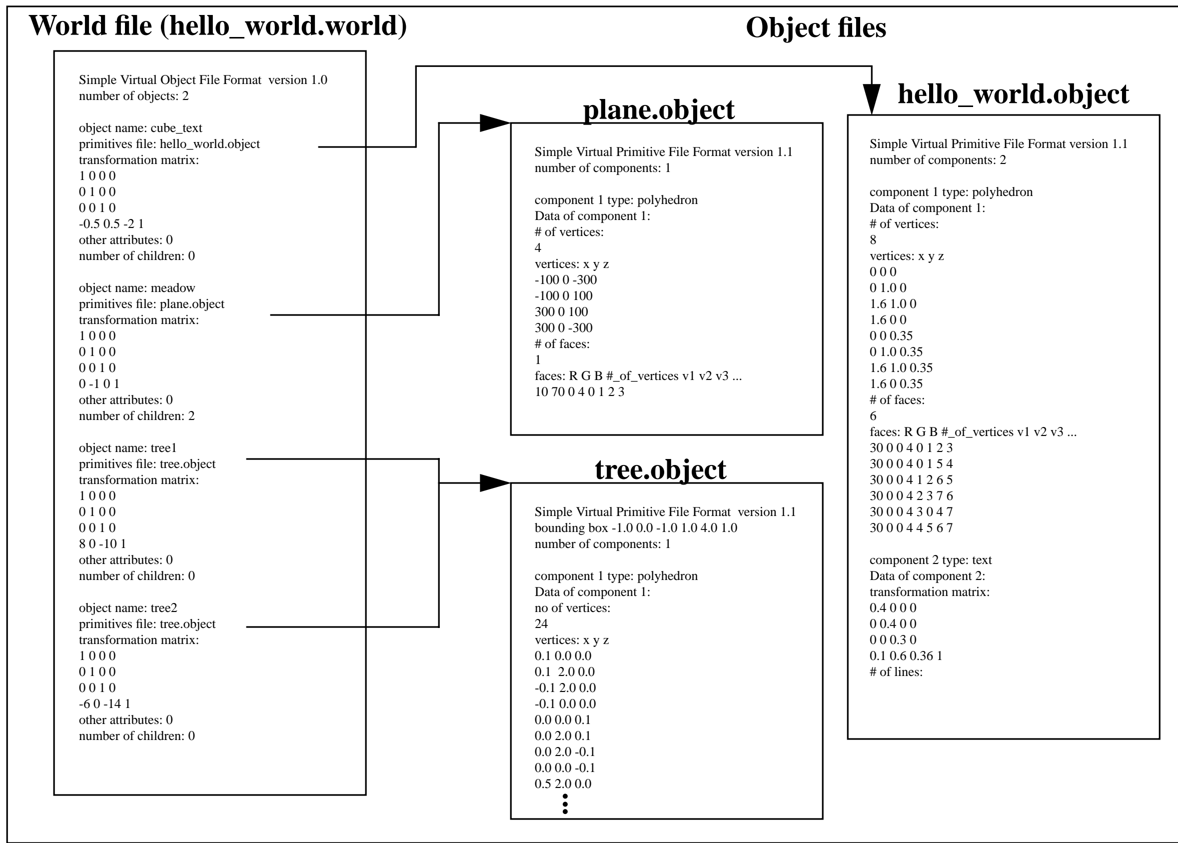


Figure 5. SVE data files

2.2. The Interaction Loop.

The interaction loop of the SVE system is shown below.

The polling of the input devices and the rendering of the world is done by the system. Functionality can be added both to the input handling and the rendering procedures. This loop is started by calling `SVE_beginEventLoop()`. The loop is stopped when `SVE_stopEventLoop()` or `SVE_abort()` is called, or (unless over-ridden by an application function) hitting the `ESC` key.

2.2.1 Input Handlers

The data from the input devices is automatically put in a global `SVE_state` data structure. The processing of events is done by a notifier-callback mechanism (used on most of the 2D direct manipulation systems like SUNVIEW, X,. etc.): the system notifies the application when an event occurs by calling the appropriate callback routine(s). (See "Concepts of a notifier mechanism and the event lookup table" on page 17)

Each event type has their own list of callback routines. Event types include keyboard input (`SVE_KEY_PRESS`, or specific key events such as `SVE_ESC_KEY`), mouse input (`SVE_LEFT_MOUSE`, `SVE_RIGHT_MOUSE`, etc.), object events (`SVE_OBJECT_SELECTION` and `SVE_OBJECT_HIGHLIGHT`), and more. A complete listing of available events can be found in `include/event.h` (in the SVE directory). Callback routines should be of the following form.

```
SVE_status function(SVE_state state)
```

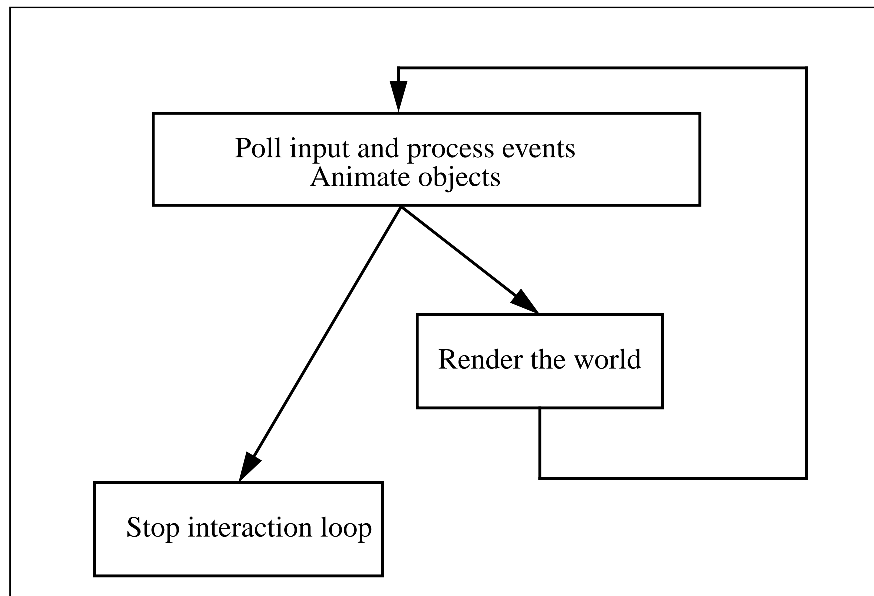


Figure 6. The interaction loop.

Each callback routine is added to the list of callbacks for a particular event type using the function `SVE_registerCallback()`, and removed from the list by calling `SVE_removeCallback()`. Event callback functions are called in the reverse order in which they were registered.

```
void SVE_registerCallback(int event, SVE_functionPtr function);
void SVE_removeCallback(int event, SVE_functionPtr function);
```

Note that the function should return a value of type `SVE_status` which has the possible values: `EVENT_IGNORED` and `EVENT_CONSUMED`. As already mentioned, some events in a specified configuration have a "default" callback routine. No callback routines have to be registered if the default behavior is sufficient. The default callback routines can be completely removed with the `SVE_removeAllCallbacks()` function.

```
void SVE_removeAllCallbacks(int event);
```

It is possible to have both user-defined and default callback routines on the list for the same event. In this case the user-defined routine will be called prior to the default one(s). If a user-defined routine returns `EVENT_CONSUMED`, the default callback (or any user-defined routines defined before this one) will not be called.

2.2.2 Animation Routines

An application may wish to alter continuously the properties of the objects maintained by the SVE system, or other data maintained by the application. An example of this would be a rotating cube. This is done by defining a routine that is called on a regular basis. This routine is known as an animation callback routine. The SVE system maintains a list of animation routines to call. These functions are called one after another after the pending events have been handled, and before anything is rendered. A new animation routine can be added to the front of the list using the function `SVE_addAnimationCallback()`.

```
void SVE_addAnimationCallback(SVE_functionPtr function);
```

The animation callback routine should be of the same form as an event callback routine, although in this case the return value of the function is ignored. An animation callback function can be removed from the list using the `SVE_removeAnimationCallback()` function. The entire list can be cleared with the `SVE_removeAllAnimationCallbacks()` function.

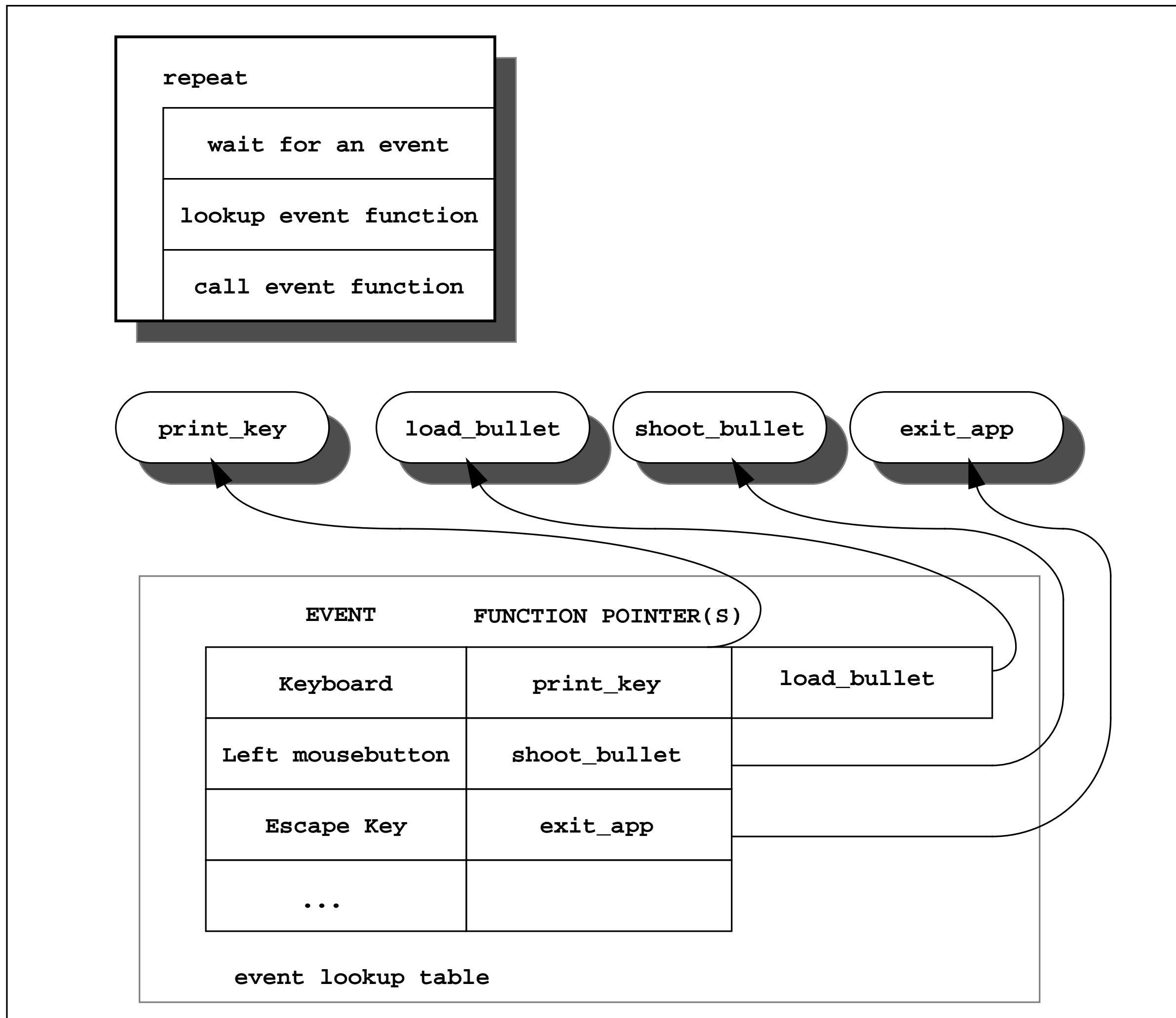


Figure 7. Concepts of a notifier mechanism and the event lookup table

```
void SVE_removeAnimationCallback(SVE_functionPtr function);
void SVE_removeAllAnimationCallbacks(void);
```

The animation callback functions are discussed in more detail in the section 3.4.2 "Animation Callbacks" on page 43.

2.2.3 Frame Drawing Routines

During the rendering process, the screen is cleared, the camera viewpoint is set, and the world is drawn (more will be said about this in section 3.4.3 "Rendering Callbacks" on page 45). Additional visual effects and other user-defined rendering functions can be added to the drawing process by registering a frame-callback routine. This routine will be called each frame, just before the world is rendered. and is added with `SVE_setFrameCallback()`.

```
void SVE_setFrameCallback(SVE_functionPtr function);
```

This function is added to the front of a list of frame callback routines. It is of the same form as the animation callback routines (its return value is also ignored). A frame callback can be removed from the

list with the SVE_removeFrameCallback() function. The entire list of frame callbacks can be removed using SVE_removeAllFrameCallbacks() the function.

```
void SVE_removeFrameCallback(SVE_functionPtr function);
void SVE_removeAllFrameCallbacks(void)
```

Note that any action taken in a frame callback which changes the viewpoint will not take effect until the frame after the one being drawn. Any object movements, therefore (which could move the viewpoint if the objects representing the user's eye point are part of the children sub-tree of the moving object), should be done in an *animation* callback routine.

2.3. Shutting Down.

```
void SVE_done(void)
```

The SVE_done() function deallocates the data structures and shuts down the I/O devices. After this function is called, SVE_init() must be called again before any other SVE functions can be called (with the exception on any function that can be called before SVE is initialized, such as SVE_setInitFilename()).

2.4. Summary and Another Example

In short, SVE provides a set of functions to initialize devices and rendering modes, regulate the events and object animations, and cleanly shut down a VE application. The interaction mechanism is based on event-callback functions, and objects that are kept in an internal database.

The total control flow of a typical SVE-based application can be described with the following diagram:

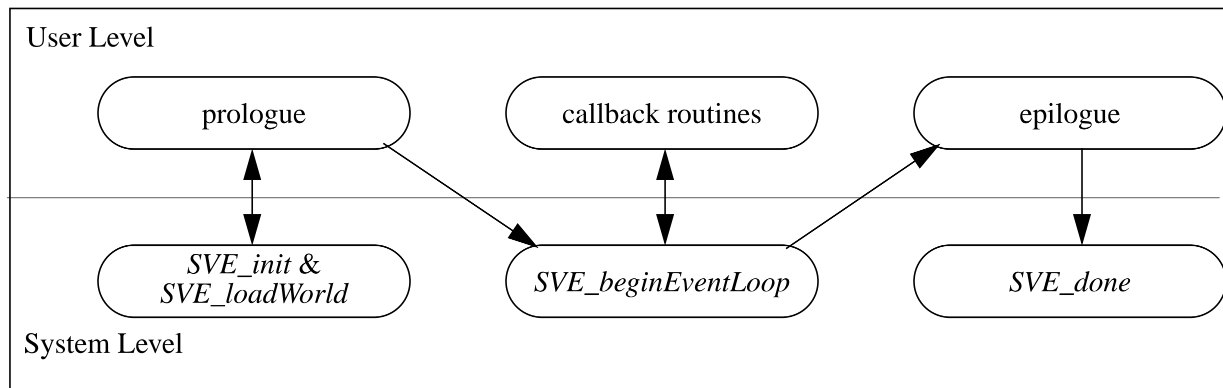


Figure 8. The control flow of the vr-application.

Here is another simple application, that includes a user-defined callback procedure:

Source v: example2, application using a callback

```
/************************************************************************
 * Example2 (sve module)
 *
 * This is the second example from the "SVE Basics" section of the
 * SVE manual.
 *
 * This example is used to show how to register a callback to SVE, and
 * handle this callback. The default key's as described in the Introduction
 * still work as they did in example1.
 ************************************************************************/

#include "sve.h"

/************************************************************************
```

```
 * This function handles the callback from the SVE_KEY_PRESS event.
 ************************************************************************/
SVE_status printKey(SVE_state state)
{
   char keyPressed;

   printf("printKey\n");
   if (state->eventType == SVE_KEY_PRESS) {
      keyPressed = ((SVE_keyEvent *)state->eventData)->keyVal;
      printf("Key %c pressed\n", keyPressed);
   }
   return(EVENT_IGNORED);
}


main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

/************************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the Initialization part of
 * the SVE Basics section of the manual for a description of the different
 * configurations you can use.
 ************************************************************************/
   printf("Starting application\n");
   SVE_init("Example2 (sve)", config, &argc, argv);



/************************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 ************************************************************************/
   if(!SVE_loadWorld("example2.world"))
   {
      printf("error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/************************************************************************
 * Tell SVE that we are interested in the SVE_KEY_PRESS event, and tell SVE which
 * function will handle this callback.
 ************************************************************************/
   printf("Registering an input callback.\n");
   SVE_registerCallback(SVE_KEY_PRESS, printKey);

/************************************************************************
 * SVE will take over control of the program until it is finnished.
 ************************************************************************/
   printf("Beginning event loop\n");
   SVE_setBackgroundColor(0.1, 0.1, 0.1);
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
}
```

One callback function is registered in this example for `SVE_KEY_PRESS` events, which echoes the keys pressed to `stdout`. Note that the callback function indicates that it has done nothing in response to the input by returning `EVENT_IGNORED`. This way, SVE's default function for handling keyboard input will always be called, allowing the user to still be able to use the keyboard to move around in the environment.

Apart from the basic functions already presented in this section, the library contains many useful routines to manage the virtual environment, The next chapter will present a more detailed and comprehensive description of SVE functions and data structures.

2.5. Tracker Devices and Other Configurations

So far, we have only presented applications that display an environment from any point of view in the environment. This display can be sent to a head mounted display or displayed as a stereo image on a monitor, but it only provides part of the requirements of an immersive virtual environment application. The item that is missing is a method to couple the point of view displayed to the actual point of view of the user. This is usually done through some sort of tracking device, such as an electromagnetic tracker.

The SVE library supports many tracking devices. Trackers are identified on a receiver to receiver basis. The details of what actual devices and receivers are used are given at run time by the initialization file, or, less often, specified by the application through a function call. Each tracker receiver provides the orientation and location of an object in the environment in reference to it's parent object. Since tracker receivers usually report its position in relationship to a reference point (the transmitter in the electromagnetic tracker example), the parent object can be thought of as determining the position of the reference point in the virtual environment. This concept is discussed in more detail in 3.6.2 "Rendered Object Tree (SVE_WORLD)" on page 58.

There are two requirements for an application to use a tracking device, it must identify that trackers are going to be used, and the particular tracker needs to be identified, usually in the initialization file.

If this application's `config` is changed so that it says "`config = SVE_NORMAL | SVE_HMD`"[1] instead of "`config = SVE_NORMAL`" it will use the trackers defined in the initialization file. This is usually done as a response to a command line argument. Here is a modified main function for the `example2` source code shown above which sets the `config` flag to use trackers when the command line includes a 't' (i.e. "`example2 t`").

```
main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

   printf("Starting application\n");
   if ((argc > 1) && (strchr(argv[1],'t') != NULL))
      config = config | SVE_HMD;

   SVE_init("Example2 (sve)", config, &argc, argv);

...

}
```

The second step is to specify what tracking devices to use to control which object. This is usually done in the initialization file, which is usually the `.sve.init` file in the directory from which the application is run. A tracker which is to control the application's viewpoint usually is associated with an object called "SVE HMD". A tracker which is to control a representation of the user's hand is usually associated with an object called "SVE cursor". Here is an example initialization file which associates the first and second tracker of an IsotrakII device with the "SVE HMD" and "SVE cursor" objects, respectively. The file also contains many other configuration details, which are discussed in more detail in 3.1.2 "Initialization File" on page 28. (For example, an alternative method to indicating that the application should use the tracking devices defined in the configuration file is to also include a "hmd true" line in the configuration file.)

1. Fish Tank VE set-ups are possible by using `SVE_TRACKER` instead of `SVE_HMD`. This results in the viewplane remaining rigidly attached to the tracking reference frame (for a stationary monitor), rather than moving the viewplane as if it were attached to the head tracker (for a head mounted display).

Source vi: .sve.init, an Example Initialization File

```
# This file contains some default variables. The read routine is not
# case sensitive.

minX                    0
minY                    0
sizeX                   640
sizeY                   480
Near                    0.01
Far                     5000.0
DefaultObjectDirectory   ../objects
DefaultMaterialDirectory ../objects
tracker 1               buckhead isotrakii /dev/ttyd2 1 SVE HMD
tracker 2               buckhead isotrakii /dev/ttyd2 2 SVE cursor
move USER to 0.0 2.2 0.0
```

As an example, the last line of the file would be translated as, "the second tracker receiver (`tracker 2`) is located on a machine called 'buckhead' (`buckhead`) and it is a Polhemus IsotrakII device (`isotrakii`) connected to the /dev/ttyd2 serial port (`/dev/ttyd2`). The receiver to use is the second one (`2`). This tracker receiver controls the 'SVE cursor' object (`SVE cursor`)."

The "tracker" line essentially introduces a new object into the object tree between the controlled object and its parent. The position of this new object (called "SVE tracker X", where X is the number ID given on the "tracker" line) is determined by the tracking device, and determines the controlled object's position because it is the controlled object's parent. Usually, the tracking device reports its position in relationship to a reference position (the electromagnetic transmitter, for example). Therefore, the controlled object's original parent object (now the parent of the "SVE tracker X" object) should be positioned to reflect the reference object's position. The "USER" object is the parent object of the "SVE HMD" and "SVE cursor" objects. The "move" line in the example initialization file places the tracking reference frame 2.2 meters above the floor. See "Object trees" on page 53 for more details.

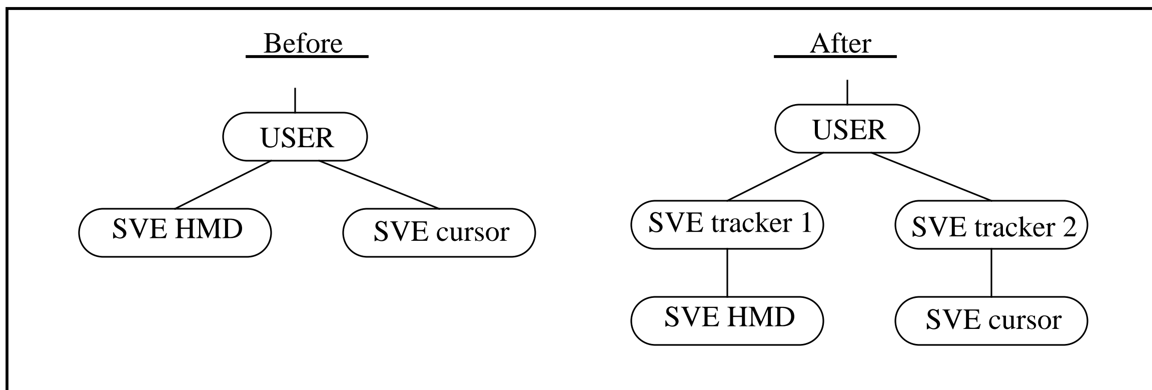


Figure 9. Objects Inserted From Initialization File in Source vi:

Lastly, in order for this to work, there must be a server program running on the machine to which the tracking device is connected. The server program communicates with the device, and sends the tracker information to all SVE applications which are interested in it. The server program is called `server-tracker`, and can be found in the SVE `bin` directory (`~vrgroup/sve/v2.0/bin`). More information on server programs can be found in 3.13. "Servers" on page 99.

3. Programming Details

This chapter describes the underlying concepts and structure of the SVE system and the many routines provided by the SVE library to integrate an application with the SVE system. The importance of each part of this chapter will depend greatly on the needs of the application.

3.1. Application configuration

The main service of the SVE library and system is providing a large number of possible configurations with minimal work from the application to deal with the configurations. There are two methods to setting and changing the configuration of the SVE system, through the configuration flags and through the initialization file. These methods are described below.

3.1.1 Configuration Flags

There are many display and interaction options available to an SVE application. A display can be flat shaded (the default) or gouraud shaded, black and white, wireframe, rendered with texture maps, and in stereo for CrystalEyes(TM) glasses or red/blue (for the right/left eye) stereo. The display can be pre-distorted for LEEP optic head mounted displays, or can have an inset that is at a higher resolution than the outer edges. An application can use a tracking mechanism and a CyberGlove(TM) hand input device. An application that requires selection-type interactions (object selection, button selection, etc.) can use a built-in selection object that follows the "SVE cursor" object (which is usually the second tracker when trackers are being used), and uses a ray to select when the mouse button is pressed (generating highlight and selection events).

Each option is independent of other options. The entire configuration is stored in a single bit-array, which is the "config" parameter passed to `SVE_init()`, the function that initializes the SVE system. Once the configuration is set, it can be retrieved with the function `SVE_getConfig()`, and changed with the function `SVE_changeConfig()`.

```
boolean    SVE_init(char *programName, SVE_config config,
                    int *argc, char *argv[]);
SVE_config SVE_getConfig();
void       SVE_changeConfig(SVE_config newConfig);
```

A particular option is set by doing a bit-wise "or" on the configuration flag variable using the appropriate option flag. For example, the following line sets the configuration flag to render with gouraud shading:

```
config = config | SVE_GOURAUD;
```

Table 2 gives a list of the available options and a short description of each. Longer descriptions of each option follows.

Table 2: SVE Configuration Options

Option flag	Option description
SVE_NORMAL	This is the standard vanilla display. The statement "`config = SVE_NORMAL;`" removes all other options.
SVE_LIGHTING	This option enables lighting effects. Note that if no light sources are defined, the environment will be quite dark.
SVE_GOURAUD	This option enables gouraud shading rendering, which is generally slower, but looks more realistic. With this option enabled, curved surfaces which are represented by many flat polygons will appear curved (except on the silhouette edges).

Table 2: SVE Configuration Options

Option flag	Option description
SVE_LIT_GOURAUD	This option is a combination of SVE_LIGHTING and SVE_GOURAUD. These options are often used in conjunction.
SVE_BW	This option causes SVE to render in black and white. This will be no faster than rendering in color, however.
SVE_WIREFRAME	Setting this option flag will cause SVE to render objects as a wireframe. This could result in a performance increase and reveals polygon edges.
SVE_TEXTURES	This option allows for textures that are defined for a geometric primitive to be displayed.
SVE_STEREO	With this option set, the SVE application will use the 120HZ refresh rate of a stereo monitor coupled with a StereoGraphics(TM) emitter to render in stereo for CrystalEyes(TM) glasses.
SVE_RBSTEREO	This option causes the display to be rendered for stereo viewing such that the left eye is rendered in blue, and the right eye is rendered in red. A stereo viewer such as the Fakespace BOOM2C (TM) will use these signals to show a black and white stereo view.
SVE_TRACKER	Setting this option enables the use of any 3D trackers that have been defined. The trackers to use are usually defined in the initialization file, but can be defined in the SVE application.
SVE_HMDMONO	This option causes the view plane to be set up for a head mounted display. The view plane will be attached to the object representing the user's head rather than the world reference point.
SVE_HMDSTEREO	This is the same as above except the eye objects are placed on either side of the user's center view (using the eye separation parameter). Only one eye is rendered, though. The default is the left eye, but can be changed using the SVE_RIGHTEYE configuration.
SVE_RIGHTEYE	The image drawn to the screen is from the point of view of the right eye, rather than the left eye.
SVE_HMD	This option is equivalent to SVE_TRACKER | SVE_HMDMONO.
SVE_GLOVE	Setting this option enables the use of a Virtual Technologies CyberGlove(TM).
SVE_SELECT	Setting this option causes the selection objects to appear, and allows other objects to be selected using the right mouse button.
SVE_PREDISTORT	This option causes the display to be pre-distorted in real time for leep-optics head mounted displays. This option requires a SGI Reality Engine.
SVE_NOAUDIO	Setting this flag prevents the application from producing audio, either locally or remotely through an audio server.
SVE_NODISPLAY	Runs application without a display.
SVE_SPATIALSOUND	This option enables the use of spatial sounds, which requires a spatialization host.

SVE_NORMAL

This is the vanilla configuration. Setting the configuration to this value turns off all other options and using `SVE_NORMAL` in a series of `or`'d configuration flags has no effect.

SVE_LIGHTING

With this configuration, defined light sources are used in conjunction with the material properties (diffuse and specular response to the light sources, as well as an ambient color) of geometries to produce a lit scene. This option requires more calculations for rendering, and therefore will result in a performance hit in the frame rate.

SVE_GOURAUD

If this option is used, the display will be rendered using gouraud shading, rather than the default flat shading. Gouraud shading calculates a color for each vertex of a polygon based on the vertex normal. For surfaces which are represented by many polygons, the vertex normals are perpendicular to the surface, rather than the polygon with which the vertex is a part. This results in a smooth shading that hides polygon edges, and gives the appearance of the surface. The modeler can specify the vertex color before hand, producing interesting effects where the polygon's color changes color smoothly from one vertex to the next.

Gouraud shading does result in a performance hit, and will cause the application to render less frames per second.

SVE_LIT_GOURAUD

This option is equivalent to SVE_LIGHTING | SVE_GOURAUD, which combines the effect of lighting with light sources defined in the scene and gouraud shading of the objects in the scene. This option is provided as a convenience, as lighting and gouraud shading are often used together.

SVE_BW

With this option on, the display will be rendered in black and white. This option will not cause any significant speed-up from a color rendering. It is mostly for displays that require black and white, as would be the case if a display only took one of the red, green, and blue display signals.

SVE_WIREFRAME

With this option on, the display will be rendered in wireframe. This means that all polygons will be represented by a line outline rather than a filled polygon. The lines are of the same color as the polygon would have been. This option negates the effect of the SVE_TEXTURES option, as texture maps are not rendered on polygons. A wire frame model is generally displayed faster than a shaded model.

SVE_TEXTURES

With this option on, polygons that have been specified to use a texture map will do so. Texture maps are part of an objects material definition. They are specified in a material file, or using the 'textured_polyhedron" primitive option in the object description file. Texture mapping is a costly operation, especially for low-end graphics engines. However, the detail that can be displayed using a texture map (as compared to a extremely detailed polygonal model) can out weigh the cost.

SVE_STEREO

This option displays a rendered frame for each eye, and switches the monitor to stereo mode (if available), which allows the screen to be viewed in stereo using shutter glasses that are sync'ed to the monitor. Essentially, the entire screen is used to display the left and right eye views, with one view on the top, and the other on the bottom. The stereo mode uses the entire display to show one after the other. An SVE application can expect a decrease in rendering speed, as a larger area is being rendered, and the view is rendered twice.

SVE_RBSTEREO

This option is similar to the SVE_STEREO option, except that the right and left eye views are rendered entirely in red and blue, respectively. A device, such as the Boom2C by FakeSpace™, can use the two outputs (red and blue) to construct a single color display in stereo (perhaps by showing the red signal to the right eye, and the blue signal to the left eye). As in the SVE_STEREO option, this is a costly option, as the scene is rendered twice, and usually to a larger area (full screen, in the case of the Boom2C).

SVE_TRACKER

This option enables the use of tracking devices. The particular tracking device must be specified in the initialization file (see below), or in the application itself.

SVE_HMDMONO

This option sets the viewing plane to be attached to the head (as it really is in a head mounted display). It also changes the default behavior for mouse (or FlitStik) button presses. The left mouse button is used to toggle the user's flying direction, the middle mouse button is used to fly in the direction the user is looking, and the right mouse button is used to accelerate.

SVE_HMDSTEREO

This option is the same as SVE_HMDMONO except that the user's two eyes are placed on either side of the center view (which is halfway between the user's eyes), using the eye separation parameter. Only one image is rendered. It is the left eye, unless the SVE_RIGHTEYE configuration flag is set.

SVE_RIGHTEYE

This option causes the image to be rendered from the point of view of the user's second (right) eye, rather than the left eye.

SVE_HMD

This option is equivalent to SVE_TRACKER | SVE_HMDMONO. It is provided for backward compatibility.

SVE_GLOVE

The option enable the use of a CyberGlove™ hand input device. The device must still be specified in the initialization file (see below), or in the application itself. See 3.11.2 "Hand Input Devices" on page 83 for more details on hand input devices.

SVE_SELECT

This option allows for a default object selection behavior. When this option is used, a standard pointer and selection object are loaded in and automatically follow the "SVE cursor" object (usually the user's hand). Pressing the left and middle mouse buttons cause a ray to shoot out from the pointer object. Any selectable object that is hit by the ray is highlighted (and an `SVE_OBJECT_HIGHLIGHT` event is generated). When the mouse button is released, the ray disappears, and the object highlighted when the button is release is "selected" (an `SVE_OBJECT_SELECTION` event occurs).

The default objects used for the pointer and ray are shown in Figure 18. "Widget Example Screen Shot." on page 91. The object used for each of these tasks can be changed in the initialization file (see below).

SVE_PREDISTORT

This option has not been tested with many other configurations, including insetting and stereo. It requires SGI GL routines specific for Reality Engine graphics, and therefore is not available for OpenGL implementations.

To control predistortion, users may set two externally declared variables, `SVE_pdExtX` and `SVE_pdExtY`. These control the horizontal and vertical extent of the source window, which contains the undistorted image. The smaller the source window, the faster the predistortion can be performed. The resolution of the destination window is controlled by the standard window globals `SVE_minX`, `SVE_minY`, `SVE_sizeX`, and `SVE_sizeY`. Note that large source extents are not really necessary HMDs with low resolution.

Users may also set several constants declared in `predistort.h`. These constants are based on the HMD viewing model contained in Robinett's paper (see Presence Vol 1 No 1). `PD_K` controls the strength of the contraction used to correct for HMD distortion. `K` models the strength of the stretch in the HMD's distortion. `T`, `B`, `R` and `L` are the normalized boundary coordinates of the undistorted graphics window, relative to the optical axis of the HMD's lens. The distance on the image from the axis to the top is `T`, to the bottom `B`, to the left `L`, and to the right `R`. The coordinates are normalized so that the longest distance is 1.

The predistortion is performed with the SGI's dynamic texturing hardware. The source image is treated as a texture. In the destination window, a point grid is precalculated and predistorted. The source image is then textured onto the predistorted grid. There is a trade-off between texel interpolation and grid precision (the number of grid points). This may be controlled with the constants `NUM_CELLSX` and `NUM_CELLSY` (internal variables).

Note that image generation and predistortion must be synchronized. If not, the predistorting routines will load incomplete frames as textures. In SVE, this is accomplished by used the same process for rendering and predistortion.

SVE_NOAUDIO

Setting this flag prevents the application from producing audio, either locally or remotely through an audio server.

SVE_NODISPLAY

Runs application without a display. The event-render cycle is still followed, so frame callbacks and object frame callbacks (if the object isn't culled) are still called, along with animation callbacks. Events can be obtained from an event server, and the tracker and audio servers can be used.

SVE_SPATIALSOUND

This option is no longer supported. It may be supported again in the future, however.

3.1.2 Initialization File

When an application calls `SVE_init()`, the SVE system first reads an initialization file that sets many of the attributes of an SVE application that do not change often, but change often enough that a re-compilation of the application is too costly. Example attributes that can be set are default directories for object, world, texture, and material files, which machines are running remote servers, how the SVE application display should appear, and what types of tracking devices should be used, and what objects they control. The default name of this file is *.sve.init*, and the file will be taken from the current directory, or your home directory if there is none in the current directory. The name of the file can be changed before SVE is initialized using the following function call:

```
void SVE_setInitFilename(char *initFilename);
```

Here is an example initialization file. Note that the '#' character denotes the beginning of a comment which runs to the end of the line, and that the file reader is case insensitive (except for file names, path names, and object names).

Source vii: Example Initialization File

```
# This file contains some default variables. The read routine is not
# case sensitive.

minX                    0
minY                    0
sizeX                   640
sizeY                   480
DisplayConfig           .sve.flighthelmet
Near                    0.01
Far                     5000.0
DefaultObjectDirectory  ../objects
DefaultMaterialDirectory ../objects
tracker 1               buckhead isotrakii /dev/ttyd2 1 SVE HMD
tracker 2               buckhead isotrakii /dev/ttyd2 2 SVE cursor
```

Each attribute in the initialization file occupies its own line. The file reader ignores white space and is case in-sensitive when reading attributes. A list of the available attributes follows. The parameters to the attributes are specified as a type followed by the parameter number. The type is one of 'f' for float, 'd' for integer, 's' for string, 'b' for a boolean string ("true", "false", "yes", "no", "t", or "f"), or 'a' for a coordinate axis ("x", "y", or "z").

Table 3: Initialization File Commands

Keyword	Parameters	Parameter description
defaultObjectDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used when objects can not be found where they are supposed to be.
defaultWorldDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used when a world file can not be found where it is supposed to be.

Table 3: Initialization File Commands

Keyword	Parameters	Parameter description
defaultTextureDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used when a texture file can not be found where it is supposed to be.
defaultTextureMap	s1	s1 = The file name of the texture that will be used on a textured polyhedron which has no texture defined for it.
defaultMaterialDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used when a material file can not be found where it is supposed to be.
defaultConfigDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used to find other configuration files.
defaultAudioDirectory	s1	s1 = The directory (or list of directories separated by colons) which will be used to find audio files specified by the application.
eventServer	s1	s1 = The address of the machine running the event-server for the application.
audioServer	s1	s1 = The address of the machine running the audio-server for the application.
worldServer	s1	s1 = The address of the machine running the world-server for the application. Applications that connect to the same world-server will share a set of SVE objects which contains the SVE objects of each individual application. The world-server is currently under development, and is not guaranteed to be stable.
VRmachine	s1	s1 = An address that will be used to set the eventServer and audioServer values if none is given in the initialization file.
defaultServerDirectory	s1	s1 = The directory (NOT a directory list) in which SVE should look for server programs if it needs to start one automatically (presently done only for tracker servers).
tracker	d1 s2 s3 s4 d5 s6	This command initializes a tracker which will determine the position and orientation of an object in the SVE application. d1 = Unique identifier (any integer < 32 not used by another tracker) s2 = The address of the machine to which the tracker is connected. A tracker-server must be running on that machine. s3 = The type of tracker. Examples of this are 'IsotrakII' and 'Bird'. s4 = The serial port identifier. An example of this is '/dev/ttyd1'. d5 = The receiver number of the tracker device. Usually >= 1. s6 = The name of the object which will follow the tracker. Note that this name can contain spaces, but ends at the end-of-line. An example of this would be 'SVE HMD' or 'SVE cursor'.
trackerHemi	d1 f2 f3 f4	This command defines the hemisphere of a tracking device, which is only useful for devices that require it (electromagnetic). d1 = Tracker identifier given in the 'tracker' command. f2, f3, f4 = (X, Y, Z) vector defining the center of the hemisphere (where positive Y is up.)
eyePosition	f1 f2 f3	f1, f2, f3 = (X, Y, Z) position of the display in relation to the 'SVE HMD' object, which usually follows the head tracking receiver. This value allows correction for the fact that the head tracking receiver is often above the head rather than at the eye's position.

Table 3: Initialization File Commands

Keyword	Parameters	Parameter description
eyeSeparation	f1	f1 = The distance (in meters) between the objects representing the user's eyes for stereo viewing. Note that this is not necessarily the actual distance between the user's eyes, as the technology may require different values for the two images to be fused by the user's eyes into a stereo view.
move	s1 to f2 f3 f4	s1 = SVE object to move. (Note that the "move", "rotate" and "scale" attributes are performed in the order given in the file, and each may appear one or more times. Generally, if these attributes are ordered differently, a different resulting position for the object will occur.) f2, f3, f4 = (X, Y, Z) vector defining how far to move the object in the X, Y, and Z directions respectively. The distances are in terms of the object's parent's coordinate system.
rotate	s1 by f2 around a3	s1 = SVE object to rotate. f2 = Degrees to rotate the given object. a3 = Axis around which the object should be rotated. Rotations are given in terms of the object's parent's coordinate system.
scale	s1 by f2 uniformly (or) s1 by f2 along a3	s1 = SVE object to scale uniformly or along a given axis. f2 = Scale factor used to scale the object in terms of its parent's coordinate system. a3 = Axis (in parent's coordinate system) along which the scale factor should be applied.
minX	d1	d1 = Bottom left corner X of the SVE application window on the display.
sizeX	d1	d1 = Width of the SVE application on the display.
minY	d1	d1 = Bottom left corner Y of the SVE application window on the display.
sizeY	d1	d1 = Height of the SVE application on the display.
DisplayConfig	s1	s1 = Filename of the configuration file for the display. The display configuration file can contain a field of view/aspect ratio description or a screen dimension, position and rotation description. It will override the "VofY" and "aspectRatio" lines in this configuration file.
VofY	d1	d1 = Field of view in Y direction in tenths of degrees.
aspectRatio	f1	f1 = Aspect ratio of the width (in X) over the height (in Y) to determine the field of view in the X direction.
near	f1	f1 = Distance to the near clipping plane.
far	f1	f1 = Distance to the far clipping plane.
FPSupperLimit	f1	f1 = The SVE application will not be rendered at a faster frame rate than this.
FPSlowerLimit	f1	This option has no effect as of yet.
audio	b1	b1 = Boolean value to determine if the application will produce any audio output, either locally or remotely through an audio server. This value overrides the SVE_NOAUDIO option flag of the 'config' parameter to `SVE_init()`.

Table 3: Initialization File Commands

Keyword	Parameters	Parameter description
hmd	b1	b1 = Boolean value to determine if the trackers will be used. This value overrides the SVE_HMD option flag of the 'config' parameter to SVE_init().
gouraud	b1	b1 = Boolean value to determine if the display will be gouraud shaded or not. This value overrides the SVE_LIT_GOURAUD option flag of the 'config' parameter to SVE_init().
pointerObjectName	s1	s1 = Name of an object which should be used instead of the regular pointer (which is the object that follows the 'SVE cursor' object when the SVE_SELECT option flag is set).
pointerObjectFile	s1	s1 = File name of the object description which should be used instead of the default pointer description. (The pointer is the object that follows the 'SVE cursor' object when the SVE_SELECT option flag is set.)
selectorObjectName	s1	s1 = Name of the object which should be used instead of the "ray" selector object (which "shoots" from the pointer object), when the SVE_SELECT option flag is set.
selectorObjectFile	s1	s1 = File name of the object description which should be used for the selector object if it is different from the default ray. This object is used for selection when the SVE_SELECT option flag is set.
glove	d1 s2 s3 s4	This command initializes the CyberGlove hand input device. d1 = Unique identifier (any integer <32 not used by another glove device). s2 = The address of the machine to which the glove is connected. Currently, remote glove servers are not supported. s3 = The serial port identifier. An example of this is '/dev/ttyd1'. s4 = The name of the object to which the graphical representation of the hand will be attached. Note that this name can contain spaces, but ends at the end-of-line. An example of this would be 'SVE cursor'.
showFrameRate	b1	b1 = Boolean value to determine if the frame rate of the application is reported to standard output, or not.
verbose	b1	b1 = Boolean value to determine if normal output from the SVE library should be allowed (the default, TRUE), or suppressed (FALSE).
debug	b1	b1 = Boolean value to determine if debugging output from the SVE library should be allowed (TRUE), or suppressed (the default, FALSE).

3.1.3 Display Configuration Files

The display configuration can be described in a separate file that is referred to in an initialization file using a "DisplayConfig" line. The display configuration can describe a head mounted display, which is a certain distance from the eye in the negative Z direction, and has a certain field of view and aspect ratio. The display could also be described using dimensions, position and rotation, as might be done for a monitor. In this case, the view plane begins as being parallel to the X-Y plane. The positioning of the display in the virtual environment is critical to operating a "fish tank" set-up, where trackers are used to view a 3D display on a monitor.

The following table describes the parameters allowed in a display configuration file.

Table 4: Display Configuration Parameters

Keyword	Parameters	Parameter Description
ViewPlanePosition	f1 f2 f3	f1, f2, f3 = (X, Y, Z) Position of the origin of the view plane. (usually the center of the screen.)
ViewPlaneRotation	f1 f2 f3	f1, f2, f3 = (X, Y, Z) Rotations around the X, Y, and Z axis applied to the view-plane (in that order).
ViewPlaneMinX	f1	f1 = Distance in meters from the view plane origin to the left of the window.
ViewPlaneMinY	f1	f1 = Distance in meters from the viewplane origin to the bottom of the window.
ViewPlaneMaxX	f1	f1 = Distance in meters from the view plane origin to the right of the window.
ViewPlaneMaxY	f1	f1 = Distance in meters from the view plane origin to the top of the window.
fovY	d1	d1 = Field of view in Y direction in tenths of degrees.
AspectRatio	f1	f1 = Aspect ratio of the width (in X) over the height (in Y) to determine the field of view in the X direction.
eyePosition	f1 f2 f3	f1, f2, f3 = (X, Y, Z) position of the display in relation to the "SVE HMD" object, which usually follows the head tracking receiver. This value allows correction for the fact that the head tracking receiver is often above the head rather than at the eye's position. Note that the eye position and the view plane position should not be the same (as this would result in an undefined view direction). To avoid this problem, be sure to position the view plane a significant distance from the eye position (generally in the negative Z direction).
eyeSeparation	f1	f1 = The distance (in meters) between the objects representing the user's eyes for stereo viewing. Note that this is not necessarily the actual distance between the user's eyes, as the technology may require different values for the two images to be fused by the user's eyes into a stereo view.

3.1.4 Directories

The SVE library uses various default directories to look for files when they are not found in the current directory, or, if the file is specified with a complete path, in the directory given with the file. These default directories can be changed at any time with the appropriate function of the ones below.

```
void SVE_setDefaultObjectDirectory(char *directory);
void SVE_setDefaultWorldDirectory(char *directory);
void SVE_setDefaultMaterialDirectory(char *directory);
void SVE_setDefaultTextureDirectory(char *directory);
```

The given directory can also be a list of path names, each separated by a colon (as is done for UNIX path name lists). In the case of multiple path names, the directories will be searched one by one in order for the desired file.

3.2. The SVE System

The SVE library encompasses what is really a system which handles the events and movements that occur and the maintenance and display of the model associated with the application. This is all done in a particular context, which includes the group of objects that make up the model, the identification of the user and viewer of the model, and other characteristics of the system at any one time. The following sections describe the particulars of the context of the system, and the architecture of the system.

3.2.1 World State

The context of the SVE system at any one time is stored in a structure of type `SVE_state`. It is passed to every callback function called by the SVE library to return control to the application. At all other times (after the `SVE_init()` function has been called), it can be retrieved using the `SVE_getWorldState()` function. The state can be changed using the `SVE_setWorldState()` function, which returns the old state which has been replaced by the state given.

```
SVE_state       SVE_getWorldState();
SVE_state       SVE_setWorldState(SVE_state newState);
```

The following table describes the fields of the `SVE_state` structure which may be useful for an application's routine. Later sections also describe the use of certain fields in the given context.

Table 5: SVE State Fields

Field	Type	Description
programName	char *	The name of the application given in the `SVE_init()` call. This is the name used as the title of the application's window.
objectTree	list	This is the root of the object tree which is rendered each frame. It is a collection of SVE object arranged in a tree structure. It is described in a later section.
viewingObject	SVE_object	This refers to the object from which the point of view is determined for each frame rendered. The point of view is the origin of this object, looking toward the "SVE view plane" object (a line from the origin of the viewingObject to the origin of the "SVE view plane" will go through the center of the screen).
viewingObject2	SVE_object	This is the second view for stereo applications. (Usually, the right eye.)
viewingMatrix	M_matrix	This is the viewing transformation placed on the modeling transformation stack which allows objects to be rendered from the point of view of the viewing object (looking at the view plane object) rather than from the world origin. This matrix is updated just before any rendering is performed.
PespectiveMatrix	M_matrix	This is the current perspective matrix, which is updated just before any rendering is performed.
currentEye	char	This value indicates which eye is being rendered, the left (or SVE_LEFT_EYE, which is the default for monoscopic displays) or the right (or SVE_RIGHT_EYE).
originObject	SVE_object	This refers to the object that represents the user's coordinate system, where the origin of the coordinate system on the "floor" at the user's "feet." This configuration is true only if trackers are being used, and the reference point of the trackers (represented by `userObject`) is correctly positioned in the virtual world (i.e. the distance between the spot below the user's feet and the tracker reference point is accurately reflected in the position of the `userObject`). Otherwise, reasonable default positions are used. The user can "fly" around the environment by moving and rotating the `originObject` appropriately, which moves the user's coordinate system around the environment.

Table 5: SVE State Fields

Field	Type	Description
userObject	SVE_object	This refers to the object which is considered to represent the user in the model, or, more precisely, represents the reference frame for tracking devices, if any, that determine the position and orientation of the user's head, hand, or other, tracked parts.
hmdObject	SVE_object	This refers to the object representing the user's head, or more precisely the top of the user's head, in the model. A tracking device which tracks the user's head usually changes the position of this object.
cursorObject	SVE_object	This refers to the object representing the user's hand or wrist in the model. A tracking device which tracks the user's hand usually changes the position of this object.
viewPlaneObject	SVE_object	This refers to an object whose position and orientation determines the position and orientation of the viewplane used to calculate the user's current perspective (from one of the viewing objects).
origin	SVE_point	This location determines the location of the user. Changes to this location will change the user object's location.
eventType	SVE_eventType	This is the type of the last event to occur.
eventData	void *	This pointer references a data structure describing the last event to occur. Based on the `eventType` value (above), this pointer can be cast appropriately to obtain the details of the event that occurred.
ntscOn	boolean	This field indicates if the monitor is in NTSC mode.
config	int	This field contains the configuration flags described above.
flightSpeed	float	This value is the speed (in m/s) which determines the distance travelled each time the user is flown using the SVE fly command or flying routines.
beginTime	struct timeVal *	This is the time at which the `SVE_init()` function was called.
frameTime	struct timeVal *	This is the time the current frame rendering was begun.
lastFrameTime	struct timeVal *	This is the time when the frame before the current frame was begun.
framesPerSecond	float	This is 1/(the time between the current and last frame renderings).

3.2.2 System Overview

Currently, the SVE system operates as a single process (with separate processes for interfacing with tracking devices and audio output), which resembles the standard event loop structure of user interface applications. Figure 10. "SVE System Overview" on page 35 shows the steps through which the system goes to maintain the model of the application which defines the environment which is displayed. The thick lined boxes are steps which the SVE system handles automatically. The other boxes are steps where the SVE system allows for routines defined by the application to be executed.

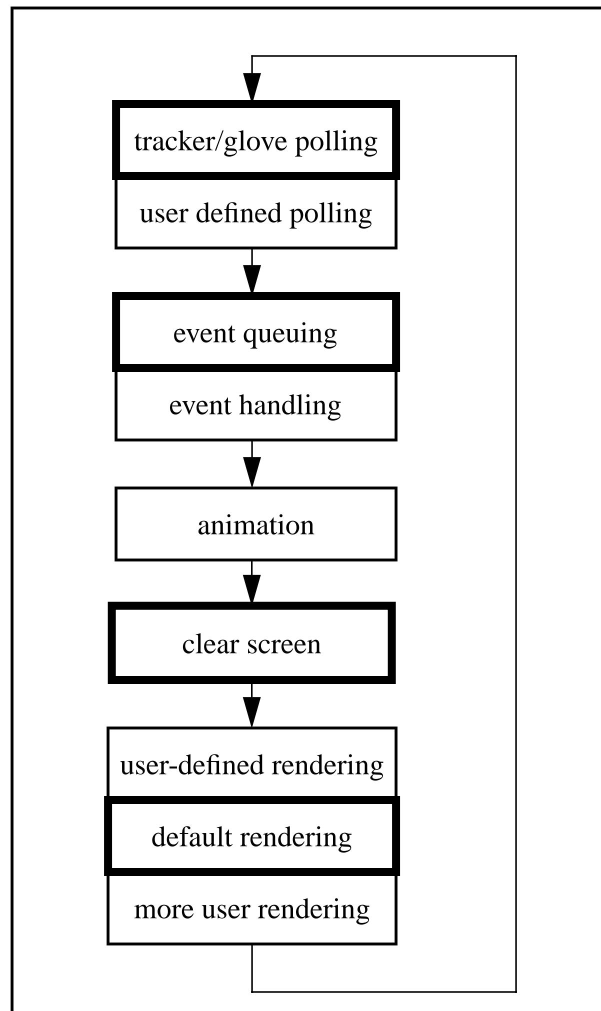


Figure 10. SVE System Overview

3.3. Events

3.3.1 Responding to Events

An SVE application responds to input devices using callback functions that are called by the event handler of the SVE system. Events can occur as a result of key presses, mouse clicks, hand gesture recognition, or object manipulation. In order to support relevant events reported by the GL library or by the X library (depending on whether the SVE application is written for GL or OpenGL) and event specific to SVE, the SVE library defines its own list of event types. For each event type, there exists a companion structure that includes the details of the event. For example, the structure associated with a key press event includes the key which was presses, and the structure associated with a right mouse button click includes the state of the button (pressed or released). The SVE system maintains a stack of callback functions for each event type. An SVE application enters a callback function onto the stack by calling `SVE_registerCallback()`.

```
void SVE_registerCallback(int event, SVE_functionPtr function);
```

The callback function, "function", should be of this form:

```
SVE_status function(SVE_state state)
```

The function should return one of two values, `EVENT_CONSUMED` or `EVENT_IGNORED`. The return value indicates to the event handled whether the event has been taken care of, or not. If the function returns `EVENT_CONSUMED`, the event handler will not call any other functions further down the event callback stack for the event type. If the function returns `EVENT_IGNORED`, the event handler will call the next function down the stack. The affect of registering a callback function which returns `EVENT_CONSUMED` is to prevent all other callback functions previously registered to respond to the particular event from being called. (Note that if no value is returned, `EVENT_CONSUMED` is assumed.)

The event type and associated structure are contained in the "state" structure which is passed to the callback function. The event type is `state->eventType`, the associated structure is reference by the `state->eventData` pointer. To examine the structure, the pointer must first be cast to the appropriate type, which is dependent on the event type. For example, a keyboard event (`SVE_KEY_PRESS`) has an event data structure of type `SVE_keyEvent`. The following event callback shows how the key pressed can be printed to standard output:

```
SVE_status handleKey(SVE_state state)
{
   char key;

   if (state->eventType == SVE_KEY_PRESS) {
      key = ((SVE_keyEvent *)state->eventData)->keyVal;
      printf("Key pressed: %c\n", key);
   }
}
```

It is important to note that event response is independent of the display rendering in the sense that many events can occur between display renderings. A series of events that require visual feedback for each event should do so using an animation callback routine which uses an event history generated by the event callback routine, or another similar technique.

A particular event callback function can be removed from the stack using the function `SVE_removeCallback()`. All of the callback functions for a particular event (including ones set by default by the SVE system) can be removed with the function `SVE_removeAllCallbacks()`. The function `SVE_getEventCallback()` returns the complete list of callback functions for a particular event as a linked list. The list is used as a stack, which means that the first function on the list is the last put on the list, and the first to be called.

```
void SVE_removeCallback(int event, SVE_functionPtr function);
void SVE_removeAllCallbacks(int event);
list SVE_getEventCallback(int event)
```

The entire list of event types that the SVE library recognizes can be found in the "*event.h*" file in the SVE include directory. A selected list of event type, along with their default behavior, can be found in Table 6. Descriptions of the event data structure types are given in Table 7. Note that two event types are provided for events generated by the windowing system (GL or X), or generated by the application through the windowing system. For these events, the event data structures contain the information provided by the native windowing system event queue.

Table 6: Selected Event Types

Event type (Structure Type)	Description	Default behavior
SVE_KEY_PRESS (SVE_keyEvent)	Key presses. Structure contains ASCII value for key pressed.	Default key press responses ('q' quits; 'x', 'y', and 'z' move in the positive x, y, and z direction, etc.).
SVE_ESC_KEY (SVE_stateChangeEvent)	Escape key pressed or released.	Quit the application
SVE_LEFT_MOUSE (SVE_stateChangeEvent)	Left mouse pressed or released	Using the trackers (SVE_HMD): reverse flying direction. Not using the trackers: start/end orientation change.
SVE_MIDDLE_MOUSE (SVE_stateChangeEvent)	Middle mouse pressed or released.	Not using the trackers: start/end orientation change.
SVE_RIGHT_MOUSE (SVE_stateChangeEvent)	Right mouse pressed or released.	Using the trackers (SVE_HMD): increase flying speed. Using the selection pointer (SVE_SELECT): cast pointer ray and test for interaction with a selectable object.
SVE_LEFTARROW_KEY (SVE_stateChangeEvent)	Left arrow key pressed or released.	Move viewer left.
SVE_RIGHTARROW_KEY (SVE_stateChangeEvent)	Right arrow key pressed or released.	Move viewer right.
SVE_UPARROW_KEY (SVE_stateChangeEvent)	Up arrow key pressed or released.	Move viewer forward.
SVE_DOWNARROW_KEY (SVE_stateChangeEvent)	Down arrow key pressed or released.	Move viewer backward.
SVE_PRINTSCREEN_KEY (SVE_stateChangeEvent)	Print screen key pressed or released.	Dumps the current view to the current disk directory in a file named out_???.rgb, where ??? is a unique number.
SVE_LEFT_MOUSE_DRAG (SVE_stateChangeEvent)	Event generated once between display renderings while the left mouse button is down and changes position.	Not using the trackers: change viewer orientation so that the display moves with the mouse.
SVE_MIDDLE_MOUSE_DRAG (SVE_stateChangeEvent)	As SVE_LEFT_MOUSE_DRAG, but generated while the middle mouse button is down.	Not using the trackers: fly viewer forward (or backward) and change the viewer's orientation such that the display moves in the opposite direction from the mouse, and does so at a rate proportional to the distance the mouse pointer is from its original position.

Table 6: Selected Event Types

Event type (Structure Type)	Description	Default behavior
SVE_RIGHT_MOUSE_DRAG (SVE_stateChangeEvent)	As SVE_LEFT_MOUSE_DRAG, but generated while the right mouse button is down.	Using the selection pointer (SVE_SELECT): change the viewer's orientation as SVE_LEFT_MOUSE_DRAG and test for intersection of the pointer ray with a selectable object.
SVE_OBJECT_HIGHLIGHT (SVE_objectEvent)	Generated when an object is being highlighted. The object is given in the SVE_objectEvent structure. Its "highlight" field indicates if the object has been highlighted or un-highlighted. When SVE_SELECT is part of the configuration, the object highlighted is the object being pointed at, but not yet selected.	None.
SVE_OBJECT_SELECTION (SVE_objectEvent)	Generated when an object is selected. The object is given in the SVE_objectEvent structure. When SVE_SELECT is part of the configuration, the object selected is the one the pointer ray is on when the right mouse button is released.	None
SVE_GESTURE (SVE_gestureEvent)	Generated when a hand gesture is recognized. The value given in the SVE_gestureEvent structure is the gesture identifier.	None
SVE_STYLUS_BUTTON (SVE_stateChangeEvent)	Generated when the button on tracker stylus is pressed or released.	None

Table 7: Event Data Structure descriptions

Event Structure	Event types	Fields
SVE_keyEvent	SVE_KEY_PRESS SVE_KEY_RELEASE	short int keyVal - ASCII value of key.
SVE_stateChangeEvent	Specific key states (SVE_A_KEY, SVE_LEFTARROW_KEY, SVE_ESC_KEY, etc.) and mouse button states (SVE_LEFT_MOUSE, SVE_LEFT_MOUSE_DRAG, etc.). All event types X that cause SVE_IS_PRESS_EVENT(X) to return TRUE.	boolean pressed - State of key or button.
SVE_mouseEvent	SVE_MOUSE_DOWN SVE_MOUSE_UP SVE_MOUSE_MOTION SVE_MOUSE_ENTER SVE_MOUSE_LEAVE All event types X that cause SVE_IS_MOUSE_EVENT(X) to return TRUE.	long int mouseButton - Bit mask of buttons pressed. See *event.h*. long int xPos - X position of mouse on screen. long int yPos - Y position of mouse on screen. long int xWinPos - X position of mouse in window. long int yWinPos - Y position of mouse in window.
SVE_objectEvent	SVE_OBJECT_SELECTION SVE_OBJECT_HIGHLIGHT All event types X that cause SVE_IS_OBJECT_EVENT(X) to return TRUE.	SVE_object object - SVE object involve in event.
SVE_gestureEvent	SVE_GESTURE	int gestureVal - Gesture ID value.
SVE_SGIEvent	SVE_OTHER_SGI_EVENT	long int eventType - GL event type. short int eventVal - GL event value.
SVE_X11Event	SVE_OTHER_X11_EVENT	Display *display - X display. Window window - OpenGL window. XEvent *eventData; - X event data.

3.3.2 Generating Events

As the SVE event handler uses an event queue (usually the one that is part of the native windowing system), generating events as if they actually occurred can be done with the following function.

```
void SVE_enterEvent(SVE_eventType eventType, void *eventData);
```

The event will be processed after all pending events. If the application enters an event, it should be sure to accompany it with a pointer to the appropriate event data structure. User defined events can be entered by using the `SVE_USER_EVENT event type`. The `eventData` pointer given will be passed to the event callback function in the `eventData` field of the given state structure.

As a convenience, functions are provided for object highlighting and object selection. An object can be "selected" (generating the SVE_OBJECT_SELECTION event) using the `SVE_selectObject()` function call.

```
void SVE_selectObject(SVE_object object);
```

An object can be "highlighted" and "un-highlighted" (generating the OBJECT_HIGHLIGHT event and causing the object in question to be rendered "highlighted" if it has a highlight color or highlight geometry) using the `SVE_highlightObject()` function call.

```
void SVE_highlightObject(SVE_object object, boolean highlight);
```

3.4. Animation and Rendering

Animation and rendering are placed in the same section not only because they are handled in a similar way, but because they are different, and it is important to recognize the difference between them. Animation implies a change in the state of the world, while rendering implies simply a view of the world. The SVE system allows for an application to do its own animation and rendering above and beyond what the system does automatically. If the application makes the mistake of doing animation-type actions when the system expects it to be doing solely rendering actions, the application will likely run into problems. It is important, therefore, to change the state of the world only when it is time to do animation, and render only when it is time to do rendering. Of course, using the automatic animation facilities provided by the SVE library will avoid these problems.

An application can specify a simple object animation through the use of the automatic animation routines presented in section 3.4.1. More complex animations and user defined rendering are handled through the use of callback function routines, similar to the callback functions used for event response. The SVE system maintains a list of animation functions (see section 3.4.2), a list of frame callback routines, and a list of object frame callback routines for each object (both described in section 3.4.3).

3.4.1 Animation Routines

SVE provides several routines to make simple object animations easier. These functions allow the programmer to set up values for velocity, acceleration, and rotation for each object. User defined data may also be added (more on that in a little bit). The object animation structure is defined as follows, although these values are usually set using the `SVE_setAnimationVar()` function described below:

```
typedef struct
{
   double  xvel, yvel, zvel,      /* Current x, y, and z velocity (in m/s) */
           xacc, yacc, zacc,       /* x, y, and z acceleration */
           xrot, yrot, zrot;       /* x, y, and z rotation (in degs/sec) */
   double  lasttime;              /* Last time the object was animated */
   SVE_animateObjectFunctionPtr UserFunc;
                                 /* User defined animation function */
   boolean animate;               /* True, if the object is to be animated. */
   void    *userdata;             /* User defined data. */
} AnimationStruct;
```

Automatic animation is initialized and begun with a call to `SVE_initAnimation()`. This function must be called before entering the SVE event loop and before any of the other animation functions described in this section can be used.

Each object has an animation flag, defined in the `SVE_object` structure, that lets SVE know whether or not that particular object is to be animated. By default, this flag is set to FALSE (indicating that the object is not to be animated) when an object is created or loaded. To allow SVE to animate the object, you must make a call to `SVE_setObjectAnimation()` and pass it the `SVE_object` structure pointer for the object you wish to animate and a value of TRUE. To turn the animation of the object off again, just call `SVE_setObjectAnimation(obj, FALSE)`.

A call to `SVE_setObjectAnimation()` alone does not automatically make the object move. Every object has nine animation variables that correspond to velocity, acceleration, and rotation (each having a separate component for X, Y, and Z). These values are all set to zero by default. You can make changes to these variables by calling the `SVE_setAnimationVar()` function. This function allows you to set the

value of an animation variable (or several animation variables when they are passed OR'd together). The animation variables are as follows:

Table 8: Animation Variable

Animation Variable	Description
X_VELOCITY	Contains the current velocity of the object in the X direction (in meters per second).
Y_VELOCITY	Contains the current velocity of the object in the Y direction (in meters per second).
Z_VELOCITY	Contains the current velocity of the object in the Z direction (in meters per second).
X_ACCELERATION	Contains the acceleration in the X direction (in meters per second^2).
Y_ACCELERATION	Contains the acceleration in the Y direction (in meters per second^2).
Z_ACCELERATION	Contains the acceleration in the Z direction (in meters per second^2).
X_ROTATION	Contains the angular velocity (in degrees per second) of the object about the X axis.
Y_ROTATION	Contains the angular velocity (in degrees per second) of the object about the Y axis.
Z_ROTATION	Contains the angular velocity (in degrees per second) of the object about the Z axis.

In addition to the standard animation variables provided for you, you may add your own animation data to an object by making a call to `SVE_setUserDefinedData()`. You, as the programmer, are responsible for maintaining and freeing the data.

To animate objects, SVE uses a default animation function. You may, if you wish, change this function by making a call to `SVE_setDefaultAnimationFunc()` and passing it the address of a function of type `SVE_animateObjectFunctionPtr`. This function will be called with `SVE_object` as the parameter for each object that requires animating during a frame.

Each object is also allowed to have its own animation function that can be used instead of or in addition to the default animation function. You can set a local animation function for an object by making a call to `SVE_setAnimationFunc()` and passing it an `SVE_object` and a pointer to a function of type `SVE_animationObjectFunction`. Before a new frame is rendered, the object's animation function is called. If this function returns `TRUE` then the default animation function is called immediately afterward. If `FALSE` is returned, the default animation function will not be called for the object. This allows you to replace or supplement the default animation function for an individual object.

The following example shows how these routines can be used to continuously rotate a cube:

Source viii: Animation Example

```
#include "sve.h"

void SetupAnimations(void);

main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

   if (argc>1 && (argv[1][0]=='v' || argv[1][0]=='V'))
      config=SVE_NORMAL | SVE_HMD;

   printf("Starting application.\n");
   SVE_init("Playin' (sve)", config, &argc, argv);

   if(!SVE_loadWorld("animate.world"))
```

```
    {
       printf("error occured during SVE_loadWorld, exiting.\n");
       SVE_done();
    }

    printf("Beginning event loop\n");
    SVE_initAnimation();
    SetupAnimations();

    SVE_beginEventLoop();

    printf("Done -- Have a nice day.\n");
    SVE_done();
}


void SetupAnimations()
/* Sets up all of the animation variables for the different objects. */
{
    SVE_object o;

    if (o=SVE_findWorldObject("cube1"))
    {
       SVE_setAnimationVar(o, Y_ROTATION, 60.0);
       SVE_setAnimationVar(o, Z_ROTATION, 40.0);
       SVE_setAnimationVar(o, X_ROTATION, 20.0);
       SVE_setObjectAnimation(o, TRUE);
    }

    if (o=SVE_findWorldObject("GT"))
    {
       SVE_setAnimationVar(o, Y_VELOCITY, 0.1);
       SVE_setAnimationVar(o, Y_ACCELERATION, 0.5);
       SVE_setObjectAnimation(o, TRUE);
    }
}
```

Please remember that when you set up your own animation functions, they should be of type `SVE_animateObjectFunction()`, which is defined by a `typedef` in sve.h as:

```
typedef boolean SVE_animateObjectFunction(SVE_object obj);
```

3.4.2 Animation Callbacks

If the application requires more extensive animation than the simple object animations available, the application can provide its own animation functions, which are called every frame before the display is rendered The routines are stored in a stack so that the last animation callback added to the list will be the first called. Animation callbacks are of the form,

```
SVE_status animation(SVE_state state);
```

The return value is ignored.

Animation callback functions are added to the animation callback list with the function `SVE_addAnimationCallback()`. An animation function already on the list can be removed with the function `SVE_removeAnimationCallback()`. The entire list of animation callback functions can be cleared using the function `SVE_removeAllAnimationCallbacks()`. As is the case for event callbacks, animation callbacks are keep in a linked list which is used as a stack. Thus, the last animation callback to be added to the stack will be the first called. The entire linked list of animation callback functions can be retrieved with the function `SVE_getAnimationCallbacks()`.

```
void SVE_addAnimationCallback(SVE_functionPtr function);
void SVE_removeAnimationCallback(SVE_functionPtr function);
void SVE_removeAllAnimationCallbacks(void);
list SVE_getAnimationCallbacks(void);
```

The following example demonstrates the use of animation and event handling. It can be found in the example directory under the name `example3.c`. The application loads in a cube, identified in the SVE

world file `example3.world`, and rotates the cube left or right when the 'l' or 'r' key is pressed respectively. Pressing the space bar will stop the cube.

Source ix: Example 3, Animation and Event Callback Demonstration

```
/**********************************************************************
 * Example3 (sve module)
 *
 * This is the example from the SVE manual.
 *
 * This example will show the power of animation callback functions. It
 * uses an event callback function to check the keyboard. When the 'r'
 * is pressed the object, a cube, will spin around. When the 'l' is
 * pressed the object will spin the otherway around. It will keep on
 * spinning until the spacebar is pressed. The default key's as described
 * in the SVE manual still work.
 **********************************************************************/

#include "sve.h"

/**********************************************************************
 * A pointer to the object to be rotated.
 **********************************************************************/
SVE_object cube;

/**********************************************************************
 * Thi animation callback function is called when you pressed an 'l'. It
 * will rotate the cube around the y-axis.
 **********************************************************************/
SVE_status rotate_right(SVE_state state)
{
   SVE_rotateObject(cube, 3, 'y');
}


/**********************************************************************
 * This animation callback function is called when you pressed an 'r'. It
 * will rotate the cube the other way around the y-axis.
 **********************************************************************/
SVE_status rotate_left(SVE_state state)
{
   SVE_rotateObject(cube, -3, 'y');
}


/**********************************************************************
 * This function handles the callback from the SVE_KEY_PRESS event. When you
 * press 'l' or 'r' it will SVE to use the correct animation callback.
 **********************************************************************/
SVE_status handleKey(SVE_state state)
{
   SVE_status retval = EVENT_IGNORED;

   if (state->eventType == SVE_KEY_PRESS) {
      switch(((SVE_keyEvent *)state->eventData)->keyVal) {
      case 'r': SVE_addAnimationCallback(rotate_right);
                retval = EVENT_CONSUMED;
                break;
      case 'l': SVE_addAnimationCallback(rotate_left);
                retval = EVENT_CONSUMED;
                break;
      case ' ': SVE_removeAllAnimationCallbacks();
                retval = EVENT_CONSUMED;
                break;
      }
   }
   return(retval);
}


main(int argc, char *argv[])
```

```
{
   SVE_config config = SVE_NORMAL;

/**********************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the SVE Basics section of
 * the manual for a description of the different configurations you can use.
 **********************************************************************/
   printf("Starting application.\n");
   SVE_init("Example3 (sve)", config, &argc, argv);

/**********************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 **********************************************************************/
   if(!SVE_loadWorld("example3.world"))
   {
      printf("error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/**********************************************************************
 * Find in the world an object called cube.
 **********************************************************************/
   cube = SVE_findWorldObject("cube");

/**********************************************************************
 * Tell SVE that we are interested in the SVE_KEY_PRESS event, and tell
 * SVE which function will handle this callback.
 **********************************************************************/
   printf("Registering an input callback.\n");
   SVE_registerCallback(SVE_KEY_PRESS, handleKey);

/**********************************************************************
 * SVE will take over control of the program until it is finnished.
 **********************************************************************/
   printf("Beginning event loop\n");
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
}
```

The cube rotation is handled in the animation callbacks `rotate_right()` and `rotate_left()`. Note that pressing the 'l' or 'r' key adds the appropriate callback to the list, therefore the rotations are additive. In other words, pressing the 'l' key and then the 'r' key results in a cube that appears to be stationary, although the cube is actually rotated left by one animation callback, and then rotated right by another animation callback. Pressing one of the keys many times results in a larger accumulated rotation, and the cube appears to rotate faster. Pressing the space bar removes all of the animation callbacks in the list, and therefore effectively stops the cube.

3.4.3 Rendering Callbacks

Rendering callback functions are also known as "frame callbacks" as they are called once per frame. Note that a display can consist of 0 (no rendering done), 1, or 2 (stereo) frames. Frame callbacks are called after the viewing and model transformations are set up to render to the frame. In the case of stereo viewing, a frame callback function can determine which eye is being rendered by using the "`currentEye`" field of the "`state`" structure, which will be `SVE_LEFT_EYE` or `SVE_RIGHT_EYE`. There are two types of frame callbacks, one which is global to the environment, and one which is associated with an object. There are two types of *global* frame callbacks, one which is called before the SVE object tree is rendered, and one which is called after the SVE object tree is rendered. The *object* frame callbacks are called just before the particular object is rendered by the SVE system, but after the transformations are set up to render in the object's coordinate system. Thus, anything rendered, either using an SVE rendering function or native 3D rendering routines, will be rendered in the object's coordinate system.

It should be noted at this point that the SVE library uses one of two possible windowing and rendering environments, either the GL library from SGI, or the X windowing system with the OpenGL library. The application programmer is free to choose either one by way of changing the application's makefile to include the appropriate SVE library (see APPENDIX A: "Starter Kit" on page 105 for more details). If the application programmer wishes to use native 3D graphics or windowing routines and wishes to support both environments, the C `#ifdef` directive can be used to determine which SVE library has be included at compile time (either `GL` or `OPENGL` is defined, depending on whether GL or OpenGL rendering is performed, respectively).

The global frame callback functions are of the form,

```
SVE_status frameFunction(SVE_state state);
```

The return value is ignored. The functions to add to, remove from, remove all from, and get the frame callback list are:

```
void SVE_setFrameCallback(SVE_functionPtr function);
void SVE_removeFrameCallback(SVE_functionPtr function);
void SVE_removeAllFrameCallbacks(void);
list SVE_getFrameCallback(void);
```

The corresponding functions that deal with the frame callback list which is used after all of the SVE objects are rendered are:

```
void SVE_setFrameEndCallback(SVE_functionPtr function);
void SVE_removeFrameEndCallback(SVE_functionPtr function);
void SVE_removeAllFrameEndCallbacks(void);
list SVE_getFrameEndCallbacks(void);
```

The functions that deal with the frame callback list associated with a particular object are:

```
void SVE_setObjectFrameCallback(SVE_object object,
                                SVE_objectFunctionPtr function);
void SVE_removeObjectFrameCallback(SVE_object object,
                                   SVE_objectFunctionPtr function);
void SVE_removeAllObjectFrameCallbacks(SVE_object object);
list SVE_getObjectFrameCallback(SVE_object object);
```

Where the frame callback function has the form,

```
SVE_status objectFrameFunction(SVE_object object, SVE_state state);
```

The "object" parameter identifies which object is about to be rendered. The return value is ignored.

3.4.4 Object Rendering

For most VE applications, the built in rendering facilities of SVE are sufficient to efficiently display the correct view of the virtual world without any additional programming besides object animation through altering the attributes of objects (see 3.7. "Object Appearance" on page 63). Occasionally, though, an application may have more complex requirements where by the application needs to be able to render an SVE object or object tree which is not part of the normal rendering tree (see 3.6. "Object trees" on page 53), or re-render parts of the normal rendering tree in a different context. For example, imagine a crystal ball which contains a scaled down version of one of a set of different worlds, or an application may need to render a different view of the world in a different window (which would also require native windowing system programming). The frame callback routines described in section 3.4.3 provide a place to take care of these special cases. This section describes the routines that can be used to render SVE objects and object trees that are not necessarily part of the normal rendering tree.

Before using any of these rendering routines, however, you should be sure that there isn't an easier method that achieves the desired result through changing object attributes. For example, if the application wishes to show one object in some situations, and another object in other situations, it is easy to make only one object visible at a time by changing the `visible` attribute of each object appropriately. Thus, the complexities of rendering correctly are avoided (or, more precisely, taken care of by SVE).

In order to use the rendering routines effectively, it is important to understand the steps SVE goes through in order to render a scene. When the SVE main loop enters the "render the world" phase (shown in Figure 6. "The interaction loop." on page 16), the system follows the following steps:

```
1. Generate the viewing an perspective matrixes from the position and orientation
   of the eye point (usually represented by the "SVE eye" object) and the
   viewplane (usually represented by the "SVE viewplane" object).
2. Determine the geometry of objects (where more than one geometry has been
   specified for an object, and where objects are set to always face the user).
3. Perform view culling. If the object or its child objects are not visible by
   the viewer, then the object is not rendered.
4. Render lights
5. Call the user's frame callback functions.
6. For each object in the object tree that was not culled in step 3,
   a. Set the 3D renderer to render in the object's coordinate system,
   b. Call the user's object frame callback functions,
   c. Render the object's geometry.
7. Call the user's frame end callback functions.
8. Display the frame in the SVE window.
```

As part of one of the frame callback functions, an application can cause an SVE object to be rendered in the "current coordinate reference frame" using the `SVE_renderObject()` function. A list of SVE objects (which is how a SVE object tree is represented) can be rendered in the current coordinate reference frame using the `SVE_renderObjectList()` function. The entire world tree contained in a SVE world state structure (either the current world state, or the state of a different world altogether) can be rendered in the current coordinate reference frame using the `SVE_renderWorld()` function. The current coordinate reference frame is determined by which type of frame callback the rendering function appears in. In the regular frame callback (step 5) or frame end callback (step 7) functions, the coordinate reference frame is in world coordinates. In the object frame callback functions (step 6b), the coordinate reference frame is defined by the object's coordinate system, which is defined by the coordinate transformations of it and its parent and its ancestors combined.

Rendering In a Frame Callback

If these functions are used outside of a frame callback function, the coordinate system will not be correctly set up, and they will not work as expected. It should be noted, also, that any object frame callback functions for objects rendered with these functions will be called, giving the potential for an endless loop of rendering. It is therefore good practice to ensure that object frame callbacks only render once per frame. It should also be noted that if an object is culled because its bounding volume does not intersect the viewing volume (i.e. it can't be seen by the viewer), then the object frame callbacks will not be called even if objects rendered by those callbacks *would* be in the viewing volume (but see the "Object Culling" section below).

```
void SVE_renderObject(SVE_object obj);
```

The `SVE_renderObject()` function renders an SVE object, including its child sub-trees, after first determining its geometry (as in step 2) and performing view culling (step 3). Note that the object's frame callback functions will be called.

```
void SVE_renderObjectList(list objectTree);
```

The `SVE_renderObjectList()` function renders each SVE object in the given list, including their child sub-trees, after first determining their geometries (step 2) and performing view culling (step 3). Note that the objects' frame callback functions will be called. If `SVE_WORLD` is given for the `objectTree` parameter, the current world tree will be used.

```
void SVE_renderWorld(SVE_state worldState);
```

The `SVE_renderWorld()` function renders the world tree of the world state given as is done with the `SVE_renderObjectList()`. In addition, any lights contained in the world tree will be rendered (as in step 4), although those lights will not affect objects that have already been rendered for the current frame.

If the world state given is not the current world state, then the viewing and perspective matrixes must be set before `SVE_renderWorld()` is called. This can be done with the following function.

```
void SVE_getViewingAndPerspectiveMatrix(SVE_state worldState, M_matrix viewing,
                                         M_matrix perspective);
```

If the new world state does not contain an eye point object and view plane object, then the current world state (`SVE_worldState`) should be given for the `worldState` parameter. The objects in the new world state will be rendered, therefore, as if they were in the current world's coordinate system.

Rendering a New Frame

If, for some reason, the application wishes to force a new frame to be rendered, then it can use one of the following functions (even outside the "frame render" phase of the SVE loop).

```
void SVE_renderNow(SVE_state worldState, SVE_functionPtr frameCallback);
```

The `SVE_renderNow()` function performs all of the steps given above for render (1 through 8) except that for step 5, where the frame callbacks are called, only the given function, `frameCallback`, is called, and step 7 (calling the frame end callbacks) is skipped. Thus, if this function appears in a frame callback, or frame end callback (which will produce strange results, as the previous frame will not have completed and displayed its results, and will be lost), and the given function is not the same callback function, then an endless loop is avoided.

```
void SVE_renderNowWithFrameCallbacks(SVE_state worldState, boolean newFrame);
```

The `SVE_renderNowWithFrameCallbacks()` function performs the frame rendering just as if the SVE loop had reached the "frame render" phase. Thus, all steps (1 through 8) are performed unless `newFrame` is `FALSE`. If `newFrame` is `FALSE`, then steps 1, 2, 3, 4, 5, and 8 are skipped.

If the application needs to call `SVE_renderNowWithFrameCallbacks()` in a frame callback routine (which will result in the loss of the previous frame up to the point of the call unless `newFrame` is `FALSE`), it is important to take precautions against an endless loop. The following code, for example, will avoid an endless loop.

```
void frameCallback(SVE_state state)
{
   static boolean inFrameCallback = FALSE;

   if (!inFrameCallback) {
      inFrameCallback = TRUE;
      SVE_renderNowWithFrameCallbacks(state);
      inFrameCallback = FALSE;
   }
}
```

Object Culling

In step 3, before any SVE objects are rendered, it is first determined whether the objects are visible by the viewer. This is normally done by testing to see if each object's bounding volume intersects with the viewing volume (the volume visible by the viewer). Note that an object's bounding volume includes its own geometry and the geometry of its children (and their children, and so on). (Object boundaries are discussed in section 3.7.3 "Object Boundaries" on page 72.) This default behavior can be changed by adding a new culling function, or replacing the default culling function.

A new culling function should be of the following form.

```
SVE_vistype SVE_cullFunction(SVE_object o, M_matrix worldToEye);
```

The SVE object `o` is the object which is being tested for culling. The `worldToEye` matrix can be used to transform a point from world coordinates to eye coordinates (where a point (x_w, y_w, z_w, 1) is transformed to (x_e, y_e, z_e, w), and where the point (x_e/w, y_e/w, z_e/w) is visible if it is in the volume $-1<x<1$, $-1<y<1$, and $-1<z<1$).

The function should return one of the following predefined values: SVE_ALL_VISIBLE (if the object and its children are complete visible. The SVE renderer will assume the object's children are also completely visible), SVE_PART_VISIBLE (if the only part of the object is visible. The SVE renderer will also check each of the object's children for visibility), SVE_CHILDREN_ONLY_VISIBLE (if the object's geometry is not visible, but its children may be visible), SVE_NONE_VISIBLE (if neither the object nor the object's children are visible. The SVE renderer will not render the object or its children), and SVE_VISIBILITY_UNKNOWN (the SVE renderer will render the object, and will check its children for visibility).

If more than one culling function is defined, then they are called in order until one returns SVE_NONE_VISIBLE or SVE_CHILDREN_ONLY_VISIBLE. If no function returns one of these values, the object's visibility is set to the value returned by the last culling function.

The SVE system begins with one default culling function. The application can add new culling functions to the list using the SVE_addCullingFunction() function. New functions are called before function that already existed in the list. A culling function can be removed by name using SVE_removeCullingFunction(). All culling functions can be removed using SVE_removeAllCullingFunctions(), which leaves no culling functions (in this case, every object will use the SVE_PART_VISIBLE value for its visibility). The list of culling functions can be obtained using the SVE_getCullingFunctions() function.

```
void SVE_addCullingFunction(SVE_cullFunctionPtr function);
void SVE_removeCullingFunction(SVE_cullFunctionPtr function);
void SVE_removeAllCullingFunctions(void);
list SVE_getCullingFunctions(void);
```

The default culling function is SVE_getObjectVisibility(). This function can be used in a user-defined culling function to determine the object's visibility.

```
SVE_vistype SVE_getObjectVisibility(SVE_object obj, M_matrix worldToEye);
```

3.4.5 Navigation

A common response to an event or animation callback is to move the user around the environment. The SVE library includes functions to move the user which use the flightSpeed field of the global state structure to determine how far the user moves each call to the function.

The default navigation method, which is common to many virtual environment applications, allows the user to "fly" when the middle mouse button is pressed (it is a callback for the SVE_MIDDLE_MOUSE_DRAG event), moves the user in the direction he or she is looking. This function, SVE_fly(), takes the global state structure, and changes the position of the "origin" of the world (which is an object referred to by the originObject field of the state structure, usually called "ORIGIN") to move the user in the direction of the last view rendered.

```
SVE_status SVE_fly(SVE_state state);
```

Another function, SVE_flyWithDirection(), will move the user in the direction of the negative Z direction of the matrix given. The matrix could be the world position matrix of an object, which can be retrieved with the SVE_getWorldMatrix() function. This function could be called by the user's animation or event callback routine.

```
void SVE_flyWithDirection(SVE_state state, M_matrix direction);
```

Two other functions provide the same movement of the user except that the user is constrained to be inside the bounding volume of a given object. It should be noted that this does not strictly constrain the user inside the object itself, as the object could be smaller than the bounding volume (usually a box) around it. These functions work very well, however, for square rooms.

```
void SVE_flyInObject(SVE_state state, SVE_object o);
void SVE_flyInObjectWithDirection(SVE_state state, SVE_object o,
                                  M_matrix direction);
```

The flight speed, which is the velocity of the user as he or she flies, can be changed using the `SVE_setFlightSpeed()` function.

```
void SVE_setFlightSpeed(float speed);
```

3.5. Objects

In the SVE system, an object is an entity identified with a unique character string name. It has numerous attributes, including a location, orientation, and scale in relationship to a "parent" object; and a (possibly empty) list of child objects. The child objects are considered part of its parent. Therefore characteristics of an object generally affect its children. For example, an object can be invisible, which would imply that its children (and their children and so on) are also invisible. Since an object's position is defined relative to its parent, when an object moves, its children will move as well as if they are rigidly attached to the parent.

An object may or may not have a geometric representation. Objects with no geometry are used to group other objects and serve as place holders in the object tree.

As mentioned above, each SVE object is identified by a unique character string name. Thus, if an object is created using a name that is already used by another object, the name of the other object is changed. Thus, an object name always refers to the last object created with that name. The new name of the other object is of the format "name:X" where X is an integer which is not associated with any other object with the same "name" and format. Application designers are therefore discouraged from using the colon(:) character in object names. It is possible to get a list of objects which were created using the same name.

Because the SVE system enforces the constraint of unique names, it is important that an application changes an object's name only through the use of the function `SVE_changeObjectName()`.

```
void SVE_changeObjectName(SVE_object object, char *newName);
```

The structure that defines an SVE object contains many fields. A table describing some of the fields of the `SVE_object` structure that an application may wish to examine or change, as well as a description of each, follows.

Table 9: SVE Object Fields

Field	Type	Description
name	char *	Character string name of the object (for examination only).
parent	SVE_object	Parent object. NULL if the object has no parent.
children	list	Linked list of child objects. Empty if the object has no children.
visible	boolean	Determines if an object (and its children) is visible.
selectable	boolean	Determines if an object is selectable. Many selection and collision detection routines will only consider an object if this flag is TRUE.
position	M_matrix	Matrix representation of an object's position in relation to its parent.
highlight	boolean	Determines if the object is being highlighted. It is possible to set an object's highlight color, or a completely separate geometry that an object uses when it is highlighted.
cullable	int (NOT_CULLABLE, CULLABLE_BOX, or CULLABLE_SOHERE)	Determines if an object is considered for view-frustrum culling, and if so, whether the object is represented as a bounding box or bounding sphere when the culling calculations are done. Culling is usually efficient only for objects containing a complex geometric representation which are not viewed constantly. (If the user is always inside a room object, it should be NOT_CULLABLE, for example, as it is always in the user's view.)
facingViewerUpright	boolean	Setting this to TRUE gives the object the characteristic that its positive Z axis will always point at the viewer, while staying on the world's X-Y plane.

Table 9: SVE Object Fields

Field	Type	Description
facingViewer	boolean	Setting this to TRUE gives the object the characteristic that its positive Z axis will always point at the viewer.
UserPtr	void *	Pointer available for use by the application.

3.5.1 Creating Objects

An object can be created from scratch using the `SVE_createEmptyObject()` function.

```
SVE_object SVE_createEmptyObject(char *name);
```

Usually, however, objects are retrieved from disk or copied from existing objects. The `SVE_createObjectCopy()` function creates a copy of another object.

```
SVE_object SVE_createObjectCopy(SVE_object source, char *newName,
                                 boolean copyChildren);
```

If the `source` object is `NULL`, the function creates an empty object. If the `newName` character string is `NULL`, the name of the source object is used (causing the source object name to be changed). The "copyChildren" parameter specifies if the entire object tree (which the source object as its root) will be copied, or if just the source object will be copied. In the later case, the object will have no children.

3.5.2 Loading and Saving Objects

An object's geometry definition can be retrieved using the `SVE_loadObject()` routine.

```
SVE_object SVE_loadObject(char *filename, char *name, boolean *posInitialized);
```

The file identified by the given `filename` can be an SVE object file format or a Wavefront™ .obj file format. If the file is not found in the current directory, the default object directory list will be used to search for the file elsewhere. The `name` parameter specifies the object's name. The `posInitialized` parameter indicates if an initial position for the object has been defined (`TRUE`) or not (`FALSE`). Note that the object file does not contain many of the attribute specifications defined in a world file. In fact, the object file only contains the geometric description of the object, and an optional bounding volume specification. Attributes such as "visible", "selectable", and "position" must be set by the application, if they are to be different from the default, or (if `posInitialized` is `TRUE`) if the pre-defined position is to be over-ridden.

An object's geometry definition can be saved to a file using the `SVE_saveObject()` function.

```
boolean SVE_saveObject(SVE_object o, char *filename);
```

3.5.3 Finding Objects

Regardless of whether an object becomes part of an object tree (described below) or not, every object created is referenced in an object repository maintained by the SVE system. It is possible to find any object that has been created if the name of the object is known. The `SVE_findObjectInRepository()` function accomplishes this search.

```
SVE_object SVE_findObjectInRepository( char *name);
```

3.6. Object trees

The SVE system maintains a tree of objects which is traversed once for each frame rendered. Objects in this tree that have a geometric representation are therefore rendered to the display once each frame. It is possible for other trees of objects to be created and manipulated, and for lists of objects to be created and manipulated. An object tree is an ordered, rooted tree where each node may have many children. The children are represented as a linked list of object nodes. In the SVE system, the relationship between a parent object and a child object is one of location, orientation and scale (often referred to simply as its position). Another way to express this relationship is to say that a child's position is defined in terms of its parent's coordinate system, which is, in turn, defined as its position in relationship to its parent's coordinate system, and so on. All objects in a tree have a position that is ultimately defined in terms of the coordinate system of the root node (the "world position" - see 3.7.1 "Object Position" on page 63).

3.6.1 Object tree manipulation

An example of an object tree called "root" is shown in the Figure 11. "Object Tree Example" on page 53.

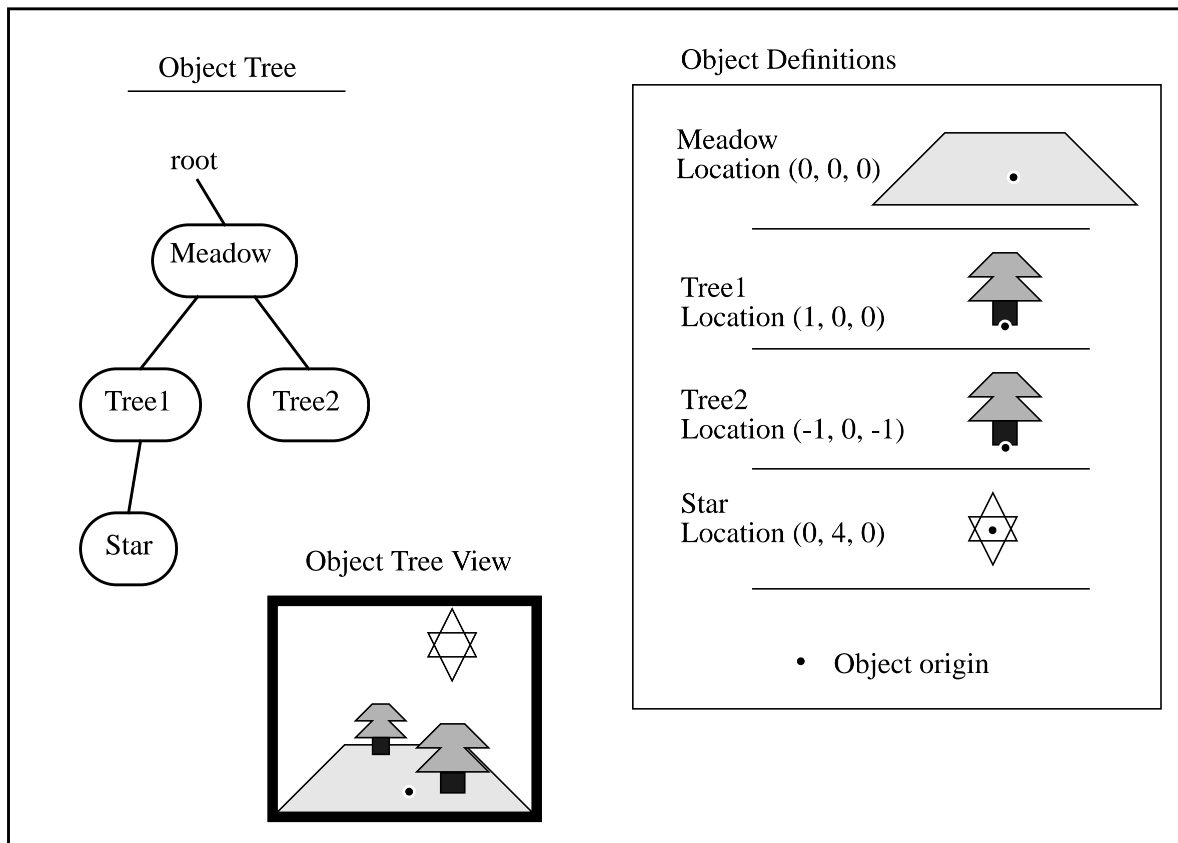


Figure 11. Object Tree Example

The object definitions are shown with only a location component, although the objects could have an orientation and scale component. The location component describes a vector between the origin of the parent object to the origin of the child object. The view that would result if the object tree were to be rendered is shown as well. Note that the tree "root" is of the `list` type, as is the list of children for each object. The children objects are maintained as a linked list of objects where new objects are placed at the end of the list.

Objects can be added to a tree or object list using two different methods which differ in how the object's current position is handled. In one case, the object's position in relation to the root of the object tree is maintained. Thus, when the object is attached to another object, it's current position is replaced with a new position from the new parent which keeps the object where it was. The `SVE_attachToObject()` function does this.

```
void SVE_attachToObject(SVE_object child, SVE_object parent);
```

This is the function that works best when an object is "grabbed" by another object, and will be moved from where it is currently, and then returned to its previous place in the tree, but at its new location. Note that if the object is being moved from one tree to another, the function assumes that the roots of both trees are in the same coordinate system.

The other method of adding an object as a child of another object does not affect the position of the object at all. If the object has a position defined, it will determine where the object is in the coordinate system of the new parent object.

```
void SVE_addChildToObject(SVE_object child, SVE_object parent);
```

This function is faster than the previous one in most cases.

An object can be removed from an object tree with one of the following functions, depending on whether the object pointer or object name is to be given:

```
SVE_object SVE_removeObject(char *name, list *objectTree);
void SVE_removeObjectEntry(SVE_object object, boolean attachChildrenToParent);
```

Note that the object itself is not deleted, and can be added to a tree again. The `attachChildrenToParent` flag indicates if the removed object's children should be re-attached to the removed object's parent (`TRUE`), or remain the children of the removed object (`FALSE`). The `SVE_deleteObject()` function actually deletes the object. If the `attachChildrenToParent` flag is `FALSE`, then its child objects are also deleted, otherwise the child objects are attached to the object's parent.

```
void SVE_deleteObject(SVE_object object, boolean attachChildrenToParent);
```

The following example illustrates the manipulation of the object tree through grabbing and releasing of objects using a cursor object controlled by the user's hand. This example will only work (without modification, at least) with two trackers, as the only way to move the user's hand in the example is to use a tracking device. The example can be found as `example5.c` in the examples directory.

Source x: Example5, Manipulating the Object Tree in the Environment.

```
/* Example5 (sve module)

  This example demostrates the use of object boundaries.  The second cursor
  (which is attached to the second tracker), is used as a three dimensional
  cursor.  When the left mouse button is pressed, the position of the cursor
  is used to see if an object has been selected (the cursor is in the object).
  While the mouse button is pressed, the selected object is linked to the
  cursor as a child, and therefore the object follows the cursor.  When the
  mouse button is released, the object is linked back to the world in its
  new position.

*/

#include <math.h>
#include "sve.h"

/* function prototype */
SVE_status release(SVE_state state);

/* global vars: */
SVE_object current_object, current_parent;


SVE_status grab(SVE_state state)
```

```
{
   SVE_object object;
   SVE_object pointer;
   SVE_object cursor;
   M_matrix pointerPos;
   float margin = 0.0;
   float onlySelectable = TRUE;
   SVE_status retval = EVENT_IGNORED;

   printf("trying to grab\n");
   if (SVE_IS_PRESS_EVENT(state->eventType)
      && ((SVE_stateChangeEvent *)state->eventData)->pressed) {

      cursor = SVE_getCursorObject();
      pointer = SVE_findObject("pointer", cursor->children);
      SVE_getWorldMatrix(pointer, pointerPos);
      object = SVE_objectMatrixHit(SVE_WORLD, pointerPos, margin,
                                   onlySelectable);

      if(object) {
         printf("object %s grabed\n",object->name);
         SVE_changeText(pointer, "Grabbed");
         current_object = object;
         current_parent = object->parent;

         SVE_attachToObject(object, pointer);
         SVE_removeCallback(SVE_LEFT_MOUSE, grab);
         SVE_registerCallback(SVE_LEFT_MOUSE, release);
      } /* if */
      retval = EVENT_CONSUMED;
   } /* if */
   return(retval);
}


SVE_status release(SVE_state state)
{
   M_matrix pos1, pos2;
   SVE_object pointer;
   SVE_object cursor;
   SVE_status retval = EVENT_IGNORED;

   if (SVE_IS_PRESS_EVENT(state->eventType)
      && !((SVE_stateChangeEvent *)state->eventData)->pressed) {

      cursor = SVE_getCursorObject();
      pointer = SVE_findObject("pointer", cursor->children);
      SVE_changeText(pointer, "<=>");

      SVE_attachToObject(current_object, current_parent);
      printf("object %s released\n",current_object->name);
      current_object = NULL; /* reset globals */
      current_parent = NULL;
      SVE_removeCallback(SVE_LEFT_MOUSE, release);
      SVE_registerCallback(SVE_LEFT_MOUSE, grab);
      retval = EVENT_CONSUMED;
   } /* if */
   return(retval);
}


void main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;
   SVE_object pointer;
   boolean    posInitialized;

   printf("Starting application\n");
   if ((argc > 1) && (strchr(argv[1],'t') != NULL))
      config = SVE_HMD;

   SVE_init("example 5 (sve)", config, &argc, argv);
```

```
    if(!SVE_loadWorld("example5.world"))
    {
       printf("error occured during SVE_loadWorld, exiting \n");
       SVE_done(); /* exit */
    } /* if */

    pointer = SVE_loadObject("pointer.object", "pointer", &posInitialized);
    SVE_addChildToObject(pointer, SVE_getCursorObject());

    if(config & SVE_HMD)
    {
       SVE_removeAllCallbacks(SVE_LEFT_MOUSE);
       SVE_removeAllCallbacks(SVE_LEFT_MOUSE_DRAG);
       SVE_registerCallback(SVE_LEFT_MOUSE, grab);
    }

    printf("Beginning graphics loop\n");
    SVE_beginEventLoop();

    printf("Done -- Have a nice day.\n");
    SVE_done();
}
```

It is possible to search a given object to find an object of a particular name using the `SVE_findObject()` function.

```
SVE_object SVE_findObject(char *name, list objectTree);
```

This function returns the object in the given objectTree that has the given name. Note that, because every object in SVE has a unique name, if more than one object is created using the same name, only the last object created will have that name. To retrieve a list of objects that were created using the same name, the `SVE_findAllObject()` function can be used.

```
list SVE_findAllObjects(char *name, list objectTree, boolean includingChildren);
```

Setting the "includingChildren" flag to `FALSE` will limit the search to the first level of the given object tree only. This function returns a linked list of the objects found, which is not an object tree as the objects do not necessarily have the same parent. The objects in this list can be retrieved using a combination of the `SVE_getFirstObject()` function and the linked list function `getNext()` defined in APPENDIX C: 2.1. "Linked List" on page 177. (Note that the function requires the *reference* to the object list.)

```
SVE_object SVE_getFirstObject(list *objectList);
```

A demonstration of how to deal with many objects that are created with the same name is done in the following modified version of `example3.c` (the rotating cube). The code is contained in the examples directory as `example3mod.c`, which uses the world file `example3mod.world`. The world file describes many cubes, which are all simultaneously rotated when the 'l' or 'r' key is pressed.

Source xi: Modified Example 3, Rotating Many Cubes

```
/*************************************************************************
 * Example3 modified (sve module)
 *
 * This is an example from the SVE manual.
 *
 * This example will show the power of frame callback functions. It uses
 * a callback function to check the keyboard. When the 'r' is pressed
 * all cube objects will spin around. When the 'l' is pressed the
 * they will spin the otherway around. They will keep on spinning until
 * the spacebar is pressed. The default key's as described on page 6 of
 * the SVE manual still work.
 *************************************************************************/

#include "sve.h"

/*************************************************************************
 * The list of objects to be rotated
 *************************************************************************/
```

```
list cubeList;


void rotate_cubes(float angle)
{
   list nextCube;
   SVE_object cube;

   nextCube = cubeList;
   while (!listEmpty(nextCube)) {
      cube = SVE_getFirstObject(&nextCube);
      SVE_rotateObject(cube, angle, 'y');
      nextCube = getNext(nextCube);
   }
}

/**********************************************************************
 * This animation callback function is called when you pressed an 'l'. It will
 * rotate the cubes around the y-axis.
 **********************************************************************/
SVE_status rotate_right(SVE_state state)
{
   rotate_cubes(3);
}


/**********************************************************************
 * This animation callback function is called when you pressed an 'r'. It will
 * rotate the cube the other way around the y-axis.
 **********************************************************************/
SVE_status rotate_left(SVE_state state)
{
   rotate_cubes(-3);
}


/**********************************************************************
 * This function handles the callback from the SVE_KEY_PRESS event. When you
 * press 'l' or 'r' it will SVE to use the correct animation callback.
 **********************************************************************/
SVE_status handleKey(SVE_state state)
{
   SVE_status retval = EVENT_IGNORED;

   if (state->eventType == SVE_KEY_PRESS) {
      switch(((SVE_keyEvent *)state->eventData)->keyVal) {
      case 'r': SVE_addAnimationCallback(rotate_right);
                retval = EVENT_CONSUMED;
                break;
      case 'l': SVE_addAnimationCallback(rotate_left);
                retval = EVENT_CONSUMED;
                break;
      case ' ': SVE_removeAllAnimationCallbacks();
                retval = EVENT_CONSUMED;
                break;
      }
   }
   return(retval);
}


main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

/**********************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the SVE Basics section of
 * the manual for a description of the different configurations you can use.
 **********************************************************************/
   printf("Starting application.\n");
```

```
   SVE_init("Example3 (sve)", config, &argc, argv);

/*********************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 *********************************************************************/
   if(!SVE_loadWorld("example3mod.world"))
   {
      printf("error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/*********************************************************************
 * Find in the world an object called cube.
 *********************************************************************/
   cubeList = SVE_findAllObjects("cube:12", SVE_WORLD, TRUE);

/*********************************************************************
 * Tell SVE that we are interested in the SVE_KEY_PRESS event, and tell
 * SVE which function will handle this callback.
 *********************************************************************/
   printf("Registering an input callback.\n");
   SVE_registerCallback(SVE_KEY_PRESS, handleKey);

/*********************************************************************
 * SVE will take over control of the program until it is finnished.
 *********************************************************************/
   printf("Beginning event loop\n");
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
}
```

The following routines add and remove objects to and from an object list:

```
void SVE_addToObjectList(SVE_object o, list *objectTree);
SVE_object SVE_removeFromObjectList(SVE_object o, list *objectList);
SVE_object SVE_removeFirstObject(list *objectList);
```

Although it is possible to use these functions to add and remove objects to and from the first level of the object tree, since an object tree root is really a list of objects, it is highly recommended that the `SVE_attachToObject()`, `SVE_addChildToObject()`, and `SVE_removeObject()` functions are used instead for object trees that will be rendered.

For debugging purposes, the following function is provided to print an object tree to standard output:

```
void SVE_printObjectList(list objectList, boolean printChildren);
```

The `printChildren` flag determines whether the entire tree or just the first level of the tree will be displayed.

3.6.2 Rendered Object Tree (SVE_WORLD)

The SVE system maintains one object tree that is rendered at each frame. The object tree contains a hierarchical tree of geometries, as well as information to determine the point of view from which those geometries are viewed. This object tree is contained in the `objectTree` field of the `SVE_state` structure which is passed into most SVE callback routines. In addition, any function that calls for an object tree or object list or reference to either will accept the constant `SVE_WORLD` to indicate that the state's `objectTree` is to be used.

The following functions perform many common object tree routines using the state object tree implicitly:

```
SVE_object SVE_findWorldObject(char *name);
```

Returns the first object in the `SVE_WORLD` object tree which has the given name.

```
void SVE_addToWorldTree(SVE_object o);
```

Adds the given object to the `SVE_WORLD` object tree. The object will be at the first level.

```
void SVE_getWorldMatrix(SVE_object o, M_matrix m);
```

Retrieves the global position matrix of the object, which is the product of every position matrix of the objects on the path from the root of the `SVE_WORLD` object tree to the given object.

Default Tree

The object tree which is maintained by the SVE system contains a default sub-tree which is used to represent components of the user, such as the current position of the body, the hand, the head, and the eye-point. The following figure shows a representation of the default world object tree, and the result of adding the example object tree shown above to the SVE world tree. Note that the difference between the "HMD

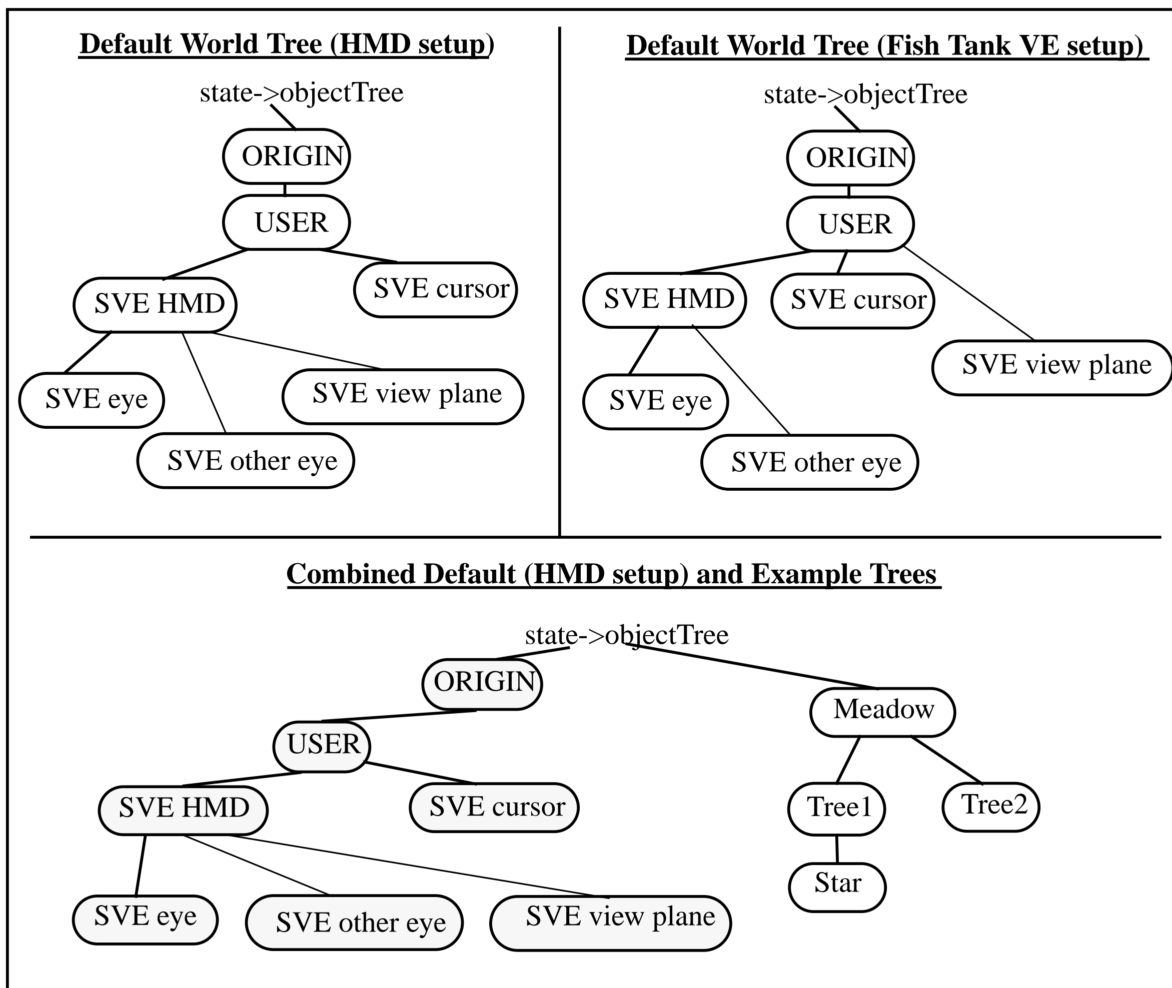


Figure 12. Default World Tree and Example World Tree

setup" and the "Fish Tank VE setup" is the placement of the view plane object ("SVE view plane") in the tree. If the view plane (which represents the window through which the viewer sees the environment) moves with the user's head, as is the case with head mounted displays, then the view plane is attached to the HMD object. If the view plane is stationary in the reference coordinate system of the user, as is the case in Fish Tank VE setups, where the display is stationary even though the user's head moves, then the view

plane is attached to the "USER" object, which represents the reference coordinate system of the tracking device.

Each default object serves a particular purpose. The following table lists the default objects and their purpose. The purposes of each default object can be reassigned to other objects in the object tree by simply

Table 10: Default Object Associations and Uses

Object	State field Associated With	Purpose
ORIGIN	originObject	Establishes a coordinate system for the user, where (conceptually) the origin is at the feet of the user (or, more precisely, at a particular spot on the floor). If the origin object's position changes, then the user's coordinate system "moves" in the world. Thus, the user can be tele-ported or flown to different positions in the world, or become "attached" to different objects (by attaching the origin object to the world object in the object tree).
USER	userObject	Identifies the position of the user in the world. When trackers are used, this is the position of the tracker reference point (usually the position of the transmitter). In this case, the position matrix of the user object should match the position of the transmitter in relationship to the desired "origin" of the user's coordinate system (usually at the floor). If trackers are not used, this object is placed 1.8 meters above (along the positive Y axis) the origin object.
SVE HMD	hmdObject	Refers to the position of the user's head in relationship to the user's position. When the trackers are being used, this object usually follows the tracker attached to the user's head (determined in the .sve.init file).
SVE cursor	cursorObject	Refers to the position of the user's hand in relationship to the user's position. When the trackers are being used, this object usually follows the second tracker (determined in the .sve.init file).
SVE eye	viewingObject	Identifies the point of view of the display. The display is rendered from the point of view of the origin of this object, in the direction of the view plane object. When the trackers are being used, this object is usually positioned to reflect the eye's position relative to where the tracker is attached to the user's head (this can be set in the display configuration file).
SVE other eye	viewingObject2	Identifies the second point of view of the display for stereo display set-ups.
SVE view plane	viewPlaneObject	Identifies the object whose position and orientation determines the position and orientation of the view plane. In a head mounted display set up, this object is usually attached to the "SVE HMD" object. In desktop set ups, this object is usually attached to the "USER" object.

changing the appropriate field in the world state structure. For example, the following SVE event callback function renders the display from the point of view of the "Star" object in the example tree above.

```
SVE_status changePointOfViewToStar(SVE_state state)
{
   state->viewingObject = SVE_findWorldObject("Star");
   SVE_addChildToObject(state->viewPlaneObject, state->viewingObject);
   /* May want to alter viewplane's position, and field of view/aspect ratio
      parameters */
```

```
        return(EVENT_CONSUMED);
    }
```

The hmdObject and cursorObject objects are used often in determining a user's position and in the user's interaction with the world. To simplify things, the following routines are provided to retrieve the hmdObject and cursorObject objects.

```
SVE_object SVE_getHMDObject(void);
SVE_object SVE_getCursorObject(void);
```

The world position of each object (taking into account the position of the userObject object, and its ancestors) can be obtained using these functions.

```
void SVE_getHMDPosition(M_matrix pos);
void SVE_getCursorPosition(M_matrix pos);
```

3.6.3 Tracker Objects

Tracker objects are empty geometric objects that are introduced automatically when a tracking device is initialized using a "tracker" line in the initialization file, or when SVE_initTracker() is called. The tracker initialization includes a number identifier and the name of an object which should follow the tracker (the "attached" object). The object that is created is called "SVE tracker X" where X is the given identifier number. The object is inserted between the attached object and its parent. Therefore, as the

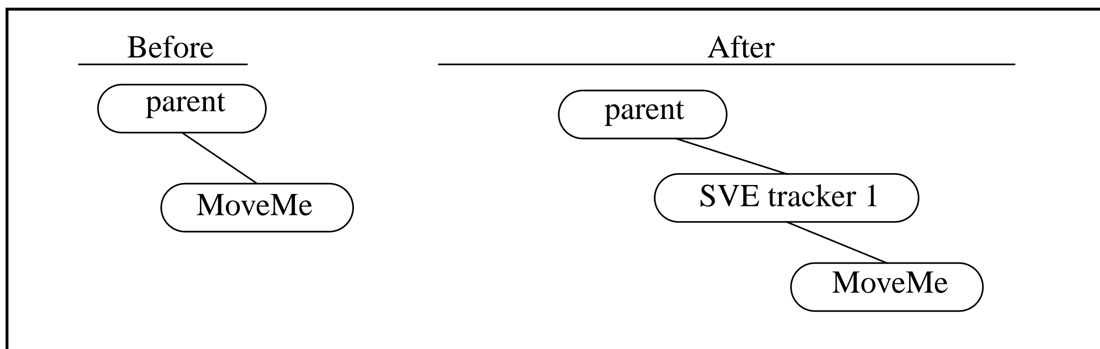


Figure 13. Inserting Tracker Object as Parent of Attached Object ("MoveMe")

tracker object changes location and orientation, the attached objects follows. Generally, the tracking device determines a location and orientation of the tracker receiver in relation to a reference coordinate system (e.g. for electromagnetic trackers, defined by the transmitter). This information is placed directly into the position matrix of the new tracker object. Thus, the parent of the tracker object, formerly the parent of the attached object, becomes the location and orientation of the reference coordinate system of the tracker in the world.

3.6.4 Loading and Saving Object Trees

Object trees are stored in SVE world files, where each object has a section describing its name, which object file to use for its geometry, other attributes of the object, and a list of child objects (complete with their descriptions).

A world file can be loaded as the SVE world object tree, which is rendered at each frame, using the SVE_loadWorld() function.

```
boolean SVE_loadWorld(char *filename);
```

The file identified by the given `filename` is searched for using the world directory list (defined in the `.sve.init` file, or just the current directory if none is defined there). If the file is not found the function returns `FALSE`.

The `SVE_addObjects()` function can be used to load an object tree from a world file and add it to the SVE world tree already in existence. It returns a pointer to the part of the world tree that was just loaded.

```
list SVE_addObjects(char *file, SVE_state state);
```

A world file can be loaded as an object tree separate from the SVE world object tree using the `SVE_loadObjects()` function.

```
list SVE_loadObjects(char *filename, SVE_object parent, SVE_state state);
```

The file will be searched for in the same manner as is done for the `SVE_loadWorld()` function. A `parent` object can be specified, causing the top level objects defined in the world file to become children of the parent object. A world file can contain an initial position for the user and an initial flight speed for when the user fly through the world. These values are stored in the `origin` and `flightSpeed` fields of the given `state` structure respectively (unless `state` is `NULL`).

The current SVE world object tree can be saved to a world file using the `SVE_saveWorld()` function.

```
boolean SVE_saveWorld(char *filename);
```

This function returns `FALSE` if it is unsuccessful. Any object whose geometry has been altered will be updated using its original filename. If two objects share the same geometry description, and only one has been altered, it will be saved to a different file using a variation of the original file name.

An object tree other than the SVE world object tree (although perhaps a sub-tree of the world tree) can be saved to file using the `SVE_saveObjects()` function.

```
boolean SVE_saveObjects(list objectList, char *filename);
```

As with the `SVE_saveWorld()` function, only objects that have changed will be saved, and the function returns `FALSE` if it is unsuccessful.

3.7. Object Appearance

The appearance of an object depends on its position, which is its location, orientation, and scale, and what geometric primitives make up its geometric description. Many geometric primitive types are used in the construction of an object's geometric description, such as simple polyhedra (textured or not textured), polylines, and text.

3.7.1 Object Position

An object's position is represented internally as a 4x4 matrix which is a composite of a translation, a rotation, and a scale transformation. This matrix is used to determine the position of each point in an object's geometry in relationship to the object's parent. The translation component determines where a point at the object's origin will be from the parent object. The rotation and scale components determine where points away for the origin will be in relationship to that origin. It is significant to note that the rotation component is a rotation around the object's origin, rather than the parent object's origin. It is important to note, also, that changes in position of a parent object will cause position changes in its child objects, as well. As an example, if a table top object has four child objects, which are its legs, and it is rotated, then the legs will rotate around the parent's origin, as shown in the following figure.

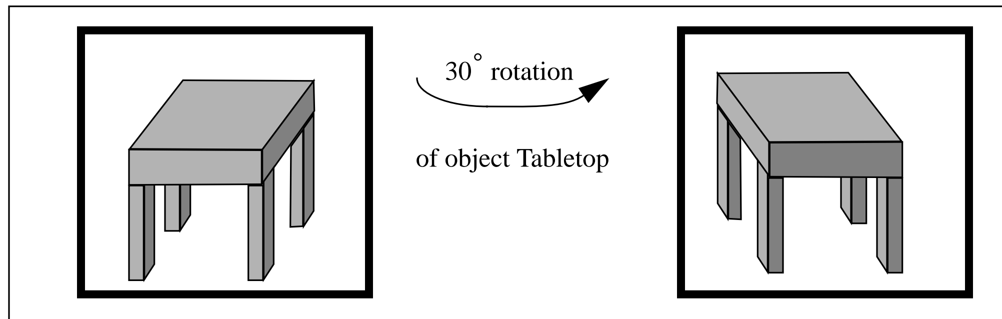


Figure 14. The Propagation of Transformations From Parent to Child Objects.

An object can be re-positioned using any of the following functions:

```
void SVE_moveObject(SVE_object o, SVE_point newOrigin);
```

Moves the object's origin to the given location. The `newOrigin` vector is an offset from the parent object's origin.

```
void SVE_translateObjectGlobal(SVE_object o, float x, float y, float z);
```

Moves the object's origin by the given increments in the X, Y, and Z directions respectively (again, in relationship to its parent).

```
void SVE_rotateObject(SVE_object o, float angle, char axis);
```

Rotates the object around its origin by the given angle in degrees around the X, Y, or Z axis depending on whether the "axis" parameter is 'x', 'y', or 'z' respectively.

```
void SVE_scaleObject(SVE_object o, float xScale, float yScale, float zScale);
```

Scales the object along the X, Y, and Z axis using the values of `xScale`, `yScale`, and `zScale` respectively.

If you wish to set the composite 4x4 matrix directly, you should use the function,

```
void SVE_setNewObjectPosition(SVE_object o, M_matrix newPosition);
```

Other routines are available that translate, rotate and scale an object in *another* object's coordinate system. For example, a lamp contained in a room could be scaled along the *room's* X axis, even though it may not be the lamp's X axis.

The `SVE_translateWRT()` function performs a translation to a new position in the coordinate system of the `coordObj` object, if `absolute` is `TRUE`, or by the vector given in `newPos`, if absolute is `FALSE`. The functions `SVE_moveTo()` and `SVE_moveBy()` correspond to `SVE_translateWRT()` for an `absolute` value of `TRUE` and `FALSE`, respectively. The `SVE_rotateWRT()` function rotates the object in the coordinate system of `coordObj` along the given `axis`, 'x', 'y', or 'z'. The `SVE_scaleWRT()` function scales the object in the coordinate system of `coordObj` along the given `axis`, 'x', 'y', 'z', or 'a' (all axis, which results in a uniform scale).

```
void      SVE_translateWRT(SVE_object obj, SVE_object coordObj,
                           SVE_point newPos, boolean absolute);
void      SVE_moveTo(SVE_object obj, SVE_object coordObj, SVE_point newPos);
void      SVE_moveBy(SVE_object obj, SVE_object coordObj,
                     SVE_point moveVector);
void      SVE_rotateWRT(SVE_object obj, SVE_object coordObj,
                        float theDegrees, char axis);
void      SVE_scaleWRT(SVE_object obj, SVE_object coordObj,
                       float scaleVal, char axis);
```

If for some reason, you change an object's position matrix without using one of the functions described above, you *must* indicate that to the SVE system that a change has occurred, otherwise strange things may happen.

```
void SVE_reCalculateWorldMatrix(SVE_object o);
```

3.7.2 Object Geometry

Most SVE objects have a geometric appearance defined for it which can include polyhedra, polylines, text, and light sources. This geometry can be read from an object description file, as described above, or can be generated dynamically. An object can actually have many potential geometries defined for it, where one is chosen given the distance between the viewer and the object's origin. In addition to an object's regular geometry, a highlight geometry can be defined which will replace the object's regular geometry when it is being highlighted (i.e. its "`highlight`" flag is `TRUE`).

A geometry is defined as a list of primitives, each of which can be a polyhedron, a textured polyhedron, a polyline, text, or a light source. A polyhedron primitive is simply a collection of polygon faces which all use a subset of a list of vertices and normals. A textured polyhedron is a special polyhedron which contains texture vertices information used to map textures to the polygon faces. A polyline primitive is collection of continuous lines which all use a subset of a list of vertices. A text primitive is a possibly multi-line block of text which has a location, orientation, and scale in relationship to the object it is a part of. The light source primitive is a light definition, which has either a location or a direction, and color attributes. Each visible primitive can have a highlighted material associated with it which, if no highlight geometry is specified, will override the primitive's usual material when the object is being highlighted. The following SVE object file defines one instance of each of these primitive types. This file can be found as `all_primitives.object` in the objects directory.

Source xii: all_primitives.object, an Example of Each Primitive Used in the SVE Library.

```
Simple Virtual Primitive File Format  version 1.1
number of components: 5

component 1 type: light
Data of component 1:
no of attributes:
2
color      1.0  1.0   1.0
position   0.0  5.0 -10.0

component 2 type: polyhedron
Data of component 2:
```

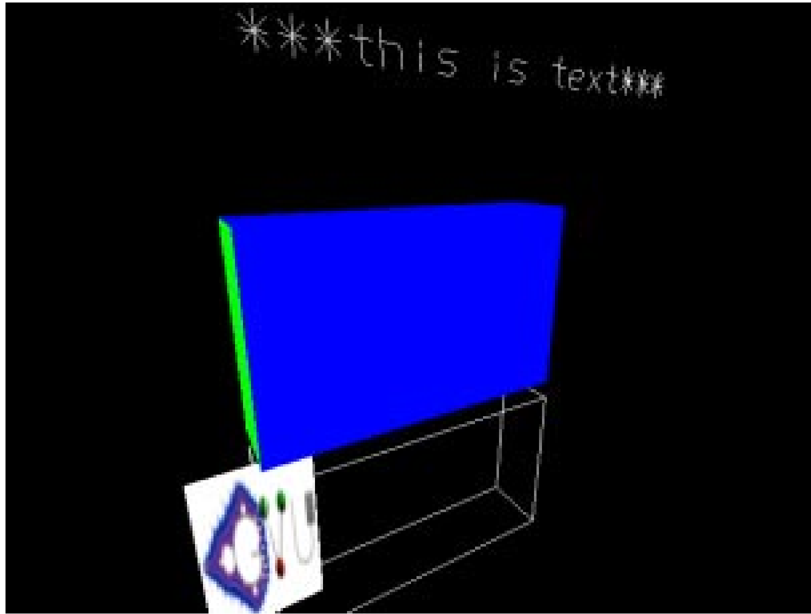


Figure 15. Snap Shot of an Object Containing All Possible Primitive Types.

```
no of vertices:
8
vertices: x y z
0      0      0
0      1.0    0
1.6    1.0    0
1.6    0      0
0      0      0.35
0      1.0    0.35
1.6    1.0    0.35
1.6    0      0.35
no of faces:
6
faces: R G B #_of_vertices v1 v2 v3 ...
255   0    0 4 0 1 2 3
  0 255    0 4 0 4 5 1
  0   0  255 4 1 5 6 2
255   0    0 4 2 6 7 3
  0 255    0 4 0 3 7 4
  0   0  255 4 4 7 6 5

component 3 type: polyline
Data of component 3:
no of vertices:
8
vertices: x y z
0      -0.1   0
0      -1     0
1.6    -1.0   0
1.6    -0.1   0
0      -0.1   0.35
0      -1.0   0.35
1.6    -1.0   0.35
1.6    -0.1   0.35
no of polylines:
6
polylines: R G B #_of_vertices v1 v2 v3 ...
255 255 255 4 0 1 2 3
255 255 255 4 0 1 5 4
255 255 255 4 1 2 6 5
255 255 255 4 2 3 7 6
```

```
255 255 255 4 3 0 4 7
255 255 255 4 4 5 6 7

component 4 type: text
Data of component 4:
transformation matrix:
0.4 0   0    0
0   0.4 0    0
0   0   0.3  0
0.1 1.5 0.36 1
no of lines:
1
***this is text***

component 5 type: textured_polyhedron
Data of component 5:
image file:      gvu.rgb
repeat texture:  TRUE
blending:        TRUE
no of vertices:
4
vertices: x y z
-0.3 -1.2 0.0
-0.3 -0.2 0.0
 0.3 -0.2 0.0
 0.3 -1.2 0.0
no of texture vertices:
4
texture vertices: u v
0 0
0 1
1 1
1 0
no of faces:
1
faces: R G B #_of_vertices v1 v2 v3 ...[t1 t2 t3 ...]
255 255 255 4 3 2 1 0 3 2 1 0
```

A snapshot of the primitives defined in this file is shown in Figure 12. "Snap Shot of an Object Containing All Possible Primitive Types." on page 63. Note that these primitives make up one SVE object.

The SVE system maintains a list of geometry definitions indexed by a string name. In the case of geometries read from file, the name of the geometry is the full path name of the file. This indexing method allows SVE to re-use a geometry definition if more than one object uses it. The mechanism to deal with duplicate geometries is automatic, and prevents unnecessary file reading and preparation for rendering. The disadvantage to this approach is that when one object wishes to change its geometry, it must get a fresh copy first to prevent other objects that use the same geometry to be affected. If a SVE application changes the geometry using a SVE function, the SVE system ensures that the object has a unique geometry before making the change. If, however, the application needs to change the geometry definition by setting a value in the geometry structure directly, it should first call this function to ensure that the object has a unique geometry:

```
void SVE_initGeometryChange(SVE_geometry *geometry, SVE_object object);
```

(The `object` parameter is the SVE object using the geometry if know, or NULL otherwise. It is used to ensure that the object's boundaries are recalculated.)

Object Geometries

As mention before, objects can have no geometry, resulting in the object having no visual presence in the virtual world, one geometry, which defines the object's appearance in the virtual world, or many geometries, where one geometry will be chosen before each rendering of the scene to use as the object's appearance. Unless otherwise specified, any routines that deal with an object's geometry when an object has more than one geometry defined will use the geometry chosen for the current (or previous) rendering phase.

When defined in an SVE world file, the object's geometry is given by the file given on the "primitives file:" line. More than one geometry can be specified by giving a list of "primitives file:" lines, with optional valid range specifications. In this case, the first geometry on the list which is valid is used as the object's geometry for the current rendering phase. A geometry is valid if no range specification are given or if the viewer is within the given valid range specification for that geometry (the ranges are given in meters). For example, the following object entry in a world file defines a cube that has different geometries (close_cube.object if the eye point is within 3 meters, medium_cube.object if the eye point is between 3 and 5 meters, and far_cube.object if the eye point is farther than 5 meters):

```
object name: cube
primitives file: close_cube.object valid to 3
primitives file: medium_cube.object valid from 3 to 5
primitives file: far_cube.object valid from 5
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 -3 1
other attributes: 0
number of children: 0
```

In the next few sections, we will show how object geometries can be constructed by the application at run-time. First, a primitive is constructed out of faces, lines, text, or a light definition. Next, the primitive is added to a geometry definition. Finally, the geometry definition is added to the object description. A geometry definition is added to that list of geometries of an object using the `SVE_addObjectGeometry()` function.

```
void SVE_addObjectGeometry(SVE_object o, SVE_geometry newGeometry,
                           float minDistance, float maxDistance);
```

The geometry is added to the end of the object's list of geometries. If `minDistance` is not negative, then the geometry is valid if the viewer is further than the given distance away (in meters using the world coordinate system). Similarly, if `maxDistance` is not negative, then the geometry is valid if the viewer is closer than the given distance. If both `minDistance` and `maxDistance` are positive integers, then they are used as the minimum and maximum values, respectively, of the range in which the geometry is valid.

A geometry in an object's geometry list can be removed from the list with the `SVE_removeObjectGeometry()` function.

```
void SVE_removeObjectGeometry(SVE_object o, SVE_geometry geometry);
```

Primitive Construction

The most straight forward method to defining an object's geometry is to use a series of function calls that define the primitives of a geometry. This method is similar to the method used by a GL or OpenGL application to render geometry primitives.

The first step of this method is to indication which primitive type you wish to create. The possible primitive types are: `POLYHEDRON`, `TEXTURED_POLYHEDRON`, `LINE`, `TEXT`, and `LIGHT`. The appropriate constant should be used for the `type` parameter of the `SVE_beginPrimitive()` function.

```
void SVE_beginPrimitive(SVE_primitiveType type);
```

This function must be called before any of the other primitive construction functions can be called. When the primitive has been defined by the functions described below, the construction is ended with the `SVE_endPrimitive()` function.

```
SVE_primitive SVE_endPrimitive();
```

This function returns a reference to the primitive that was constructed. The next section describes how to add the primitive to an object's geometry.

Between the begin and end functions, the attributes of the primitive are specified. The material and highlight material of the primitive can be specified with the `SVE_primitiveMaterial()` and `SVE_primitiveHighlightMaterial()` functions. Note that if an object has a material defined for it, that material will override the primitive material. (See 3.8. "Colors and materials" on page 74 for details on creating SVE materials.)

```
void SVE_primitiveMaterial(SVE_material material);
void SVE_primitiveHighlightMaterial(SVE_material material);
```

The line width value for each primitive (for primitives containing lines) can be specified with the `SVE_primitiveLineWidth()` function.

```
void SVE_primitiveLineWidth(int lineWidth);
```

If the primitive is a TEXT primitive, the text string is specified with the `SVE_primitiveText()` function.

```
void SVE_primitiveText(char *text);
```

The text's position, scale and rotation offsets from the object it is part of can be specified using the following functions.

```
void SVE_primitiveTextPosition(float x, float y, float z);
void SVE_primitiveTextScale(float scale);
void SVE_primitiveTextRotation(float xrot, float yrot, float zrot);
```

If the primitive is a POLYHEDRON, TEXTURE_POLYHEDRON, or LINE primitive, there are two methods to specifying the vertices of the primitive. One method is to simply begin a face or line, give the vertices in order, and then end the face or line. The other method (which is more efficient) is to first give a list of points (locations), each with a unique index value; a list of normals (if needed), each with a unique index value; and a list of texture coordinates (if needed), each with a unique index value. Then a face or line is constructed by beginning the construction, specifying each vertex in order by given the index of the vertex point (location), normal (if needed), and texture coordinate (if needed), and then ending the face or line. Indexes that are not to be used should be -1. Both of these methods can be used to construct as many faces or lines required to make up a polyhedron or polyline primitive.

A face is begun with the `SVE_beginPrimitiveFace()` function (which returns a unique face index to identify it apart from the other faces) and ended with the `SVE_endPrimitiveFace()` function (which returns the face definition, although it would usually be ignored, as it is automatically added to the primitive being defined).

```
int SVE_beginPrimitiveFace();
SVE_facePtr SVE_endPrimitiveFace();
```

A line is begun with the `SVE_beginPrimitiveLine()` function (which also returns a unique line index to identify it apart from the other lines) and ended with the `SVE_endPrimitiveLine()` function (which returns the line definition, although it would usually be ignored, as it is automatically added to the primitive being defined).

```
int SVE_beginPrimitiveLine();
SVE_facePtr SVE_endPrimitiveLine();
```

A closed line (where a line is automatically drawn from the last vertex to the first vertex) can be begun with the following function:

```
int SVE_beginPrimitiveClosedLine();
```

For the easy method, between the begin and end functions, the `SVE_primitiveVertex()` function is called for each vertex in the face or line. The vertices should be given in order (counterclockwise around a face), so that, if no normal is given, a correct normal can be calculated for the face. This is demonstrated by Figure 16. "The Normal Resulting From Different Vertex Orderings." on page 69. The reference to the vertex is returned in case it is needed.

```
SVE_vertexPtr SVE_primitiveVertex(float x, float y, float z);
```

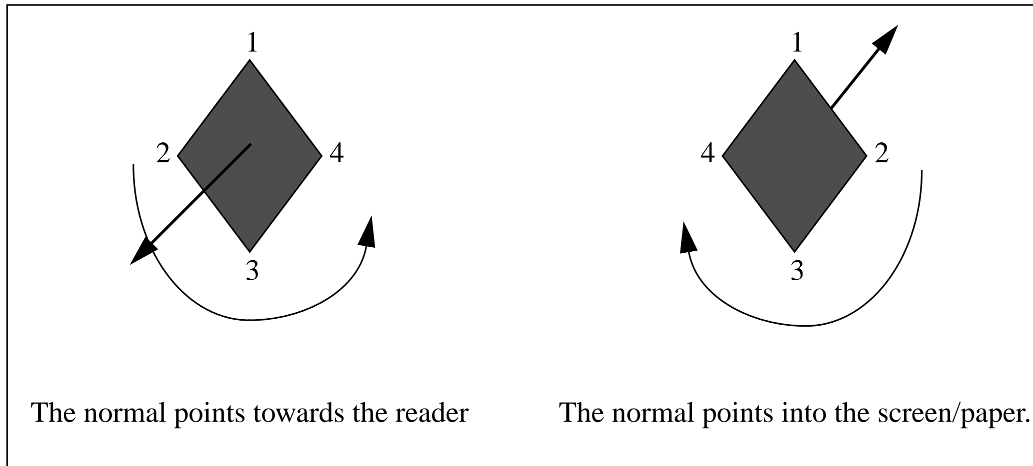


Figure 16. The Normal Resulting From Different Vertex Orderings.

If a normal or texture coordinate needs to be defined for that vertex, they should be defined before the vertex specification. The function takes an index value as the first parameter. If the application does not wish to give a unique index, it should give -1, instead, and an index will be generated automatically (and returned by the function). The functions are:

```
int SVE_primitiveNormal(int index, float x, float y, float z);
int SVE_primitiveTexCoord(int index, float s, float t);
```

These specifications are persistent. If many vertices have the same normal, only one call to `SVE_primitiveNormal()` is needed before all of the vertices specifications.

The following example routine demonstrates how a primitive can be constructed that is a rectangle that floats behind a text string (given). First, the size of the text (when it is displayed) is determined using the `SVE_getTextExtent()` function. Then the rectangle's normal is given, and then the four vertices (in counter-clockwise order). Finally, the rectangle's material is set, and the face and primitive generation is concluded.

Source xiii: Primitive Construction Example

```
SVE_primitive createLabelBacking(char *label, SVE_material faceMaterial)
/*
  Create and return a label primitive, which is a square that
  closely surrounds the label text.
*/
{
   SVE_primitive labelPrim;
   float height = 1;
   float width = 1;
   float originX = 0;
   float originY = 0;

   SVE_getTextExtent(label, &originX, &originY, &height, &width);

   /* Construct the face that will surround the text a little behind it. */
   SVE_beginPrimitive(POLYHEDRON);

   SVE_beginPrimitiveFace();

   /* normal */
   SVE_primitiveNormal(-1, 0.0, 0.0, 1.0);

   /* vertices */
   SVE_primitiveVertex(originX+width+0.1, originY-0.1, -0.1);
   SVE_primitiveVertex(originX+width+0.1, originY+height+0.1, -0.1);
```

```
        SVE_primitiveVertex(originX-0.1, originY+height+0.1, -0.1);
        SVE_primitiveVertex(originX-0.1, originY-0.1, -0.1);

        SVE_primitiveMaterial(faceMaterial);

        SVE_endPrimitiveFace();

        labelPrim = SVE_endPrimitive();

        return(labelPrim);
    }
```

In the more efficient method, the vertices of a face or line are specified by giving a point, normal, and texture coordinate index. The index refers to the index value given to the functions `SVE_primitivePoint()`, `SVE_primitiveNormal()`, and `SVE_primitiveCoord()` respectively. In each of these functions, the index can be -1, which causes the function to generate a unique index and return it. When the vertex is specified, the normal and/or texture coordinate index can be -1 if a normal and/or texture coordinate value is not associated with that vertex. The normal and texture coordinate specifications have been described above. The point specification is done with the `SVE_primitivePoint()` function.

```
int SVE_primitivePoint(int index, float x, float y, float z);
```

The vertex specification is done with the `SVE_primitiveVertexIndexes()` function.

```
SVE_vertexPtr SVE_primitiveVertexIndexes(int pointIndex, int normalIndex,
                                         int textureIndex);
```

The points, normals, and texture coordinates do not need to be specified before the face or line is begun, but each point, normal, and texture coordinate needs to be specified after the `SVE_beginPrimitive()`, and before the `SVE_primitiveVertexIndexes()` function that refers to it.

For example, the previous example could be changed to use indexes, resulting in the following code.

```
/* Construct the face that will surround the text a little behind it. */
SVE_beginPrimitive(POLYHEDRON);

/* normal */
SVE_primitiveNormal(0, 0.0, 0.0, 1.0);

/* vertices */
SVE_primitivePoint(0, originX+width+0.1, originY-0.1, -0.1);
SVE_primitivePoint(1, originX-0.1, originY-0.1, -0.1);
SVE_primitivePoint(2, originX-0.1, originY+height+0.1, -0.1);
SVE_primitivePoint(3, originX+width+0.1, originY+height+0.1, -0.1);

/* indices for button polygon */
SVE_beginPrimitiveFace();
SVE_primitiveVertexIndexes(0, 0, -1);
SVE_primitiveVertexIndexes(3, 0, -1);
SVE_primitiveVertexIndexes(2, 0, -1);
SVE_primitiveVertexIndexes(1, 0, -1);

SVE_primitiveMaterial(faceMaterial);

SVE_endPrimitiveFace();

labelPrim = SVE_endPrimitive();
```

Primitive Copying

A primitive can be copied using the `SVE_getPrimitiveCopy()` function. It is copied in its entirety.

```
SVE_primitive SVE_getPrimitiveCopy(SVE_primitive primitive);
```

Primitive Incorporation

Once a primitive has been constructed or copied, it needs to be incorporated into an object's geometry or highlight geometry. A primitive can be added to an object's geometry with the `SVE_addPrimitiveToObject()` function. This function will add the primitive to the current geometry of the object, which is the geometry chosen for the object in the current (or previous) rendering phase.

```
void SVE_addPrimitiveToObject(SVE_object o, SVE_primitive primitive);
```

A primitive can be added to an object's highlighted geometry with the `SVE_addHighlightPrimitiveToObject()` function.

```
void SVE_addHighlightPrimitiveToObject(SVE_object o, SVE_primitive primitive);
```

A primitive can be added to one of the other geometries which are defined for an object, or to a geometry definition that has not yet been assigned to an object (described below) using the `SVE_addPrimitiveToGeometry()` function.

```
void SVE_addPrimitiveToGeometry(SVE_geometry *geometry,
                                SVE_primitive primitive, SVE_object object);
```

Note that, if `geometry` is `NULL`, this routine will create a new geometry (using the empty string as its name, and then generating a new name if there are other geometries identified by the empty string). The `object` parameter is used to indicate which object needs to recalculate its boundaries with the geometry change. It can be `NULL` if the object is not known or the geometry is not attached to an object.

Independent Geometries

It is possible to create a geometry that is separate from any object. The geometry will not be rendered, as only object geometries are rendered, but it can be generated, stored, and then switched with an object's current geometry at an appropriate time to cause a quick geometry change.

As mentioned before, geometries are identified by unique string names, usually the full path file name of the file from which it was read. If an application generates a geometry from scratch, it should give it a name to identify it, and use that name whenever an instance of that geometry should be used.

A geometry is created an initialized with the `SVE_createGeometry()` function.

```
SVE_geometry SVE_createGeometry(char *name);
```

It is possible to find out if the geometry is an instance of another geometry that has already been created. If the `objectLinks` field of the geometry structure returned is greater than 1, there are other references to this geometry around. A unique copy of the geometry can be obtained (in any case) using the function `SVE_initGeometryChange()` described earlier.

Any geometry that has been created can be found using its name with the `SVE_findGeometryInRepository()` function.

```
SVE_geometry SVE_findGeometryInRepository(char *name);
```

Texture Swapping

An object can become a kind of "movie screen" by way of the texture swapping routines. Texture swapping allows for all textured polygons of an object to use one texture, which can be changed regularly for a crude sort of "texture animation". The additional textures are specified through a call to `SVE_defineObjectTextures()` or `SVE_defineObjectTextureList()`. Only one texture is "active" at a time for an object, so the application should define a callback that will determine when to switch textures. This can be as simple as stepping through the textures defined, or something more complex such as a swap based on Level of Detail.

```
int SVE_defineObjectTextures(SVE_object o, int textureMode, int n, ...);
int SVE_defineObjectTextureList(SVE_object o, int textureMode,
                                int n, char* textureNames[]);
```

For the function SVE_defineObjectTextures(), the texture mode to be used for all of the textures is given, then the number of textures, and the file names of the textures, given one by one. For the function SVE_defineObjectTextureList(), the file names of the textures are stored in an array, which is then passed into the function.

The texture swap callback function, which is called every frame before the frame is rendered, can be set with the SVE_setTextureSwapCallback() function.

```
void SVE_setTextureSwapCallback(SVE_object o, SVE_textureSwapFunctionPtr f);
```

The texture swap callback function should be of the following form:

```
typedef long SVE_textureSwapFunction(SVE_object object);
```

It should return the index (in the list of textures or texture array given in the define function) of the texture which should be used. (The array of textures is 0 based.) A default texture swapping function, which swaps the textures 60 times per second, is available. It is called SVE_defaultTextureSwap().

```
long SVE_defaultTextureSwap(SVE_object o, SVE_state state);
```

Miscellaneous Object Attributes

An object can be given all one color or material using the SVE_setObjectMaterial() function.

```
void SVE_setObjectMaterial(SVE_object o, SVE_material material);
```

The given material overrides the colors and materials of the primitives that make up the object's geometry. If the material parameter is NULL, the object will have no overall material, and the object's geometry will once again use its own colors and materials.

An object's highlight color, which is used to color an entire object when it is highlighted, and no highlight geometry is specified, can be set using the SVE_setObjectHighlightMaterial() function.

```
void SVE_setObjectHighlightMaterial(SVE_object o, SVE_material material,
                                    boolean includeText);
```

The includeText flag indicates whether the material should apply to any text in the object or not. It should be noted that if the text is the same color as the other primitives of the object, it will be unreadable if it is in front of those primitives. If the material parameter is NULL, the objects' primitives will have no highlight material.

It is often the case that an object contains many graphical primitives, and one text primitive as an identifying label. Often that label needs to be changed, and it would be tedious to search for the text primitive in an object's geometry and go through the steps of changing the text, making sure that it happens correctly. As a convenience, the SVE_changeText() function will change the first text primitive it finds in an object to the text given, freeing the old text. It returns TRUE if it was successful in finding a text primitive, and changed it.

```
boolean SVE_changeText(SVE_object o, char *newText);
```

3.7.3 Object Boundaries

Bounding volumes are calculated automatically for each SVE object which surround just the object's primitives (known as the "primitives boundaries"), and which surround the object's primitives and object's child objects (known as the "object boundaries"). These boundaries are used to determine if an object can be seen by the viewer, and for crude object collision detection. Although the boundaries can be a sphere or a box, the SVE library generates box boundaries, and routines that use the boundaries are optimized for box boundaries.

The object's primitives boundaries can be obtained using the SVE_getPrimitiveBoundaries() function. The object boundaries can be obtained using the SVE_getObjectBoundaries() function. The coordinates of the boundaries are given in terms of the object's coordinate system.

```
SVE_boundaries *SVE_getPrimitiveBoundaries(SVE_object o);
SVE_boundaries *SVE_getObjectBoundaries(SVE_object o);
```

The SVE_boundaries structure contains the following fields. If `hasSphere` is `TRUE`, the `sphereOrigin` and `sphereRadius` fields represent the origin and radius of the sphere boundary. If `hasBox` is `TRUE`, then the `boxVertex1` and `boxVertex2` fields represent two corners of the bounding box which are farthest from each other.

```
typedef struct SVE_boundaries{
     boolean   hasSphere;
     SVE_point sphereOrigin;
     float     sphereRadius;
     boolean   hasBox;
     SVE_point boxVertex1;
     SVE_point boxVertex2;
} SVE_boundaries;
```

The application can test to see if the boundaries of two object collide using the `SVE_objectBoundsCollide()` function. The `SVE_objectBoundsInBounds()` function indicates if an object's boundary is contained entirely inside another object's boundary. For each of these functions, the caller can indicate whether the object's children should be considered (using the "object boundaries") or not (using the "primitive boundaries").

```
boolean SVE_objectBoundsCollide(SVE_object o1, SVE_object o2,
                                boolean includeChildren);
boolean SVE_objectBoundsInBounds(SVE_object o1, SVE_object o2,
                                 boolean includeChildren);
```

The application can test to see if a point (in world coordinates) is inside an object's primitive boundaries using the `SVE_pointHitObjectPrimitives()` function, or inside the object boundaries using the `SVE_pointHitObject()` function. Given an object tree and a point in world coordinates, the `SVE_objectPointHit()` will return the first object (in a depth-first search) whose primitive boundaries contains the point. For all of these functions, the given margin is a margin of error, where a positive margin will effectively enlarge the boundaries, and a negative margin will shrink the boundaries.

```
boolean SVE_pointHitObjectPrimitives(SVE_object obj, SVE_point pt,
                                     float margin);
boolean SVE_pointHitObject(SVE_object obj, SVE_point pt, float margin);
SVE_object SVE_objectPointHit(list objectList, SVE_point pt,
                              float margin, boolean onlySelectable);
```

For debugging purposes, the primitives and object boundaries of all objects in an object tree can be drawn (in a frame callback, for example) using the `SVE_drawBoundaries()` function.

```
void SVE_drawBoundaries(list objects);
```

3.8. Colors and materials

The SVE system maintains a list of defined materials, each of which is identified by an integer index and a unique name. Most of the time, internal SVE structures will use the index to refer to a material. The names are usually used when reading in a material file which defines materials, and an object description file, where the material names are used to indicate materials for graphical primitives. A material can be retrieve from either index or name using the appropriate function call:

```
SVE_material SVE_getMaterialByName(char *name);
SVE_material SVE_getMaterialByIndex(int index);
```

If an application only deals in red, green, and blue color definitions, it is possible to get a material definition just from the color values. The function will use a material definition for the given color or create one if it doesn't exist already. The color values represent the ambient and diffuse color of the material. The function to use for this purpose is `SVE_getColorMaterial()`.

```
SVE_material SVE_getColorMaterial(float color[3]);
```

Any material definition contains values for ambient and diffuse colors, transparency (alpha), specular and emission colors, a shininess value, and an optional texture map. The material definition structure looks like this:

```
typedef struct SVE_materialStruct {
    char *name;
    int index;
    float ambient[3];
    float bwAmbient;
    float diffuse[3];
    float bwDiffuse;
    float specular[3];
    float bwSpecular;
    float emission[3];
    float bwEmission;
    float shininess;
    float alpha;
    int   textureMode;
    long  textureEnv;
    char *texture;
    char   *textureFilename;
    boolean transparentTexture;
} SVE_materialStruct;
typedef struct SVE_materialStruct *SVE_material;
```

A material definition can be created from scratch and initialized using the following function. Note that if another material exists by the given name, it's definition will be returned.

```
SVE_material SVE_createMaterial(char *name);
```

Values in a material definition should be changed by an appropriate function. The following functions change the various color values. Note that these functions also calculate the black and white equivalent for each color, and store it for black and white rendering.

```
void SVE_setMaterialAmbient(SVE_material material, float ambient[3]);
void SVE_setMaterialDiffuse(SVE_material material, float diffuse[3]);
void SVE_setMaterialSpecular(SVE_material material, float specular[3]);
void SVE_setMaterialEmission(SVE_material material, float emission[3]);
```

The following functions change the shininess and transparency (alpha) values respectively.

```
void SVE_setMaterialShininess(SVE_material material, float shininess);
void SVE_setMaterialAlpha(SVE_material material, float alpha);
```

The `SVE_setMaterialTexture()` function sets the texture file name, mode, and environment. The mode options are: `SVE_TEXTURE_REPEAT`, `SVE_TEXTURE_CLAMP`, `SVE_TEXTURE_GREYSCALE`. The environment modes are given in the following table, along with a description of the effect applying the texture will have on a face.

```
void SVE_setMaterialTexture(SVE_material material, char *texture,
                            int textureMode, long textureEnv);
```

Table 11: Texture Environment Modes

Mode	Component type and order	Affect on face
SVE_TEXTURE_INTENSITY	Intensity	The RGB color of the polyhedron is multiplied by the texture pixel value at each pixel.
SVE_TEXTURE_INTENSITY_ALPHA	Intensity, Alpha	The RGB color of the polyhedron is multiplied by the intensity value from the texture at each pixel, the alpha value of the polyhedron is multiplied by the alpha value from the texture at each pixel.
SVE_TEXTURE_RGB	Red, Green, Blue	Texture RGB color overrides polyhedron. Alpha of polyhedron is unaffected.
SVE_TEXTURE_RGB_LIGHTING	Red, Green, Blue	The RGB color of the texture is multiplied by the RGB color of the polyhedron.
SVE_TEXTURE_RGB_ALPHA	Red, Green, Blue, Alpha	The RGB color of the texture and the RBG color of the polyhedron are combined proportionally to the texture's alpha value (Texture alpha value of 0 results in 100% polyhedron color, value of 1 results in 100% texture color). The polyhedron's alpha value is unchanged.
SVE_TEXTURE_RGB_ALPHA_LIGHTING	Red, Green, Blue, Alpha	The RGB color and alpha values are multiplied by the RGB and alpha values of the texture for each pixel.
SVE_TEXTURE_DEFAULT	One of the above	Used to specify that the default for the number of channels contained in the texture. The SVE_TEXTURE_RGB_LIGHTING and SVE_TEXTURE_RGB_ALPHA_LIGHTING modes are the defaults for 3 and 4 component textures respectively.

As a short cut to setting a material's ambient and diffuse colors, and optionally the alpha value, these functions are provided.

```
void SVE_setMaterial3Color(SVE_material material, float color[3]);
void SVE_setMaterial4Color(SVE_material material, float color[4]);
```

3.9. Lighting and Shading

By default, SVE uses flat shading to render polygons using one color value, which represents the ambient color of the polygon's material. This results in a cartoonish looking scene with no smooth surfaces. It is simple to provide a more realistic scene by using lighting and smooth shading effects, although at a cost of rendering speed.

Lights can be defined as part of an SVE object's description, and then placed in the world object tree. Lights can be directional (parallel light rays), or can radiate from a source position. Using the defined lights, a renderer will take into account a polygon's material's diffuse and specular attributes, as well as the ambient color attribute, when rendering the polygon. An example light definition is given below. The SVE object file actually defines two light sources, the first one is a local light source which is located behind, up to the right (location is 10, 10, 10) and the second one is a light at infinity, like the sun. It is positioned straight above the origin.

Source xiv: Example SVE object with lights

```
Simple Virtual Primitive File Format  version 1.1
number of components: 2

component 1 type: light
Data of component 1:
no of attributes:
3
color     0.8   0.8  0.8
local-light TRUE
position 10.0 10.0 10.0

component 2 type: light
Data of component 2:
no of attributes:
2
color     0.8 0.8 0.8
position  0.0 1.0 0.0
```

Although lights are defined in the world object tree, the renderer will not actually use the lights unless told by the application to do so using the configuration flag. If lighting effects are desired, then the configuration flag should include `SVE_LIGHTING`.

Currently, the SVE library allows for one smooth shading effect, called Gouraud shading. Gouraud shading uses the normal at each vertex of a polygon to determine the color of that vertex (using the current lighting configuration), then linearly interpolates the shades between the vertices to determine the pixel color. When two polygons share a vertex and normal, therefore, the area around the vertex will be shaded such that the vertex and its edges will not be noticeable (except in a silhouette). When a vertex of a polygon is not given a normal value, SVE calculates the normal to the polygon (assuming the polygon's vertices are given in counterclockwise order), and uses that for each vertex of the polygon. Edges of these polygons will be visible due to shading differences. Gouraud shading effects can be "turned on" by including `SVE_GOURAUD` in the configuration flag.

Here is an example application that uses lights and Gouraud shading. Note that it is our rotating cube example, but modified to use lighting and Gouraud shading effects.

Source xv: example4, Application Using Light.

```
/************************************************************************
 * Example4 (sve module)
 *
 * This is the example from the lighting and shading section of the SVE
 * manual.
 *
 * This example has one light source shining on its world. We'll have a
 * cube spinning around to the difference in light. The default key's as
 * described in the SVE manual still work.
 ***********************************************************************/
```

```
#include "sve.h"

/*********************************************************************
 * Find the cube in the world and spin it around the y-axis.
 *********************************************************************/
SVE_status rotateCube(SVE_state state)
{
   SVE_object cube;

   cube = SVE_findWorldObject("cube");
   SVE_rotateObject(cube, 3, 'y');
}


void main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL | SVE_LIT_GOURAUD;

/*********************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use.
 * Since we use light in this example we also need to use SVE_LIT_GOURAUD.
 *********************************************************************/
   printf("Starting application.\n");
   SVE_init("Example4 (sve)", config, &argc, argv);

/*********************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 *********************************************************************/
   if(!SVE_loadWorld("example4.world"))
   {
      printf("error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/*********************************************************************
 * Tell SVE to call the function rotateCube before rendering a new frame.
 *********************************************************************/
   SVE_addAnimationCallback(rotateCube);

/*********************************************************************
 * SVE will take over control of the program until it is finnished.
 *********************************************************************/
   printf("Beginning event loop.\n");
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
}
```

As you can see the only change in this example from the previous ones is the line

```
config = SVE_NORMAL | SVE_LIT_GOURAUD
```

It tells SVE to use lighting and Gouraud shading effects instead of no lights and flat shading. The configuration `SVE_LIT_GOURAUD` is short hand for the combination of `SVE_LIGHTING` and `SVE_GOURAUD`. The placement and definition of the light appears elsewhere. Source xvi: shows the world file used for this example.

Source xvi: World File Used in Example4.

```
Simple Virtual Object File Format version 1.0
number of objects: 2

object name: lightsource
primitives file: light.object
transformation matrix:
1 0 0 0
```

```
0 1 0 0
0 0 1 0
0 0 0 1
other attributes: 0
number of children: 1

object name: meadow
primitives file: plane.object
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
other attributes: 0
number of children: 0

object name: cube
primitives file: cube.object
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
0 2 -0.5 1
other attributes: 0
number of children: 0
```

The lights themselves are defined in the file `light.object` (given above), and are loaded when you run the application. This makes it easier to play around with the light until it is placed correctly.

3.10. Sound

3.10.1 Audio support

Auditory feedback is certainly important and sometimes even a necessary part of the user interface. Apart from augmenting engagement (as in computer games), short sounds can be used as auditory cues for occurring events. This kind of feedback is especially useful while performing tasks with the trackers (e.g. grabbing objects from a pile).

The SVE library provides many functions that deal with audio. It can load multiple AIFF files and play them simultaneously. SVE can play four samples at the same time, which is a limit of the hardware. SVE itself can hold more than four samples in memory at the same time. This makes it possible to load all the necessary sound samples into memory when the program is initializing. Once the program is initialized and the sample is required, all that the program has to do is tell SVE to play the sample. The advantage to this method is that no time is lost between the point where the sound is required and the point where it starts playing.

The following source code is an example of loading sound files and playing them. Eight sound files are loaded. One is repeated continuously, the others are played when the user presses one of the number keys.

Source xvii: Sound Example

```
/*
   sound.c

   SVE sound example

   Plays a sound continuously, and can play one of 7 other sounds
   depending on which key (1-7) is pressed.

*/

#include "sve.h"

int sound[8];

SVE_status KeypressCallback(SVE_state state)
{
    char keyPressed;

    if (state->eventType == SVE_KEY_PRESS) {
        keyPressed = ((SVE_keyEvent *)state->eventData)->keyVal;
        if((keyPressed > '0') && (keyPressed < '8')) {
            SVE_audioReplaySound(sound[keyPressed-'0'], FALSE);
            return(EVENT_CONSUMED);
        }
    }
    return(EVENT_IGNORED);
}


void main(int argc, char *argv[])
{
    SVE_config config = SVE_NORMAL;

    printf("Starting application\n");

    SVE_init("Example1 (sve)", config, &argc, argv);

    sound[0] = SVE_audioOpenSound("belltree_up2.aiff");
    sound[1] = SVE_audioOpenSound("harp_glis.Cmj.aiff");
    sound[2] = SVE_audioOpenSound("jar.aiff");
    sound[3] = SVE_audioOpenSound("orch_hit.aiff");
    sound[4] = SVE_audioOpenSound("slinky_slap.aiff");
    sound[5] = SVE_audioOpenSound("stereo_uparp.aiff");
    sound[6] = SVE_audioOpenSound("tag2.aiff");
    sound[7] = SVE_audioOpenSound("tag3.aiff");
```

```
      SVE_audioReplaySound(sound[0], TRUE);

      if(!SVE_loadWorld("hello_world.world"))
      {
         printf("error occured during SVE_loadWorld, exiting \n");
         SVE_done(); /* stop process */
      } /* if */

      SVE_registerCallback(SVE_KEY_PRESS, KeypressCallback);

      printf("Beginning event loop\n");
      SVE_beginEventLoop();

      printf("Done -- Have a nice day.\n");
      SVE_done();
   }
```

The function,

```
int SVE_audioOpenSound(char *filename);
```

loads a sound file with the given file name, and returns a number which the other audio functions use to refer to it. If there was a problem loading the file, the function returns -1.

The following function causes a sound file to be played. The `sampleNo` parameter is the number returned by the `SVE_audioOpenSound` function when the sound was loaded. The "`repeat`" parameter indicates if the sound is to be repeated continuously (`TRUE`), or if it stops after playing once (`FALSE`).

```
void SVE_audioReplaySound(int SampleNo, boolean repeat);
```

Sounds can be interrupted using the function,

```
void SVE_audioStopSound(int SampleNo);
```

The following function will indicate whether a sound is finished or not (returns `FALSE` or `TRUE` respectively).

```
boolean SVE_audioCheckSound(int SampleNo);
```

The volume of the left and right audio signals can be set using the function,

```
void SVE_audioSetVolume(int left, int right);
```

The current volume levels can be retrieve with the function,

```
void SVE_audioGetVolume(int *left, int *right);
```

Sound volume represented by an integer between 0 and 255 (inclusive). It is a logarithmic scale.

Sound files are recorded with the *recordaiff* application. This application can sample sounds at various rates, durations, numbers of channels etc. The following function was used in a speech annotator to record 7-second samples with the lowest quality:

```
void record(char *filename)
{
   char command[200];

   sprintf(command, "recordaiff -nchannels 1 -rate 8000 -time 7 %s &",
          filename);
   system(command);
}
```

3.10.2 Spatial audio support

(NOTE: This configuration is not currently supported. Updates to return this functionality are planned for the future.)

Some limited functions allow the spatialization of one single sound source, using the SPARCstation nagel.cc.gatech.edu as a server.The server is started on this station by:

sounddemo

(login as vrgroup)

On the back of the ARIEL amplifier, a (continuous) sound source (CD player, radio, SGI output) has to be hooked onto the input.

The system is activated by adding *SVE_SPATIALSOUND* to config when calling `SVE_init()`. The only functions that are available to control the location of the sound are `SVE_attachSoundToObject(object)` and `SVE_changeSoundUpdateRate(number)`.

3.11. Polling Devices

As it true of the real world, an interactive application can involve many events that are continuous in nature, such as computer mouse movement and body movement. As computers generally deal in discrete events only, the continuous events need to be translated into discrete events. This is usually done by *polling*, or *sampling* the continuous event, where the continuous event is sampled at certain times and each sample represents an event. In many cases, a series of sampled events can be interpreted to mean that another, more specific, event has occurred. For example, when a hand's position moves through the position of a computer generated wall, a *collision* event might occur, or when a hand changes from having extended fingers to being a fist, a *grab* event might occur.

The SVE system maintains a list of polling devices that translate continuous events into discrete events. The "device" could be an actual hardware device, or a software animation. Polling devices are polled once through the event-render loop, during the "event handling" phase (see figure Figure 10. "SVE System Overview" on page 35). The SVE system handles two types of continuous events through polling: tracking devices and hand input (glove) devices. A tracking device reports a position and orientation of a tracker, which is usually attached to a point of interest such as a head or hand. A hand input device reports the posture of the hand, perhaps as a set of finger and wrist bend angles. Other continuous events can be handled using SVE's polling system by writing an open, polling, and close function, and registering it with the SVE system. The polling system provides the ability to associate a polling device with a SVE object.

3.11.1 Tracking Devices

Tracking devices generally determine the position and orientation of an SVE object in the environment. The default behavior of the SVE tracker polling device is to copy the position and orientation of the tracker to the position and orientation of the SVE object associated with that tracker. It is important, therefore, to note that the position of the object is also determined by the position of its parent object, which would represent the position of the tracker's reference (usually the transmitter of an electromagnetic tracker) in the environment.

The usual method to introducing trackers to the application is to specify them in the initialization file. This causes the trackers to be automatically initialized and polled (if the `SVE_HMD` option is set in the configuration flags). Here are some example lines in the initialization file that set up the default situation of one tracker to determine the position of the "SVE HMD" object (and, thus determine the viewpoint because this is the `SVE_worldState->hmdObject`), and another tracker to determine the "SVE cursor" object (often held by the hand, or attached to a 3D mouse). Note that these lines specify that the trackers are receiver 1 and 2 of an IsotrackII™ tracking system which is plugged into the `/dev/ttyd2` port of a machine called "`buckhead`".

```
tracker 1              buckhead isotrakii /dev/ttyd2 1 SVE HMD
tracker 2              buckhead isotrakii /dev/ttyd2 2 SVE cursor
```

The creation of a tracker interface has the effect of introducing a new object, called "SVE tracker X", where X is the id number given. This object is made a child of the parent of the "attached" object, whose name was given ("SVE HMD" and "SVE cursor" in this example). The "attached" object is then made a child of the "SVE tracker X" object. (See "Tracker Objects" on page 61.) When new position information is obtained from the tracking device, it is stored in the position of the "SVE tracker X" object, effectively moving the "attached" object in the coordinate system of its former parent. Note that the "attached" object may have a position transformation that is combined with the tracker position to determine the "attached" object's world position. This can be used, for example, to correct for tracker devices that are rotated to be mounted on the side of a hand held device.

The tracker identification number of the trackers in the example are 1 and 2 respectively, given in the second column of the line. That number can be used to refer to the tracker if you wish to change what

object it is associated with (which object it determines the position and orientation of). The function to change the object association is,

```
void SVE_attachTracker(int trackerId, char *objectName,
                          SVE_pollFunctionPtr pollFunction);
```

Note that it is possible to specify your own polling function. The format of the polling function is discussed below. Generally, you will want to use the standard tracker polling function, which reads the tracker information from the serial port and changes the position of the associated object appropriately. If any buttons are associated with the tracker, they are translated to mouse button events, and queued. The standard function can be specified by setting the `pollFunction` parameter to `NULL`, or by explicitly specifying the standard function, which is `SVE_updateTracker()`.

```
SVE_status SVE_updateTracker(SVE_pollDevice device, SVE_state state);
```

It is possible to use a tracker during run-time versus specifying it in the configuration file. To do so, use the following function,

```
boolean SVE_initTracker(int trackerId, char *machine, int type, char *port,
                           int receiver, char *attachTo, float hemiVector[3],
                           SVE_pollFunctionPtr pollFunction);
```

This function returns `TRUE` on success, `FALSE` otherwise. Note that, for the most part, the parameters match the ones given in the configuration file tracker specification. The `type` parameter is one of the types listed in the `tracker.h` file of the tracker library. Currently, it should be one of the following: `BIRD`, `ISOTRAKII`, `FASTRAK`, or `BOOM`. The `port` parameter should be the name of the port to which the tracker it attached ("/dev/ttyd1" or "/dev/tty00" for example). The default for the `hemiVector`, which is really only used for electro-magnetic trackers, is [0, -1, 0], which assumes that the tracker transmitter is above the user. If the transmitter is in front of the user, where the user is facing down the negative Z axis when looking towards the transmitter, the hemisphere vector should be [0, 0, 1].

The `attachTo` parameter is the name of the SVE object to attach to. It is not necessary that the object exist at the time of the call to `SVE_initTracker()`. The tracker will automatically associate itself to the object when it is created. As in the `SVE_attachTracker()` function, the `pollFunction` parameter is the polling function for the tracker. It should be `NULL` or `SVE_updateTracker()` to use the default.

There is no restriction on the number of tracker devices that actually use the same tracker receiver. It is possible to have two objects, therefore, that follow the same tracker receiver. If they have the same parent, or if their parents are at the same location and orientation, the two objects will be at the same position. Of course, if you wish to have two objects at the same position, it is more efficient to attach one to the other, and set its (the attaching one's) location to [0, 0, 0].

3.11.2 Hand Input Devices

Currently, the SVE system is capable of handling the CyberGlove™ hand input device. The use of this device is enabled by setting the SVE_GLOVE option of the configuration flags. The glove device is treated in a way similar to the tracking devices in terms of initialization and polling. There are two methods to using a glove device: specifying the glove device in the initialization file and initializing a glove device during run time. A glove device can be specified in the initialization file using a similar format to the one used to specify a tracker. A typical example line in the initialization file that uses a glove that is attached to the "SVE cursor" object is,

```
glove 1                    buckhead /dev/ttyd1 SVE cursor
```

The second column gives the glove's identification number which is used to refer to it in the functions to follow. The next columns give the name of the machine and the port that the glove is attached to, and the name of the object that the hand is attached to. The function equivalent of this is,

```
boolean SVE_initGlove(int gloveId, char *machine, char *port, char *attachTo);
```

This function returns TRUE on success, FALSE otherwise. As in the tracker device, the attachTo parameter identifies the object with which to associate the glove device. The SVE object of that name does not need to exist at the time of the initialization call.

When the glove device is initialized, a hand geometry is created as an SVE object tree, which is attached to the object with which the glove device is associated. The geometry represents the parts of the hand which will be controlled by the sensors on the glove. A diagram of the hand SVE object tree is shown in Figure 17. Note that it is possible to find the location of the finger-tips by finding the location of the appropriate

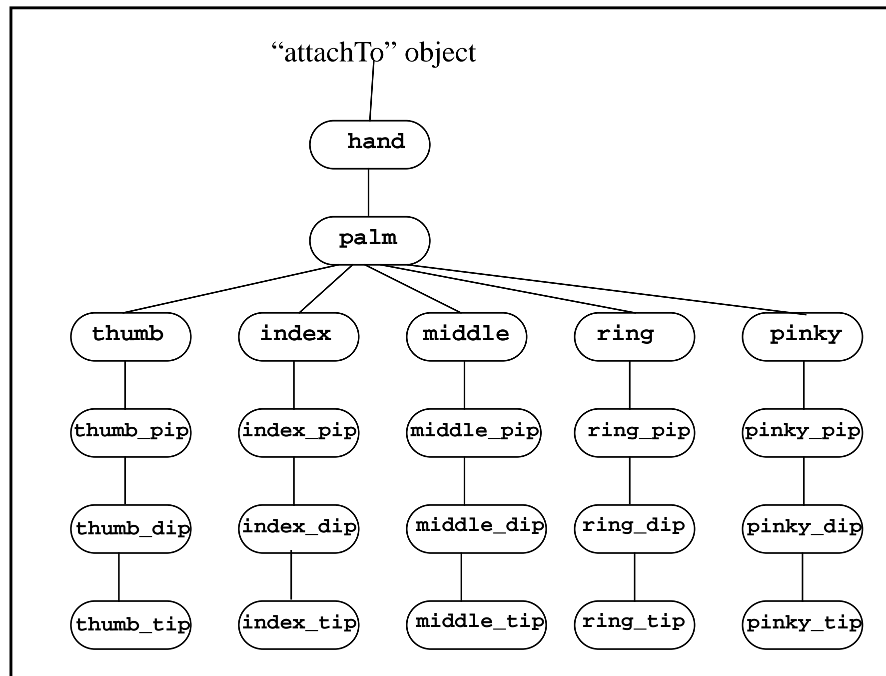


Figure 17. Hand SVE Object Tree

finger-tip object (the hand size is approximated, though). Since the "palm" object is often used for interactions between the hand and the environment, a function is provided to obtain the "palm" object directly for a glove device attached to a particular object. The function is,

```
SVE_object SVE_getPalmObject(SVE_object attachedTo);
```

As an example, the following line of code will get the palm object for the hand geometry attached to the cursor object:

```
palm = SVE_getPalmObject(SVE_getCursorObject());
```

Glove Calibration

The CyberGlove needs to be calibrated for each individual's hand, although it is possible to save and retrieve that calibration for later sessions. The following procedure will calibrate the glove device 1. The three steps must be done in order, although they can be repeated as many times as needed.

be set interactively through these three steps (which must be done in order, but can be done at any time):

- Place the hand flat on a table top with the fingers together and the arm straight out from the hand and press '0'.
- Make a fist with the hand, curling the thumb in front of the fingers so that all of the finger joints are bent as much as possible. Bring the fist back as if to knock on a door and press '9'.
- Spread the fingers out (as in "five") and twist the wrist so that the hand is twisted away from the body and press '5'.

After these steps, the hand in the Virtual Environment should closely resemble the hand in the physical world. A hand calibration can be saved using the function,

```
void SVE_saveHandFile(char *filename, int gloveId);
```

A hand calibration file can be read back in, and used for a particular glove device using the function,

```
void SVE_readHandFile(char *filename, int gloveId);
```

In both routines, the `gloveId` parameter is the glove identification number of the glove device.

Gesture Recognition

Gesture recognition can be obtained in two fashions, which can be used concurrently if desired. A gesture file, which defines a set of gestures, can be created and read in with the function,

```
void SVE_readGestureFile(char *filename, int gloveId);
```

The `gloveId` parameter specified the glove device to use for gesture recognition. The gesture file format is described in detail in the Appendixes. Gestures can defined most precisely using this method, as the file can specify that only certain joints are relevant, and gestures that change over time can be defined. Here is an example gesture file.

Source xviii: Gesture File Example

```
# This is a gesture file

# First define the default ranges used for joints

# 0 to 90 degree bends
joint_positions: 4
bounds:
-45 15   0 # min max consider_as
 15 45  30
 45 75  60
 75 135 90

thmb rot
thmb mphl
thmb ip
thmb abdt

indx mphl
indx pxip
indx dsip

midl mphl
midl pxip
midl dsip

ring mphl
ring pxip
ring dsip

pnky mphl
pnky pxip
pnky dsip

# Abduction angles: 0 to 30 degrees
joint_positions: 4
bounds:
-45  5   0 # min max consider_as
  5 15  10
 15 25  20
 25 75  30

midl abdt
ring abdt
pnky abdt
```

```
# Next, define gestures

name: grab
thmb rot  0.8 0.8 0.9 1.0 # 0, 30, 60, 90 degrees
thmb mphl 0.9 0.9 1.0 1.0
thmb ip   0.9 0.9 1.0 1.0
# thmb abdt 0.5 0.7 0.9 1.0

indx mphl 0.5 0.6 1.0 1.0
indx pxip 0.5 0.6 1.0 1.0
indx dsip 0.5 0.6 1.0 1.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.5 0.6 1.0 1.0
midl dsip 0.5 0.6 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
ring pxip 0.5 0.6 1.0 1.0
ring dsip 0.5 0.6 1.0 1.0

pnky mphl 0.5 0.6 1.0 1.0
pnky pxip 0.5 0.6 1.0 1.0
pnky dsip 0.5 0.6 1.0 1.0

name: point
indx mphl 1.0 0.9 0.0 0.0
indx pxip 1.0 0.9 0.0 0.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.0 0.0 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
ring pxip 0.0 0.0 1.0 1.0

pnky mphl 0.5 0.6 1.0 1.0
pnky pxip 0.0 0.0 1.0 1.0

name: pinky_wave
indx mphl 0.5 0.6 1.0 1.0
indx pxip 0.0 0.0 1.0 1.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.0 0.0 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
ring pxip 0.0 0.0 1.0 1.0

pnky mphl 0.0 0.0 1.0 1.0
pnky pxip 0.0 0.0 1.0 1.0

transitions:
pnky mphl backward
pnky pxip backward

end:
pnky mphl 1.0 1.0 0.0 0.0
pnky pxip 1.0 1.0 0.6 0.0

name: thumbs_up
priority: 10
thmb rot  1.0 0.8 0.0 0.0
thmb mphl 1.0 1.0 0.0 0.0
thmb ip   1.0 1.0 0.0 0.0
thmb abdt 0.0 0.5 0.9 1.0

indx pxip 0.0 0.0 1.0 1.0

midl pxip 0.0 0.0 1.0 1.0

ring pxip 0.0 0.0 1.0 1.0

pnky pxip 0.0 0.0 1.0 1.0
```

Alternatively, gestures (really just hand postures) can be defined by example using the function,

```
void SVE_saveCurrentGesture(int gloveId, int priority,
                            boolean replaceOldGesture);
```

Each time this function is called, the hand position at the time of the last polling of glove sensor values is saved. The gesture is identified by the `priority` parameter. If the `replaceOldGesture` flag is `TRUE`, the gesture will be the only one associated with the priority number, otherwise the gesture is added to a list of gestures identified with the priority number. The priority value that identifies the gesture also indicates which gesture will be recognized first if many gestures could be recognized (low priority value indicates higher priority in order)

In both cases, gestures made by a given glove device will not be recognized until the following function is called with a `TRUE` flag.

```
void SVE_recognizeGestures(int gloveId, boolean flag);
```

If this function is called with a `FALSE` flag, gestures are once again not recognized. Whenever a gesture that has been defined is recognized, a `GESTURE` event will be entered onto the GL event queue with the identifying gesture priority as the event's value. When the system is recognizing gestures, it generates a `GESTURE` event with a `NULL_GESTURE` value each time the system goes through the SVE loop and the current hand position does not match a gesture definition.

Gestures are recognized by "polling" each joint of the hand for an opinion as to whether a particular gesture is being made. Each joint is assigned an angle range (the choices are defined in the gesture file), and a probability value for each possible gesture. If a joint with that angle range could be part of the gesture, the system reports a probability of 1.0 or less (if it isn't too sure). If the joint in that angle range could NOT be part of the gesture it reports a probability of 0.0. The probabilities of each joint are multiplied together, and compared to a threshold and other gesture probabilities. Thus, a joint can "veto" the probability of a gesture being made.

The list of gestures for a particular glove device can be reset with the function,

```
void SVE_resetGestureList(int gloveId);
```

The following example demonstrates the use of the glove. If you are not using the trackers, you will want to press 'g' to move the glove out of the way of the view. Different gestures can be saved by pressing the 's' key. A sample gesture file can be read by pressing 'l'. After going through the calibration routine, you can save the calibration file with the 'h' key and read it back in with the 'H' key.

Source xix: glove_example

```
/* Glove example

   starts up with the trackers if 1st parameter is "t",
   reads a world and registers an event-callback routine for recognizing
   glove gestures and the glove button.  Guestures are saved everytime the
   's' key is pressed.

   The glove can be calibrated to a hand by pressing '0' when the hand is
   stretched out fingers together, pressing '9' when the hand is a closed fist
   with the wrist back as if to knock on a door, and pressing '5' when the
   fingers are spread out and the wrist is twisted so that the hand is turned
   to the right.  The order is important, but this procedure can be repeated
   at any time.

   A calibrated hand can be saved using the 'H' (shift 'h') key, and recalled
   using the 'h' key.

   If the trackers aren't being used, the glove will obscure the view.  Press
   'g' to move it to be just in front of the viewer.

   A set gesture list (contained in "testgestures.gest") can be loaded using
   the 'l' key.
*/
```

```
#include "sve.h"

int priority = MIN_GESTURE;

SVE_status gestureCallback(SVE_state state)
{
   int gestureNum;

   if (state->eventType == SVE_GESTURE) {
      gestureNum = ((SVE_gestureEvent *)state->eventData)->gestureVal;
      if (gestureNum != NULL_GESTURE)
         printf("Gesture %d event recognized\n", gestureNum);
   }
   return(EVENT_IGNORED);
}

SVE_status gloveButtonCallback(SVE_state state)
{
   if (SVE_IS_PRESS_EVENT(state->eventType)) {
      printf("Glove button is now %d\n",
             ((SVE_stateChangeEvent *)state->eventData)->pressed);
   }
   return(EVENT_IGNORED);
}


SVE_status setGloveStuff(SVE_state state)
{
   int allright;
   SVE_object hmd;
   SVE_status retval = EVENT_IGNORED;

   if (state->eventType == SVE_KEY_PRESS) {
      switch(((SVE_keyEvent *)state->eventData)->keyVal) {
      case 'g':
         hmd = SVE_getHMDObject();
         SVE_copyMatrix(hmd->position, state->cursorObject->position);
         SVE_translateObjectGlobal(state->cursorObject, 0.0, -0.1, -0.2);
         retval = EVENT_CONSUMED;
         break;
      case 's':
         SVE_recognizeGestures(1, TRUE);
         SVE_saveCurrentGesture(1, priority, TRUE);
         priority++;
         retval = EVENT_CONSUMED;
         break;
      case 'l':
         SVE_recognizeGestures(1, TRUE);
         SVE_readGestureFile("testgesture.gest", 1);
         retval = EVENT_CONSUMED;
         break;
      case 'H':
         SVE_saveHandFile("testhand.hand", 1);
         retval = EVENT_CONSUMED;
         break;
      case 'h':
         SVE_readHandFile("testhand.hand", 1);
         retval = EVENT_CONSUMED;
         break;
      } /* switch */
   }
   return(retval);
}


void main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL | SVE_GLOVE;

   printf("Starting application\n");

   /* Checking command line for a 't' as the first paramter */
```

```
        if ((argc > 1) && (strcmp(argv[1],"t") == 0))
           config = SVE_HMD | SVE_GLOVE;

        SVE_init("Glove example", config, &argc, argv);

        if(!SVE_loadWorld("glove_example.world"))
        {
           printf("error occured during SVE_loadWorld, exiting \n");
           SVE_done();
        } /* if */

        printf("Registering input callbacks\n");
        SVE_registerCallback(SVE_KEY_PRESS, setGloveStuff);

        SVE_registerCallback(SVE_GESTURE, gestureCallback);

        printf("Beginning event loop\n");
        SVE_beginEventLoop();

        printf("Done -- Have a nice day.\n");
        SVE_done();
    }
```

3.11.3 User-Defined Polling Devices

The polling devices handled by SVE both use an underlying polling device system that is available to the SVE application to create other polling device-type events. Generally, a polling device goes through three phases: initialization and opening of the device, polling the device for information, closing the device. In addition, a polling device can be associated with a particular SVE object and a block of data specific to the device. When a polling device is created and added to the polling device list, the SVE system calls the open function once, then calls the polling function each time through the event-render loop, and finally calls the close function when the SVE application calls `SVE_done`.

A polling device can be created using the `SVE_createPollingDevice()` function.

```
SVE_pollDevice SVE_createPollingDevice(int type, int deviceId,
                                       SVE_pollFunctionPtr openFunction,
                                       SVE_pollFunctionPtr pollFunction,
                                       SVE_pollFunctionPtr closeFunction,
                                       char *attachTo, void *data);
```

The `type` parameter identifies what kind of polling device is being created. The SVE system uses `SVE_TRACKER_DEVICE` for tracking devices, `SVE_GLOVE_DEVICE` for hand input devices, and `SVE_MISC` for other devices. An application is free to use `SVE_MISC` for a device type, or another integer which does not conflict with the ones defined.

The `deviceId` is a unique identifier for each polling device. The three functions are the ones called to open, poll, and close the device respectively. The form of these functions is,

```
SVE_status SVE_pollFunction(SVE_pollDevice device, SVE_state state);
```

The `attachTo` parameter is the name of a SVE object with which the object should be associated. The object does not need to exist at the time of this function call, and may not exist for any of the calls to the user-defined functions. If the object does exist, the user-defined functions can obtain it using `device->attachedTo`.

The `data` parameter is a pointer which is available to the user-defined functions using `device->data`.

After a polling device is created, it needs to be added to the SVE system's polling device list. This is done using the `SVE_addPollingDevice()` function, giving the value returned by the `SVE_createPollingDevice` function.

```
void SVE_addPollingDevice(SVE_pollDevice device);
```

The polling device information can be retrieved using the `SVE_findPollingDevice()` function, which returns a pointer to polling device structure. Of particular interest are the `data` and `attachedTo` fields

(see structure description in section 4.11. of APPENDIX C:), which are the data assigned to the device when is was created, and the `SVE_object` that the device is associated with.

```
SVE_pollDevice SVE_findPollingDevice(int type, int id);
```

3.12. 3D interactors

The most interesting Virtual Environments applications usually contain some sort of interactivity in it, whether it is the user interacting with the elements of the environment, or the elements of the environment interacting with each other. The set of three dimensional widgets, including buttons, menus and color selectors which float in space is a specialized group of these interactors. The SVE library provides a framework for 3D widgets, as well as other entities that respond to events that occur. These widgets are treated as special SVE objects, which react to given events with a given behavior. The SVE object representing a widget of a particular widget type is given a name of the form, "<widget_type_name>:<widget_name>". The SVE system maintains the set of registered widget types, allows for widgets to be loaded in from a file (the format of which is determined by the widget developer), calls the appropriate function when events occur in which the widget is interested, and allows for direct access to information on any particular widget. An explanation of the functions used to register a widget type, and then make instantiations of it, follow. The given code is from an example application ("widgetExample.c"), which defines a simple button widget, and then creates two of them to interact with (one of the them is loaded from a file). The following figure shows what is seen when this example application is run.

Figure 18. Widget Example Screen Shot.

3.12.1 Registering a Widget Type

The first step to building a widget type is to register the type. This is done using the function `SVE_registerWidgetType()`. The function takes a string name, which is used to identify the widget type, functions to create an instance of the widget, to create an instance of the widget from a file, and to call when one of the given list of events occurs. The list of events is given as a linked list, which can be built using the `createList()` and `addToList()` functions of the `list` library (see APPENDIX C: section 2.1. "Linked List" on page 177).

```
void SVE_registerWidgetType(char *type, SVE_createWidgetFunctionPtr createFunc,
                            SVE_readWidgetFunctionPtr fileCreateFunc,
                            SVE_widgetFunctionPtr deleteFunc,
                            SVE_widgetFunctionPtr eventFunc, list eventList);
```

In our simple button example, the "simple_button" widget is registered in the `initSimpleButtons()` function. Note that the simple button widget is only interested in `SVE_OBJECT_SELECTION` events, which occur when an object is selected.

```
void initSimpleButtons()
/*
   Register the simple button widget by given an identifying name, the
   creation function, the creation from file function, and the event
   callback routine, which will be called anytime an object selection
   event occurs.
*/
{
    list eventList = createList();

    addToList(&eventList, (void *) SVE_OBJECT_SELECTION);
    SVE_registerWidgetType("simple_button", createButton, fileCreateButton,
                          deleteButton, selectButton, eventList);
}
```

3.12.2 Creating a Widget

The create function for the widget should be in the following form.

```
SVE_object createFunc(char *name, void *data);
```

The function takes a character string which is the widget's name, and a pointer to data specific to the widget. The data pointer refers to a structure specific to the widget type which is set up before, and passed to the instantiation function, `SVE_makeWidget()`. The create function returns a SVE object which represents the widget.

A widget create function should, at least, create an empty SVE object using the name given, and return it. In our simple button example, the `createButton()` function creates an empty SVE object, then creates the button and text geometry from the data given, and returns the object.

Source xx: Simple Button Widget Example

```
SVE_object createButton(char *name, void *data)
/*
   This is the widget creation function for a simple button.  It creates an
   empty object, assigns a button geometry to it, and returns it.
*/
{
   SVE_object button;
   float color[3] = {0.8, 0.8, 0.3};
   SVE_primitive primitive;

   button = SVE_createEmptyObject(name);
   button->geometry = SVE_createGeometry(name);
   createButtonPrimitives(button, (simpleButtonStructPtr) data);

   return(button);
}
```

The data for the simple button widget is defined in this structure definition:

```
/* Define the data required for each button object */

typedef struct simpleButtonStruct {
   char            *label;
   int              materialIndex;
   SVE_functionPtr  buttonCallback;
} simpleButtonStruct, *simpleButtonStructPtr;
```

The primitives for the button's text and background are constructed in the `createButtonPrimitives()` and `createButtonPrimitive()` functions. Note that the object containing the button is set to be `selectable`, so that it can be selected, and a `SVE_OBJECT_SELECTION` event can occur for the object.

```
SVE_primitive createButtonPrimitive(simpleButtonStructPtr data)
/*
```

```
  Create and return the simple button primitive, which is a square that
  closely surrounds the button's text.
*/
{
   SVE_primitive buttonPrim;
   float height = 1;
   float width = 1;
   float originX = 0;
   float originY = 0;
   SVE_material faceMaterial;

   /* Find the material for the button's face */
   faceMaterial
      = SVE_getMaterialByIndex(data->materialIndex);

   /* Compute the extent of the button's text */
   if (data->label != NULL)
      SVE_getTextExtent(data->label, &originX, &originY, &height, &width);

   /* Construct the face that will surround the text a little behind it. */
   SVE_beginPrimitive(POLYHEDRON);

   /* vertices */
   SVE_primitivePoint(0, originX+width+0.1, originY-0.1, -0.1);
   SVE_primitivePoint(1, originX-0.1, originY-0.1, -0.1);
   SVE_primitivePoint(2, originX-0.1, originY+height+0.1, -0.1);
   SVE_primitivePoint(3, originX+width+0.1, originY+height+0.1, -0.1);

   /* indices for button polygon */
   SVE_beginPrimitiveFace();
   SVE_primitiveVertexIndexes(0, -1, -1);
   SVE_primitiveVertexIndexes(3, -1, -1);
   SVE_primitiveVertexIndexes(2, -1, -1);
   SVE_primitiveVertexIndexes(1, -1, -1);

   SVE_primitiveMaterial(faceMaterial);

   SVE_endPrimitiveFace();

   buttonPrim = SVE_endPrimitive();

   return(buttonPrim);
}


void createButtonPrimitives(SVE_object button, simpleButtonStructPtr data)
/*
  Create the button's geometry, which contains the button's text with a
  square backdrop in the button's color.
*/
{
   float color[3] = {0.8, 0.8, 0.3};    /* Highlight Color */
   SVE_primitive primitive;

   primitive = createButtonPrimitive(data);
   SVE_addPrimitiveToObject(button, primitive);

   /* Create and add the button's text to the geometry */
   if (data->label != NULL) {
      SVE_beginPrimitive(TEXT);
      SVE_primitiveText(data->label);
      primitive = SVE_endPrimitive();
      SVE_addPrimitiveToObject(button, primitive);
   }

   /* Important!  Allow the button to be selected by the selection ray! */
   button->selectable = TRUE;
   SVE_setObjectHighlightMaterial(button, SVE_getColorMaterial(color), FALSE);
}
```

3.12.3 Creating a Widget From a File

When an application is loading in a SVE world file, it is possible for an object to be defined by a widget file description, rather than the usual SVE object file description. The widget file name is specified on the "primitives file:" line of the SVE world file. The widget file is identified by a header line which reads:

```
Simple Virtual Widget File: <widget type name>
```

Note that the widget type name corresponds to the type name used when registering the widget type. Therefore, it is important to register all widget types before loading a SVE world file, so that they will be recognized. Here is an example widget file for a simple button widget.

```
Simple Virtual Widget File: simple_button

# This file defines a simple button

label: Hit me too
Color: 0.2 0.2 0.7
```

The `fileCreateFunc` parameter specified for the `SVE_registerWidgetType()` function should be a function of the following form:

```
void *fileCreateFunc(SVE_object object);
```

The purpose of this function is to read the widget file, set up the data structure specific to the widget type, and return that data structure. The SVE object which will represent the widget is given to the function. The file can be parsed using two functions, FP_`getNextToken()`, which returns the next string of characters which do not contain any white space (spaces, tabs, etc.), and FP_`getRemainingLine()`, which returns the rest of the characters on the current line, which may include white space, and will include the carriage return at the end. The file will be read beginning from the line following the header line. comments, which are begun by a '#' character, and continue to the end of the line, will be automatically ignored. The file reading functions return `END_OF_FILE` when the end of the file has be reached. (See APPENDIX C: section 2.3. "File Parser Utility" on page 183 for more information on file reading.)

The file reader for the simple button widget is shown below. It recognizes three attributes, which can appear in any order: "label:", which is followed by the button's label (and can contain spaces), "material:", which is followed by the name of the material used for the button's background, and "color:", which if followed by three values 0-1 to indicate the red, green, and blue color combination for the button's background.

```
void *fileCreateButton(int FileNo, SVE_object button)
/*
   This is the widget file creation function for a simple button.  It reads
   from the given file the characteristics of the button, constructs the
   button's data structure, and returns it.
*/
{
    simpleButtonStructPtr buttonData;
    char *label;
    char *word;
    char *materialName;
    SVE_material material = NULL;
    float color[3] = {0.7, 0.7, 0.7};

    buttonData = (simpleButtonStructPtr) malloc(sizeof(simpleButtonStruct));

    do {
        if ((word = FP_getNextToken(FileNo)) != NULL) {
            if (strcasecmp(word, "label:") == 0) {
                /* Read the button's label text. */
                label = FP_getRemainingLine(FileNo);
                buttonData->label = strdup(label);
                buttonData->label[strlen(label)-1] = '\0';
            } else if (strcasecmp(word, "material:") == 0) {
                /* Read the button's material name */
                materialName = FP_getNextToken(FileNo);
```

```
                material = SVE_getMaterialByName(materialName);
                if (material != NULL)
                    buttonData->materialIndex = material->index;
                else {
                    /* If we can't find the material, use a default
                       instead. */
                    material = SVE_getColorMaterial(color);
                    buttonData->materialIndex = material->index;
                }
            } else if (strcasecmp(word, "color:") == 0) {
                /* Construct the button's material from the RGB color
                   triple given. */
                color[0] = FP_getNextFloat(FileNo);
                color[1] = FP_getNextFloat(FileNo);
                color[2] = FP_getNextFloat(FileNo);
                material = SVE_getColorMaterial(color);
                buttonData->materialIndex = material->index;
            } else
                FP_reportFileError(FileNo, "Unknown attribute %s", word);
        }
    } while (word != END_OF_FILE);

    if (material == NULL) {
        /* If no material was specified, assign a default material. */
        material = SVE_getColorMaterial(color);
        buttonData->materialIndex = material->index;
    }

    /* No callback function can be specified in the file */
    buttonData->buttonCallback = NULL;

    createButtonPrimitives(button, buttonData);
    return((void *) buttonData);
}
```

3.12.4 Widget Event Function

The event function specified when a widget type is registered is similar to the normal event callback functions of the SVE library. The only difference is that the widget event function receives the widget object, and the data associated with that particular widget. The data is given as a "`void *`", and needs to be cast to be a pointer to the specific data structure for the widget.

```
SVE_status eventFunc(SVE_object widget, SVE_state state, void *data);
```

This event function is called whenever one of the specified events occurs, whether or not it really involves the particular widget. In our simple button example, we first check to make sure that the object being selected is the same object as the widget, then, if the widget is the one selected, we call the callback function associated with that button.

```
SVE_status selectButton(SVE_object button, SVE_state state, void *data)
/*
   This is the event callback function for the simple button widget.  It
   will be called anytime an object is selected.  If the object selected
   is the given widget ("button"), then call the callback routine stored
   for the widget.
*/
{
    SVE_object selectedObject;
    simpleButtonStructPtr buttonData;

    buttonData = (simpleButtonStructPtr) data;

    if (SVE_IS_OBJECT_EVENT(state->eventType)) {
        selectedObject = ((SVE_objectEvent *)(state->eventData))->object;
        if ((selectedObject == button)
             && (buttonData->buttonCallback != NULL))
            return(buttonData->buttonCallback(state));
        else return(EVENT_IGNORED);
    } else return(EVENT_IGNORED);
}
```

3.12.5 Widget Instantiation

The previous sections all involve defining what a particular widget type is and how it behaves. This can be thought of as a class of widgets, such as buttons. For a widget to actually be used, it much be instantiated, or created using the definition for a particular type or class of widgets. The function that accomplishes this is called `SVE_makeWidget()`. It takes the string that identifies which widget type to use, a string name to identify the particular widget being created, and a reference to a data structure which is particular to the widget type, and has be set up for the widget being created. The function returns the SVE object that has been created to represent that widget. If the widget object is to be seen, it needs to be added to the world object tree. However, a widget that has been created will respond to events even if it is not part of the world object tree (although, obviously, it will never be selected or highlighted).

```
SVE_object SVE_makeWidget(char *type, char *name, void *data);
```

In our simple button widget example, we have written a function specific to the simple button widget type to create simple buttons, and return the SVE object that represents the button. This function sets up the data structure used by the simple button widget type, and passes that on to the instantiation function.

```
SVE_object makeSimpleButton(char *name, char *label, SVE_material material,
                            SVE_functionPtr buttonCallback)
/*
   The function will create an instance of a simple button widget, giving
   it the given name, label, material, and callback function.  The
   callback function will be called when the button is selected.
*/
{
    SVE_object buttonObject;
    simpleButtonStructPtr buttonData;
    float color[3] = {0.7, 0.7, 0.7};
    SVE_material defaultMaterial;

    /* Create the data structure specific to simple button widgets. */
    buttonData = (simpleButtonStructPtr) malloc(sizeof(simpleButtonStruct));
    buttonData->label = strdup(label);
    if (material != NULL)
       buttonData->materialIndex = material->index;
    else {
       defaultMaterial = SVE_getColorMaterial(color);
       buttonData->materialIndex = defaultMaterial->index;
    }
    buttonData->buttonCallback = buttonCallback;

    /* Create an instance of the simple button widget */
    buttonObject = SVE_makeWidget("simple_button", name, (void *) buttonData);

    return(buttonObject);
}
```

In the example application that uses the simple button widget, a callback function is defined, called `buttonHit()`, and the simple button widget type is initialized and one button is instantiated. The button is added to the object tree so that it is directly in front of the viewer.

```
SVE_status buttonHit(SVE_state state)
/* Callback routine for an instance of a simple button widget. */
{
   SVE_object selectedObject;

   if (SVE_IS_OBJECT_EVENT(state->eventType)) {
      selectedObject = ((SVE_objectEvent *)(state->eventData))->object;
      printf("Button '%s' has been hit\n", selectedObject->name);
   }
   return(EVENT_CONSUMED);
}


main(int argc, char *argv[])
{
```

```
    SVE_config config = SVE_SELECT;  /* allow selection of the widget */
    SVE_object button;
    SVE_point buttonPos = {-1, 2, -2};
    float buttonColor[3] = {1, 0, 0};
    SVE_material buttonMaterial;

    printf("Starting application.\n");
    SVE_init("Widget Example (sve)", config, &argc, argv);

/* This must come before loading the world file, or button widgets in the
   world file will not be recognized */
    initSimpleButtons();

/*
   Load in the world file. This function returns FALSE when the world
   could not be loaded correctly.
*/
    if(!SVE_loadWorld("widgetExample.world"))
    {
       printf("error occured during SVE_loadWorld, exiting.\n");
       SVE_done();
     }

/*
   Create an instance of the simple button widget, and add it to the
   world tree.
*/
    buttonMaterial = SVE_getColorMaterial(buttonColor);
    button = makeSimpleButton("Hit", "Hit me", buttonMaterial, buttonHit);
    SVE_addToWorldTree(button);
    SVE_moveObject(button, buttonPos);

    printf("Beginning event loop\n");
    SVE_beginEventLoop();

    printf("Done -- Have a nice day.\n");
    SVE_done();
}
```

3.12.6 Widget Deletion

If a widget object is deleted, then the given `deleteFunc` of the widget type will be called. It is of the form

```
SVE_status deleteFunc(SVE_object widget, SVE_state state, void *data);
```

The delete callback should free any memory that it is using for the widget. The following function does this for our button example.

```
SVE_status deleteButton(SVE_object button, SVE_state state, void *data)
/*
   This is the delete callback function for the simple button widget.  It
   will be called anytime an object is deleted.
*/
{
    simpleButtonStructPtr buttonData;

    buttonData = (simpleButtonStructPtr) data;
    free(buttonData);

    return(SVE_OK);
}
```

3.12.7 Retrieving Widget Data

Given a widget type, and a name of a widget of that type, the `SVE_getWidgetData()` function can return a pointer to the data associated with that particular widget. This is valuable to set certain variables in the data structure which could not be set when it was created.

```
void *SVE_getWidgetData(char *type, char *name);
```

A common example of this is demonstrated in the following function, which sets the callback function for a simple button widget. This could be used, for example, to set the callback function for widgets created from a file.

```
void addCallbackToButton(char *name, SVE_functionPtr buttonCallback)
{
    simpleButtonStructPtr buttonData;

    buttonData = SVE_getWidgetData("simple_button", name);
    if (buttonData != NULL)
       buttonData->buttonCallback = buttonCallback;
    else fprintf(stderr, "Error -- Can't add callback to %s, there is no"
                "button of that name\n", name);
}
```

3.13. Servers

It is often the case that the machine that is rendering the graphics for the SVE application is not the machine to which the desired tracking device it attached, and is not the machine to which the user wishes to output audio or receive key or mouse events. These things can be handled on remote machines using server programs that run on the machine providing the service, and which communicates with the SVE application using that service. Server programs are separate programs that need to be run on the machine with the service. They are found in the `bin` directory of the SVE directory structure. In most cases, the initialization file indicates to the application which machines to look for server programs (See "Initialization File" on page 28). Here are descriptions of each server used by the SVE system.

---

server-tracker

The location of a tracker device used by the application is specified in the initialization file, or when the tracker is initialized by the application. Any time a tracker is used, even if it is on the same machine as the application, a tracker server needs to be running for the communication with the trackers (so that other applications on different machines can also use the same tracker device). The tracker server is called `server-tracker`.

---

server-event

The event server creates a window on the bottom left part of the screen on the machine it is run. Any key presses and mouse clicks that occur when the mouse pointer is in this window will be sent on to an interested application. The event server is called `server-event`.

---

server-audio

The audio server will load and play audio files when told to do so by an application communicating with it. The application gives the server complete path names for audio files, so as long as the audio server is run on a machine on the correct file server, then there will be no problem loading audio files no matter where the audio-server is run. An application can play a sound file continuously, play it only once, or stop it at any time, just as if the audio was being taken care of by the SVE system at the host machine. The audio server is called `server-audio`.

3.14. Porting Version 1.5 Applications to Version 2.0

The following items are things to keep in mind when porting SVE version 1.5 applications to SVE version 2.0: (This information is contained in a file called "`CONVERTING.FROM.V1.5`" in the root of the SVE directory tree.)

Makefile

The `Makefile` has changed somewhat. Be sure to copy over a `Makefile` from an example directory (`examples/look` is suggested), and use it. Things you will need to change in the Makefile:

```
include .../makeinc...
```

Make sure that this line has the right directory for the appropriate "`makeinc`" file, which is the root of the SVE directory tree. For Georgia Tech, it should read:

```
include /net/hg43/vrgroup/sve/v2.0/makeinc.SUFFIX.GRAPHICS
```

where "`SUFFIX`" should be replaced by "`sgi`" or "`hp`" depending on whether you are compiling your program on an HP or SGI machine, and "`GRAPHICS`" should be replaced by "`none`", "`gl`", or "`opengl`" depending on which graphics library you want to use.

For example, if you are compiling on an SGI using the OpenGL library, the include line should read:

```
include /net/hg43/vrgroup/sve/v2.0/makeinc.sgi.opengl
```

In addition, you will need to set "PROGRAM" and "SRC" to be the program named and source file(s) of the application.

SVE Initialization

The SVE_init function has two additional arguments which are a pointer the number of command line arguments and the array of command line arguments (from the parameters of the "main" function). These are used by X (if appropriate) to set geometry, display, etc. from the standard X command line arguments.

```
boolean SVE_init(char *programName, SVE_config config, int *argc, char *argv[]);
```

The `SVE_GOURAUD` flag, which used to provide gouraud shaded polygons with light sources, now only give gouraud shaded polygons. To enable light sources, the flag `SVE_LIGHTING` should be used. The flag `SVE_LIT_GOURAUD`, which is an or combination of `SVE_GOURAUD` and `SVE_LIGHTING`, can be used in place of the old `SVE_GOURAUD` flag to achieve the same effect.

Events

The event type for event callbacks have changed. For example, `KEYBD` should be replaced by `SVE_KEY_PRESS`, and `LEFTMOUSE` should be replaced by `SVE_LEFT_MOUSE`. See the `include/event.h` file for additional event types.

The event data is represented differently. The `eventVal` field of the `SVE_state` structure has been replaced by a pointer, `eventData`, which points to an event structure containing the data relevant for the event. The possible event structures are given in the `include/event.h` file. For example, here are the old and new ways of getting a key press event, a mouse button event, and an object selection event:

```
=====================================================================
OLD (KEYBD event):
 SVE_status handleKey(SVE_state state)
 {
     switch(state->eventVal) {
     }
 }

NEW (SVE_KEY_PRESS event):
 SVE_status handleKey(SVE_state state)
 {
     if (state->eventType == SVE_KEY_PRESS) {
         switch(((SVE_keyEvent *)state->eventData)->keyVal) {
         }
     }
 }


=====================================================================
OLD (LEFTMOUSE event):
 SVE_status handleMouse(SVE_state state)
 {
     if (state->eventVal == 1) {
     }
 }

NEW (SVE_LEFT_MOUSE event):
 SVE_status handleMouse(SVE_state state)
 {
     if (SVE_IS_PRESS_EVENT(state->eventType)
         && ((SVE_stateChangeEvent *)state->eventData)->pressed) {
     }
 }


=====================================================================
OLD (SVE_OBJECT_SELECTION event):
 SVE_status objectSelect(SVE_state state)
 {
     SVE_object selectedObject;

     selectedObject = state->selectedObject;
 }

NEW (SVE_OBJECT_SELECTION event):
 SVE_status objectSelect(SVE_state state)
 {
     SVE_object selectedObject;

     if (SVE_IS_OBJECT_EVENT(state->eventType)) {
         selectedObject = ((SVE_objectEvent *) (state->eventData))->object;
     }
 }

=====================================================================
```

GL Functions

Irix GL functions and data types must be replaced. Matrix data is now represented by the type "`M_matrix`". Valid matrix operations can be seen in the file `include/matrix.h`. Object position matrixes should be manipulated through SVE functions (`SVE_rotateObject()`, `SVE_scaleObject()`, etc.).

SVE Function Changes

The `SVE_initGeometryChange()` function takes one more parameter: an `SVE_object`. This object field is optional, and can be `NULL`. However, when changing the shape of a geometry which is part of an

object, that object needs to re-generate its boundaries. This function will do that automatically, if you pass in the `SVE_object`. You can also do this yourself with the `SVE_reCalculatedObjectBoundaries()` function.

The previous discussion is also true of the `SVE_addPrimitiveToGeometry()` function, which takes an additional `SVE_object` parameter for the same reason.

The `SVE_registerWidgetType()` function takes an additional function parameter that is called when a widget instance is deleted. The delete function is of type `SVE_widgetFunctionPtr`.

Rendering Changes

The (undocumented) feature of Wavefront texture materials using the texture as an alpha mask when the "d" value is 1 (an early attempt at transparent textures) has been removed. Use 2 or 4 component texture maps to achieve transparency effects.

Object File Change

The SVE object file format has changed slightly. Old SVE object files (version 1.0) are not readable by a version 2.0 applications. Old SVE object files can be converted to the new format (version 1.1) using the "`sveObjTo11`" utility found in the SVE bin directory.

4. Future Directions

In the near future, the following additions will be made by members of the VE-group.

- Conversion routines from/to other VR systems and CAD modelers.
- Spatial sound cues.
- Model based rendering optimizations
- Networking and shared environments.
- Voice and text annotation
- Scripts and dynamics/kinetics.

Scripts and dynamics/kinetics are very interesting solutions for managing the interaction between the user and the objects and adding behavior to objects (here at the GVU center, we have quite a few people working on dynamics, simulation and animation.).

We are currently working on allowing SVE applications to be a part of a shared virtual environment.

APPENDIX A: Starter Kit
The Simple Virtual Environment Library Starter Kit, Version 2.0

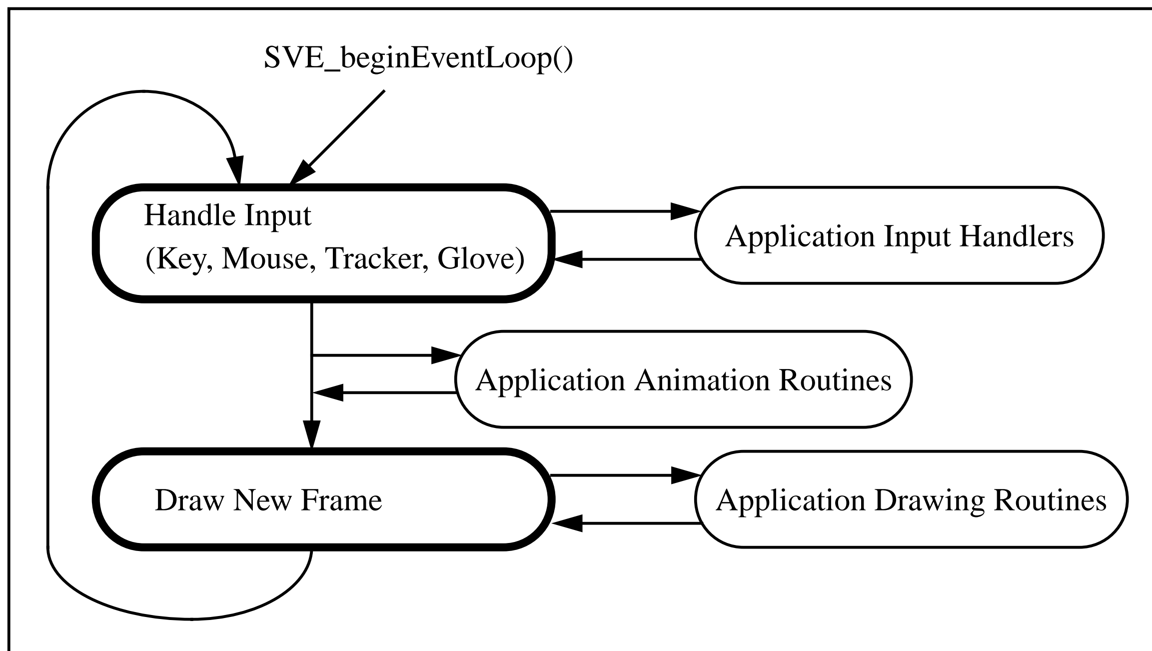
Drew Kessler, Rob Kooper, Larry Hodges
drew@cc.gatech.edu, kooper@cc.gatech.edu, hodges@cc.gatech.edu

1. Introduction

The Simple Virtual Environment Library (SVE) is intended to provide a system of functions, event handlers, and a 3D description of a Virtual World that allows for easy creation of simple Virtual Environment (VE) applications and provides an extensive, straight-forward addition of functionality. What this means is that, using the library, you can create a simple Virtual World which has minimal interaction (a pure Architectural walkthrough, for example), but is extensible enough to add additional features (such as allowing a user to move the kitchen sink) in a straight-forward and consistent manner.

2. General Overview

The SVE system works using a "don't call us...we'll call you" method (also known as a callback method). Basically the application initializes the SVE system, gives the world description filename, specifies any special cases it wants to handle itself, and then turns control over to the SVE system. When the application relinquishes control, this is what happens:



The steps which are circled by bold lines are done by the SVE system. The steps circled by regular lines are callback routines written by the application programmer which will be called, if they exist, at the given moments during the execution cycle.

3. An Example

What follows is a very simple example application with a complete explanation following it. The file is called "`example1.c`" and resides in the directory "`~vrgroup/sve/v2.0/examples/example1`". If you wish to copy it and compile it, be sure to copy the files `Makefile`, `example1.world`, and

.sve.init which are in the examples/example1 directory as well. Note that SVE looks for the description file and any object files it needs: first in the current directory, and then in any directories specified in the .sve.init file (on the defaultObjectDirectory line). Thus, you need not copy the object files unless you wish to modify them. A section describing these file formats follows later in this manual.

To compile the example, you will need to make one change to the Makefile which you copied from the examples/example1 directory. The line that begins with "include .../makeinc..." needs to be changed to use the correct, full path name of one of the "makeinc" files in the SVE root directory. The file that should be included depends on what platform you are compiling on, and what 3D graphics library should be used by SVE to render the scene. For example, if the SVE root directory is "/net/hg76/vrgroup/sve/v2.0/", and you are compiling on an SGI workstation, and wish to use the OpenGL graphics library, then this line should read:

```
include /net/hg76/vrgroup/sve/v2.0/makeinc.sgi.gl
```

In addition to the Makefile, you will need to change the .sve.init file to indicate where the SVE system should look to find the objects that the application needs. In this case, given the SVE root directory mention before, the DefaultObjectDirectory line of the .sve.init file should read:

```
DefaultObjectDirectory /net/hg76/vrgroup/sve/v2.0/objects
```

After these two changes, you should be able to type "make" at the command line, and the example1.c file that you copied will be compiled into an executable called "example1", which you can run.

```
/***********************************************************************
 * Example1 (sve module)
 *
 * This example shows a simple Virtual Environment using SVE. The default
 * key's as described in the SVE manual will work with this example.
 *
 ***********************************************************************/

#include "sve.h"


main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

/***********************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the SVE BASICS section of
 * the manual for a description of the different configurations you can use.
 ***********************************************************************/
   printf("Starting application\n");
   SVE_init("Example1 (sve)", config, &argc, argv);

/***********************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 ***********************************************************************/
   if(!SVE_loadWorld("example1.world"))
   {
      printf("Error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/***********************************************************************
 * SVE will take over control of the program until it is finnished.
 ***********************************************************************/
```

```
    printf("Beginning event loop.\n");
    SVE_beginEventLoop();

    printf("Done -- Have a nice day.\n");
    SVE_done();
}
```

This application initializes the SVE system with the call `SVE_init`(), giving the application name and its configuration (`SVE_NORMAL` - this means no trackers or other special features are used). The application then loads the Virtual World described in the "`example1.world`" file with the call `SVE_loadWorld`(). The SVE system loop (described in the Overview above) is begun with a call to `SVE_beginEventLoop`(). If the users presses 'q' in the application window, or the program aborts for any reason, control returns to the application and `SVE_done`() is called to shut down the SVE system.

You will notice that when the cursor is in the application window (the one displaying the graphics), that you can alter your view by pressing the arrow keys (to change X and Z), pressing 'y' or 'Y' (shift-'y') to change Y, and by pressing and dragging the mouse to rotate the scene. If you press 'q', the application shuts down.

At this point, it should be noted that configurations, such as window size, window position, and many other things in addition to default directories can be specified in an initialization file. This file is generally named `.sve.init` and should be located in the directory from which the application is run. The `sve.init` file in the `examples` directory provides an example of this file. To be used, it must be renamed to `.sve.init` (note the preceding dot in the name). The format and options of this file are covered in detail in the User's Guide.

4. Second Example

In the second example ("`example3.c`" in the `examples/example3` directory), we introduce the user specified input handler and user defined animation. These routines are called in addition to SVE's own input handler. In this example application, we have decided to alter a specific object in the Virtual World each time a frame is rendered, depending on keyboard input from the user. If the "r" key is pressed the cube in the scene will rotate to the right, if the "l" key is pressed the cube will rotate to the left, and if the space bar is pressed the cube will stop rotating completely. If you wish to compile this program, you need to copy the files `example3.c`, `example3.world,` and `Makefile` from the `examples` directory and type "`make example3`" (after making the appropriate change to the `Makefile` and `.sve.init` file as discussed for the first example). The code follows.

```
/************************************************************************
 * Example3 (sve module)
 *
 * This is the example from the SVE manual.
 *
 * This example will show the power of animation callback functions. It
 * uses an event callback function to check the keyboard. When the 'r'
 * is pressed the object, a cube, will spin around. When the 'l' is
 * pressed the object will spin the otherway around. It will keep on
 * spinning until the spacebar is pressed. The default key's as described
 * in the SVE manual still work.
 ************************************************************************/

#include "sve.h"

/************************************************************************
 * A pointer to the object to be rotated.
 ************************************************************************/
SVE_object cube;
```

```
/**********************************************************************
 * Thi animation callback function is called when you pressed an 'l'. It
 * will rotate the cube around the y-axis.
 **********************************************************************/
SVE_status rotate_right(SVE_state state)
{
   SVE_rotateObject(cube, 3, 'y');
}


/**********************************************************************
 * This animation callback function is called when you pressed an 'r'. It
 * will rotate the cube the other way around the y-axis.
 **********************************************************************/
SVE_status rotate_left(SVE_state state)
{
   SVE_rotateObject(cube, -3, 'y');
}


/**********************************************************************
 * This function handles the callback from the SVE_KEY_PRESS event. When you
 * press 'l' or 'r' it will SVE to use the correct animation callback.
 **********************************************************************/
SVE_status handleKey(SVE_state state)
{
   SVE_status retval = EVENT_IGNORED;

   if (state->eventType == SVE_KEY_PRESS) {
      switch(((SVE_keyEvent *)state->eventData)->keyVal) {
      case 'r': SVE_addAnimationCallback(rotate_right);
                retval = EVENT_CONSUMED;
                break;
      case 'l': SVE_addAnimationCallback(rotate_left);
                retval = EVENT_CONSUMED;
                break;
      case ' ': SVE_removeAllAnimationCallbacks();
                retval = EVENT_CONSUMED;
                break;
      }
   }
   return(retval);
}


main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL;

/**********************************************************************
 * Initialize SVE. This should always be the first call to SVE. This will
 * tell SVE what configuration to use. Look at the SVE Basics section of
 * the manual for a description of the different configurations you can use.
 **********************************************************************/
   printf("Starting application.\n");
   SVE_init("Example3 (sve)", config, &argc, argv);
```

```
/**********************************************************************
 * Load in the world file. This function returns FALSE when the world
 * could not be loaded correctly.
 **********************************************************************/
   if(!SVE_loadWorld("example3.world"))
   {
      printf("error occured during SVE_loadWorld, exiting.\n");
      SVE_done();
   }

/**********************************************************************
 * Find in the world an object called cube.
 **********************************************************************/
   cube = SVE_findWorldObject("cube");

/**********************************************************************
 * Tell SVE that we are interested in the SVE_KEY_PRESS event, and tell
 * SVE which function will handle this callback.
 **********************************************************************/
   printf("Registering an input callback.\n");
   SVE_registerCallback(SVE_KEY_PRESS, handleKey);

/**********************************************************************
 * SVE will take over control of the program until it is finnished.
 **********************************************************************/
   printf("Beginning event loop\n");
   SVE_beginEventLoop();

   printf("Done -- Have a nice day.\n");
   SVE_done();
}
```

This example demonstrates many SVE concepts. We have shown that an object in the SVE world object tree can be retrieved by the character string name it is given in the description file (using the `SVE_findWorldObject()` function). The value returned can be used as a reference to that object. while the object is still present in the object tree, it will be rendered in each frame.

We have also demonstrated a few callback routines. We have an input handler, `handleKey`, which will be called every time a key is pressed and which checks for an 'l', 'r', or space bar input. This is done with a call to `SVE_registerCallback()` with SVE_KEY_PRESS as the first parameter.

When an 'r' or 'l' key is pressed, we have added an animation callback to a function which will rotate the cube for each frame. This is accomplished with a call to `SVE_addAnimationCallback()`. When the space bar is pressed, the list of animation callbacks is cleared with the function `SVE_removeAllAnimationCallbacks()`. It is in this animation callback where the application can alter the object tree model that SVE stores and renders each frame. Similar routines can be used to draw something that the SVE system would not draw automatically from the object's definition. These are called frame callbacks.

Event, animation, and frame callback functions used in SVE (with some exceptions shown in the User's Guide) need to be declared in the format:

```
SVE_status functionName(SVE_state state)
```

The `SVE_state` data structure contains the current state within SVE, including the definition of the world, the user's viewpoint and orientation, and the most recent events (such as `eventData`, as used above). See the User's Guide for more details.

5. Using Tracking Devices

Most VE applications would not be complete without head tracking, which affects the point of view from which the scene is rendered to the screen or head mounted display. For an SVE application, adding head tracking by a supported tracking device is easy. In terms of programming, only one change needs to be made: the config parameter to SVE_init() should include the SVE_HMD or SVE_TRACKER flag. For example, to change the previous examples to use a tracker, and display the results on a head mounted display, the line that sets the config variable should read:

```
SVE_config config = SVE_NORMAL | SVE_HMD;
```

(This is usually done as a result of a command line parameter, though, so that applications can be run with and without using the trackers without re-compiling.)

Once the application code has been change and compiled, then two additional things must be done. First, the application must be told which tracking device to use, which receiver of that device to use, and what machine that tracker is attached to. This is done by adding a "`tracker`" line to the `.sve.init` file. As an example, the following line, if it appeared in the `.sve.init` file:

```
tracker 1                buckhead isotrakii /dev/ttyd2 1 SVE HMD
```

would be translated as, tracker 1, which is attached to a machine called buckhead, is a Polhemus IsotrakII and is attached to the /dev/ttyd2 serial port. The tracker receiver to use is the first one, and the tracker should move the "SVE HMD" object (which represents the user's head in the virtual world).

Lastly, a program called a "tracker server" must be run on the machine to which the tracker device is attached (even if it is the same machine that the example is going to be run on). The program is called "`server-tracker`" and can be found in the SVE `bin` directory. After that is running, then the example executable can be run. The application and tracker server should connect, the tracker device should be contacted, and the tracker should control the head position and orientation. When the application is done, then the tracker server program should quit automatically after a few minutes. If it does not go away on its own, it can be killed using a program called "`kill-servers`", which is also in the SVE `bin` directory.

6. Hardware Set Up

Currently the hardware is set up so that the trackers and head mounted display (HMD) can be turned on simply by flipping the red switch on the power strip in the VR cabinet (and the one on the floor beside the cabinet if you wish to use the Polhemus trackers). Since the HMD has a limited lifetime, it should be turned off when not in use. If the HMD is being used, the first tracker should be attached to the front or top of the HMD.

7. Where to Find Additional Help

Additional documentation can be found in the directory "`~vrgroup/sve/v2.0/doc`". In particular, the User's Guide contains a complete description of the system with a function and date structure reference.

In addition, the manual and other information on SVE is available on-line at:

```
http://www.cc.gatech.edu/gvu/virtual/SVE/SVE.html
```

APPENDIX B: File Formats

1. Introduction

The SVE system uses many different files to allow for specification of the environment used by the application, the application's configuration (including what devices to use, where to find them, and how to present the environment to the user), and specifying how a device is used (for example, by defining the gestures that a hand device might make that the VE application might want to recognize). The format of these files are defined in this appendix, first by example, then through a formal description.

For the formal descriptions, the following notation is used:

{} specifies a list of alternatives (1 must be chosen),

[] denotes an optional section,

[]* a section that can occur zero or more times,

[]+ denotes a section that occurs one or more times.

Basic elements:

↵ = newline

<n> = integer

<f> = float

<b> = boolean {'true', 'false', 't', 'f', 'yes', 'no', 'y', or 'n'}

<AXIS>= axis specification ('x', 'y', or 'z').

<string>= string, any character in the range [32..] (including single spaces, but no control characters).

<stringID>= string, any character in the range {['0'..'9'], ['a'..'z'], ['A'..'Z'], '_'} (no spaces).

Boldfaced and capitalized entries refer to other grammatical expression.

Indentation is for clarity only, it is not necessary.

Note that the file reading routines will treat the '#' sign as the beginning of a comment which continues to the end of the current line, and is ignored.

1.1. Environment Description Files - World and Objects

There are two different file formats used to describe the Virtual World which SVE renders: the *world description file* (of which there is only one), and *object description files* (of which there can be many). The world description file refers to all the objects that are used and their positions in the virtual scene. These are listed in a tree (for hierarchical grouping). Each of the object entries has a transformation matrix to convert the object coordinates to the world coordinate system. The object description files describe the geometry of the individual objects, which can be made up of many primitives (*polyhedrons*, *text*, *polylines* etc.).

The advantage of defining the geometrical information of the objects in separate files is that an object can be used among several worlds and in several places in the same world.

Here is the world description file used for the examples ("`example1.world`"):

```
Simple Virtual Object File Format version 1.0
number of objects: 2

object name: meadow
primitives file: plane.obj
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
```

```
0 -1 -1 1
other attributes: 0
number of children: 0

object name: text
primitives file: hello_world.obj
transformation matrix:
1 0 0 0
0 1 0 0
0 0 1 0
-0.5 0.5 -5 1
other attributes: 0
number of children: 0
```

Here is the object file, "`all_primitives.obj`", which contains the primitives *polyhedron*, *polyline*, *text*, *light* and *textured_polyhedron*:

Source xxi: Object File Example

```
Simple Virtual Primitive File Format  version 1.1
number of components: 5

component 1 type: light
Data of component 1:
no of attributes:
2
color      1.0  1.0   1.0
position   0.0  5.0 -10.0

component 2 type: polyhedron
Data of component 2:
no of vertices:
8
vertices: x y z
0      0      0
0      1.0    0
1.6    1.0    0
1.6    0      0
0      0      0.35
0      1.0    0.35
1.6    1.0    0.35
1.6    0      0.35
no of faces:
6
faces: R G B #_of_vertices v1 v2 v3 ...
255   0    0 4 0 1 2 3
  0 255    0 4 0 4 5 1
  0   0  255 4 1 5 6 2
255   0    0 4 2 6 7 3
  0 255    0 4 0 3 7 4
  0   0  255 4 4 7 6 5

component 3 type: polyline
Data of component 3:
no of vertices:
8
vertices: x y z
0      -0.1   0
0      -1     0
1.6    -1.0   0
1.6    -0.1   0
0      -0.1   0.35
0      -1.0   0.35
1.6    -1.0   0.35
1.6    -0.1   0.35
no of polylines:
6
polylines: R G B #_of_vertices v1 v2 v3 ...
255 255 255 4 0 1 2 3
255 255 255 4 0 1 5 4
255 255 255 4 1 2 6 5
```

```
255 255 255 4 2 3 7 6
255 255 255 4 3 0 4 7
255 255 255 4 4 5 6 7

component 4 type: text
Data of component 4:
transformation matrix:
0.4 0   0    0
0   0.4 0    0
0   0   0.3  0
0.1 1.5 0.36 1
no of lines:
1
***this is text***

component 5 type: textured_polyhedron
Data of component 5:
image file:       gvu.rgb
repeat texture:   TRUE
blending:         TRUE
no of vertices:
4
vertices: x y z
-0.3 -1.2 0.0
-0.3 -0.2 0.0
 0.3 -0.2 0.0
 0.3 -1.2 0.0
no of texture vertices:
4
texture vertices: u v
0 0
0 1
1 1
1 0
no of faces:
1
faces: R G B #_of_vertices v1 v2 v3 ...[t1 t2 t3 ...]
255 255 255 4 3 2 1 0 3 2 1 0
```

1.2. Initialization File

The initialization file is read when the SVE_init() routine is called to start the SVE system in motion, The default name for the initialization file is ".sve.init", although that can be changed by the application. The initialization file contains the information the SVE system needs to set up the application for a particular physical setup. The setup includes where the window is placed on the screen, how the view is rendered for a particular display, and where the application can find the files and the tracking and other input devices it needs. Here is an example initialization file:

Source xxii: Initialization File Example

```
# This file contains some default variables. The read routine is not
# case sensitive.

minX                     0
minY                     0
sizeX                    640
sizeY                    480
vofY                     700
AspectRatio              1.333
Near                     0.01
Far                      5000.0
ShowFrameRate            True
DefaultObjectDirectory   ../objects
DefaultWorldDirectory   ../objects
DefaultMaterialDirectory ../objects
tracker 1                buckhead isotrakii /dev/ttyd2 1 SVE HMD
tracker 2                buckhead isotrakii /dev/ttyd2 2 SVE cursor
```

1.3. Display Configuration File

The display configuration file defines the viewing parameters to use for a particular display. This may include a view plane that is rotated from perpendicular to the view direction. The boundaries of the viewing volume to use for a display can be specified in one of two ways. Either the field of view in the vertical direction (Y) and the aspect ratio of the horizontal field (X) of view to the vertical field of view can be given, or the size and location of the view plane can be given (where the view volume can be defined by planes intersecting the eyepoint and the edges of the view plane. Here is an example display configuration which defines a typical head mounted display:

Source xxiii: Example Display Configuration File, Head Mounted

```
# This file contains a definition of a view plane for an SVE
# application.
# This particular file defines the LEEP optic set up of the Virtual
# Research Flight Helmet.

ViewPlanePosition 0.0 0.0 -0.4
fovY 584
AspectRatio 1.289
```

The second example shows a configuration file which defines a typical tracker and monitor (or projection screen) setup:

Source xxiv: Example Display Configuration File, Monitor

```
# This file contains a definition of a view plane for an SVE
# application.
# This particular file defines a big monitor facing upwords, with the
# person viewing from  1.0 meter above.

ViewPlanePosition 0.0 -1.0 0.0
ViewPlaneRotation -90.0 0.0 0.0
ViewPlaneMinX -0.17
ViewPlaneMinY -0.13
ViewPlaneMaxX  0.17
ViewPlaneMaxY  0.13
```

1.4. Gesture Description File

The gesture file contains definitions of hand postures and gestures to be used as input events in the SVE system. Postures are static hand poses; gestures are hand poses that change in time. The gesture file allows for the definition of the angle states of the joints, a start hand pose, and a transition and end hand pose (if needed). Here is an example gesture file (from the examples directory). The “grab” gesture is a fist hand posture, the “pinky_wave” gesture is a straight pinky bending and then straightening out, while the other fingers are curled in.

Source xxv: Gesture File example

```
# This is a gesture file

# First define the default ranges used for joints

# 0 to 90 degree bends
joint_positions: 4
bounds:
-45 15 0 # min max consider_as
 15 45 30
 45 75 60
 75 135 90

thmb rot
thmb mphl
thmb ip
thmb abdt

indx mphl
```

```
indx pxip
indx dsip

midl mphl
midl pxip
midl dsip

ring mphl
ring pxip
ring dsip

pnky mphl
pnky pxip
pnky dsip

# Abduction angles: 0 to 30 degrees
joint_positions: 4
bounds:
-45 5 0 # min max consider_as
 5 15 10
 15 25 20
 25 75 30

midl abdt
ring abdt
pnky abdt

# Next, define gestures

name: grab
thmb rot 0.8 0.8 0.9 1.0 # 0, 30, 60, 90 degrees
thmb mphl 0.9 0.9 1.0 1.0
thmb ip 0.9 0.9 1.0 1.0
# thmb abdt 0.5 0.7 0.9 1.0

indx mphl 0.5 0.6 1.0 1.0
indx pxip 0.5 0.6 1.0 1.0
indx dsip 0.5 0.6 1.0 1.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.5 0.6 1.0 1.0
midl dsip 0.5 0.6 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
ring pxip 0.5 0.6 1.0 1.0
ring dsip 0.5 0.6 1.0 1.0

pnky mphl 0.5 0.6 1.0 1.0
pnky pxip 0.5 0.6 1.0 1.0
pnky dsip 0.5 0.6 1.0 1.0

name: point
indx mphl 1.0 0.9 0.0 0.0
indx pxip 1.0 0.9 0.0 0.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.0 0.0 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
ring pxip 0.0 0.0 1.0 1.0

pnky mphl 0.5 0.6 1.0 1.0
pnky pxip 0.0 0.0 1.0 1.0

name: pinky_wave
indx mphl 0.5 0.6 1.0 1.0
indx pxip 0.0 0.0 1.0 1.0

midl mphl 0.5 0.6 1.0 1.0
midl pxip 0.0 0.0 1.0 1.0

ring mphl 0.5 0.6 1.0 1.0
```

```
ring pxip 0.0 0.0 1.0 1.0

pnky mphl 0.0 0.0 1.0 1.0
pnky pxip 0.0 0.0 1.0 1.0

transitions:
pnky mphl backward
pnky pxip backward

end:
pnky mphl 1.0 1.0 0.0 0.0
pnky pxip 1.0 1.0 0.6 0.0

name: thumbs_up
priority: 10
thmb rot 1.0 0.8 0.0 0.0
thmb mphl 1.0 1.0 0.0 0.0
thmb ip 1.0 1.0 0.0 0.0
thmb abdt 0.0 0.5 0.9 1.0

indx pxip 0.0 0.0 1.0 1.0

midl pxip 0.0 0.0 1.0 1.0

ring pxip 0.0 0.0 1.0 1.0

pnky pxip 0.0 0.0 1.0 1.0
```

2. Formal Definition of the World Description File

This section describes the world file format in grammar notation. The following notation is used:

WORLDFILE:

```
Formal Description of file format   version 1.1↵
[user position: <f> <f> <f>]↵
[user speed: <f>]↵
number of objects: <n>↵
↵
[OBJECT_ENTRY]*
```

The worldfile specifies all of the objects that are to be used in the virtual environment. The only important entry in the header is the *number of objects*. The integer given should be equal to the number of root-objects in the scene (see the object entry for more about root objects etc.) The *user position:* and *user speed:* lines are optional. They specify a starting position for the user and a speed for when the user is flying.

OBJECT_ENTRY:

```
object name: <stringID>↵
PRIMITIVES-FILE-ENTRY+
transformation matrix:↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
other attributes: <n>↵
[ATTRIBUTE-ENTRY]*
number of children: <n>↵
[OBJECT_ENTRY]*
```

The object is specified by a primitives file and a transformation matrix that transforms all the points of the object (and its children) to world coordinates. The matrix defines how the object will be placed in the virtual scene: it enables translation, rotation and scale transformations (see "Fundamentals of Interactive Computer Graphics", Foley and van Dam).

Extra attributes can be specified for each object. Objects can be hierarchically grouped by specifying children. This is specified in a depth-first notation, as show in Figure 19.

PRIMITIVES-FILE-ENTRY

```
{
primitives file: <stringID>↵,
primitives file: <stringID> valid to <f>↵,
primitives file: <stringID> valid from <f>↵,
primitives file: <stringID> valid from <f> to <f>↵
}
```

The primitives file entry defines the geometry of the object. It can be a Wavefront .obj file or an SVE primitives file (look at the object description for the file format of these) or "`NULL`", which results in an empty object which will not be rendered. A list of "primitives file:" lines can be given. The list will be

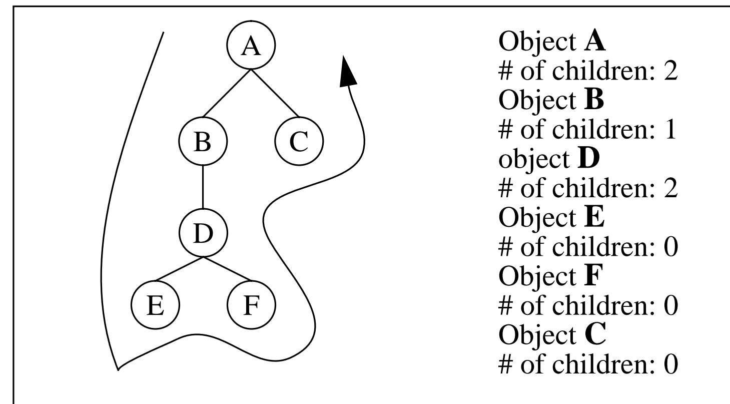


Figure 19. Depth first notation of a tree hierarchy.

searched from first to last for a valid geometry description, and that one (and only that one) will be rendered. A geometry description is valid if it is not followed by a "valid" keyword, or if the object is close enough (for the "valid to" specification), far enough (for the "valid from" specification), or within a range (for the "valid from ... to ... " specification). The floating values are distances in meters.

ATTRIBUTE-ENTRY:

```
{
color COLOR_ENTRY↵,
visible <b>↵,
visible_sphere <f>↵,
selectable <b>↵,
highlight_primitives_file: <stringID>↵
cullable <b>↵
facing_viewer_upright↵
facing_viewer↵
title <f> <f> <f> <f> <string>↵
linewidth <n>↵
}
```

Currently, these extra attributes can be assigned to objects:

- color: this color will override the original color of the object primitives. This function is useful during highlighting etc.
- visible: the object and its children are invisible when the flag FALSE is used.
- visible_sphere: the object is only rendered when the viewpoint is less than <f> meters.
- selectable: this will set the boolean Selectable in the object structure.
- highlight_primitives_file: gives an object file which defines the geometry for an object when it is highlighted.
- cullable: if TRUE, then the object will not be drawn if the bounding volume of the object is outside of the view of the user. If FALSE, then the object will always be drawn (and no work will be done to determine if it is in the user's view or not).
- title x y z s string: a text string will be rendered on the x,y,z (object) coordinates with size s.
- linewidth: is used to change the width of polylines and text primitives.
- facing_viewer_upright: the object will orient itself to always face in the viewer's direction, rotating only in the X-Z plane.

• facing_viewer: the object will orient itself to always face the viewer.

COLOR_ENTRY:

```
<f> <f> <f>
```

Red, green and blue values in the range [0,1] (entries larger than 1.0 will be divided by 256!)

3. Formal Definition of the Object Description File

OBJECT FILE:

```
Simple Virtual Primitive File Format version 1.1↵
[BOUNDARY-ENTRY]
number of components: <n>↵

[PRIMITIVE-ENTRY]+
```

BOUNDARY-ENTRY:

```
{bounding box <f> <f> <f> <f> <f> <f>↵,
 bounding sphere <f> at <f> <f> <f>↵}
```

Each object can have a boundary description that can be used for collision detection[1] etc. These entries are done in object coordinates, currently two descriptions can be used:

- bounding box: 2 points that determine the box, the lower left and the upper right corner.
- boundary sphere: is determined by its radius. Because the origin of the object can be at a different coordinate than (0,0,0), three additional parameters are needed to determine the x,y,z - origin of the sphere.

By default, the object will calculate its boundaries when the file is read in if none is given. The calculated bounding volume will be a bounding box, as it is more straight forward to calculate even if objects are stretched non-uniformly and rotated. Most SVE routines that use bounding volumes are optimized for bounding boxes.

PRIMTIVE-ENTRY:

{**LIGHT,**
POLYHEDRON_ENTRY,
POLYLINEN_ENTRY,
TEXT_ENTRY,
TEXTURED_POLYHEDRON_ENTRY}

LIGHT

```
component <n> type: polyhedron↵
Data of component <n>:↵
no of attributes:↵
<n>↵
[ambient COLOR_ENTRY↵]
[color COLOR_ENTRY↵]
[position <f> <f> <f>↵]
[local-light <b>↵]
[spot-direction <f> <f> <f>↵]
[spot-light <f> <f>↵]
```

1. These algorithms are currently still under development, the descriptions are only loaded into the boundaries-entry of the object.

The light primitive is based on the light source used in the GL and OpenGL renderers. Those that are already familiar with these light sources will probably recognize the names and their meanings.

- ambient: Specifies the background color of the light source. This light is always there also when it is not reaching the object. The numbers represent the Red Blue and Green components.
- color: Specifies the colorvalues (in Red, Green and Blue) that the light source emits. This light is used to calculate the color of the objects.
- position: The position in the Virtual world where the light source is located.
- local-light: If this is set to TRUE, the light source will be a local light source. If this value is FALSE, the light source will be at infinity (pointing at the origin from its position).
- spot-direction: The direction the spotlight emits its light.
- spot-light: The first number describes the intensity as a function throughout the cone and the second number tells how wide the cone is in degrees.

POLYHEDRON_ENTRY:

```
component <n> type: polyhedron↵
Data of component <n>:↵
no of vertices:↵
<n>↵
vertices: x y z↵
[<f> <f> <f>↵]+
no of faces:↵
<n>↵
faces: R G B #_of_vertices v1 v2 v3 ...↵
[COLOR_ENTRY <n> <n> <n> [<n>]+ ↵]
```

The polygons are defined by referring to the vertices lists. These indices begin at 0 (zero).

POLYLINE_ENTRY:

```
component <n> type: polyline↵
Data of component <n>:↵
no of vertices:↵
<n>↵
vertices: x y z↵
[<f> <f> <f>↵]+
no of polylines:↵
<n>↵
polylines: R G B #_of_vertices v1 v2 v3 ...↵
[COLOR_ENTRY <n> <n> <n> [<n>]+ ↵]
```

Polylines are a series of lines. The series is not automatically "closed." If the polyline should be a closed line, then the line from the first to the last point should be included.

TEXT_ENTRY:

```
component <n> type: text↵
Data of component <n>: ↵
transformation matrix:↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
<f> <f> <f> <f>↵
no of lines:↵
<n>↵
[<string> ↵]+
```

The transformation matrix determines the position, orientation and scale of the text relative to the other primitives of that same object. Text strings are always rendered in white, unless it is overridden the (by having the extra attribute *color* in the world file format).

TEXTURED_POLYHEDRON_ENTRY:

```
component <n> type: textured_polyhedron↵
Data of component <n>:↵
image file: <stringID>↵
repeat texture: <b>↵
{
blending: {TRUE, FALSE, GREYSCALE}↵
texture environment: {default, intensity, intensity_alpha, rgb,
                    rgb_lighting, rgb_alpha, rgb_alpha_lighting}↵
}
no of vertices:↵
<n>↵
vertices: x y z↵
[<f> <f> <f> ↵]+
no of texture vertices:↵
<n>↵
texture vertices: u v↵
[<f> <f> ↵]+
no of faces:↵
<n>↵
faces: R G B #_of_vertices v1 v2 v3 ... [t1 t2 t3 ...]↵
[COLOR_ENTRY <n> [<v>]+ [<t>]+↵]
```

The textured polygon entry is similar to the regular polyhedron, with its only major difference being the addition of three more lines at the beginning of the definition. These lines contain a list of two dimensional points which map a texture on a face, an index to that list for each vertex index in the face definition, and the name of the image file to map.

For mapping the texture map on the polygons, each vertex has two extra coordinates that refer to the (x,y) coordinates of a 2D plane with a grid of copies of the image which are side by side. These values are typically called *u,v* or *s,t* coordinates and the image is defined in the range [0...1]. A of list texture vertices made up of (0,0), (0,2), (2,2), (2,0) will result in four images on the face being defined (see Figure 20.). A texture vertices list of (0,0), (0,0.5), (1,0.5), (1, 0) will result in the lower half of the image to be displayed on the face. A polyhedron face is therefore defined as its color values, the number of vertices, a list of

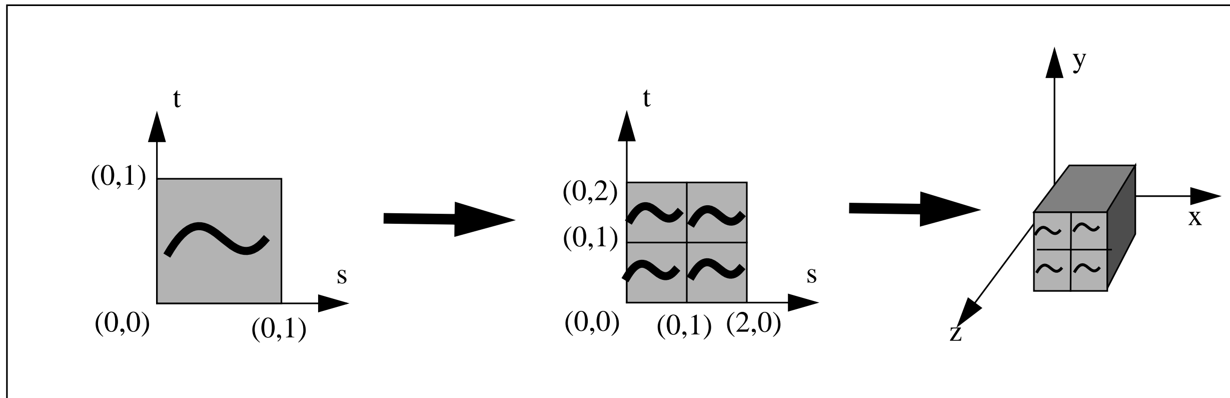


Figure 20. Mapping s,t Coordinates to Polygons

indexes to the vertex list to define the vertices of the face, and then a list of indexes into the texture vertices list defining, for each vertex in the vertex list.

The texture definition include three additional entries, *image file*, *repeat texture,* and either *blending* or *texture environment*:

- The *image file* entry must be a file that is of the SGI rgb format (a variety of applications can be used to convert .gif and other file formats to these[1]).
- The *repeat texture* flag determines whether the texture should be repeated when the s,t coordinates exceed the range [0,1].
- The *blending* flag is one way to determine how the texture is handled: TRUE will blend the colors of the texture map with the colors of the faces, FALSE will discard the face colors. The special flag value GREYSCALE will handle the texture map as a black-and-white image that is blended with the colors of the original faces. Using greyscale texture maps instead of color ones will increase the rendering performance and should be used as much as possible.
- The *texture environment* flag is the other way to determine how to handle the texture. The choices are:

default:

Used to specify that the default for the number of channels contained in the texture. This is the texture environment used by textured Wavefront objects. The default is labeled with a <D> in the following list. (They represent the old "blending: true" texture type for each channel).

intensity: <D>

1 Channel. The RGB color of the polyhedron is multiplied by the texture pixel value at each pixel.

intensity_alpha: <D>

2 Channel. The RGB color of the polyhedron is multiplied by the intensity value from the texture at each pixel, the alpha value of the polyhedron is multiplied by the alpha value from the texture at each pixel.

rgb:

3 Channel. Texture RGB color overrides polyhedron. Alpha of polyhedron is unaffected.

rgb_lighting: <D>

3 Channel. The RGB color of the texture is multiplied by the RGB color of the polyhedron.

1. Golden oldies are the *from* applications, for example *fromgif <sourcefile> <destfile>*, source of these can be found in the directory *~4Dgifts/iristools/imgtools* binaries are usually installed in */usr/sbin* on SGI machines.

rgb_alpha:

4 Channel. The RGB color of the texture and the RBG color of the polyhedron are combined proportionally to the texture's alpha value (Texture alpha value of 0 results in 100% polyhedron color, value of 1 results in 100% texture color). The polyhedron's alpha value is unchanged.

rgb_alpha_lighting: <D>

4 Channel. The RGB color and alpha values are multiplied by the RGB and alpha values of the texture for each pixel.

COLOR_ENTRY:

```
<f> <f> <f>
```

Red, green and blue values in the range [0 to 1.0] (entries larger than 1.0 will be divided by 256!)

4. Formal Definition of the Initialization File

The options for the initialization file are converted to lower-case before they are examined.

INITIALIZATION-FILE:

```
[
   DIRECTORY-ENTRY ,
   VIEW-PARAMETER-ENTRY ,
   OBJECT-INITIALIZATION-ENTRY ,
   DEVICE-SPECIFICATION-ENTRY ,
   CONFIGURATION-ENTRY ,
   SERVER-LOCATION-ENTRY
]+
```

DIRECTORY-ENTRY:

```
{
   defaultObjectDirectory <stringID>[:<stringID>]*↵,
   defaultWorldDirectory <stringID>[:<stringID>]*↵,
   defaultTextureDirectory <stringID>[:<stringID>]*↵,
   defaultMaterialDirectory <stringID>[:<stringID>]*↵,
   defaultConfigDirectory <stringID>[:<stringID>]*↵,
   defaultAudioDirectory <stringID>[:<stringID>]*↵,
   defaultServerDirectory <stringID>↵
}
```

All of these options specify a directory or directories in which to search for the given file types. For all options except `defaultServerDirectory`, the given string can be a single directory, or a list of directories separated by colons. The `defaultObjectDirectory` option specifies where to look for SVE or Wavefront object files. The `defaultWorldDirectory` option specifies where to look for SVE world files. The `defaultTextureDirectory` option specifies where to look for texture files. The `defaultMaterialDirectory` option specifies where to look for Wavefront material files. The `defaultConfigDirectory` option specifies where to look for display configuration files. The `defaultAudioDirectory` option specifies where to look for audio files. Finally, the `defaultServerDirectory` option specifies where to find the tracker, audio, and event server programs.

VIEW-PARAMETER-ENTRY:

```
{
   eyePosition <f> <f> <f>↵,
   eyeSeparation <f>↵,
   minX <n>↵,
   sizeX <n>↵,
   minY <n>↵,
   sizeY <n>↵,
   VofY <n>↵,
   aspectRatio <f>↵,
   near <f>↵,
   far <f>↵,
   DisplayConfig <stringID>↵,
}
```

These parameters define the rendered view of the environment, either by defining the eyepoint and other viewing parameters, or by defining the limits of the view volume and window placement on the screen. The meaning of each parameter is described below:

- The `eyePosition` option gives the (X, Y, Z) position of the display in relation to the 'SVE HMD' object, which usually follows the head tracking receiver. This value allows correction for the fact that the head tracking receiver is often above the head rather than at the eye's position.
- The `eyeSeparation` option gives the distance (in meters) between the objects representing the user's eyes for stereo viewing. Note that this is not necessarily the actual distance between the user's eyes, as the technology may require different values for the two images to be fused by the user's eyes into a stereo view.
- The `minX`, `sizeX`, `minY`, and `sizeY` options define the rendered window on the screen in screen coordinates, where the (`minX`, `minY`) coordinate defines the bottom left corner of the window.
- The `VofY` option gives the field of view in the Y direction in tenths of degrees.
- The `aspectRatio` option gives the aspect ratio of the width (in X) over the height (in Y) to determine the field of view in the X direction given the field of view in the Y direction.
- The `near` and `far` options give the near and far clipping plane locations, respectively, as distances from the eyepoint in meters.
- The `DisplayConfig` option gives the filename of the configuration file for the display. The display configuration file can contain a field of view/aspect ratio description or a screen dimension, position and rotation description. It will override the `VofY` and `aspectRatio` lines in this configuration file.

OBJECT-INITIALIZATION-ENTRY:

```
{
   move <string> to <f> <f> <f>↵,
   rotate <string> by <f> around <AXIS>↵,
   scale <string> by <f> {uniformly, along <AXIS>}↵,
   pointerObjectName <string>↵,
   pointerObjectFile <stringID>↵,
   selectorObjectName <string>↵,
   selectorObjectFile <stringID>↵
}
```

These options affect objects that may appear in the environment. The objects do not need to be in existence when the initialization file is read. For the object position options, when an object is created with the name given by these options, then the position is initialized with the given (possibly series of) position changes,

and the fact that this was done is reported by the object creation routine. A description of each option follows:

- The `move` option gives a translation positional change for an object specified by the string name. The (X, Y, Z) values are translations in meters in the local coordinate system of the object.
- The `rotate` option gives a rotation positional change about a given axis ('x', 'y', or 'z') for an object specified by the string name. The rotation is in degrees in the object's local coordinate system. Note that multiple rotatations for the same object can be specified by a series of `rotate` options, and that the order of the rotations is important (different orders may given different results).
- The `scale` option given a scale positional change for an object specified by the string name along a given axis ('x', 'y', or 'z') in the local coordinate system, or uniformly in all directions.
- The `pointerObjectName` option gives the name of an object which should be used instead of the regular pointer which is the object that follows the "SVE cursor" object when the `SVE_SELECT` option flag is set).
- The `pointerObjectFile` option gives the file name of the object description which should be used instead of the default pointer description. (The pointer is the object that follows the "SVE cursor" object when the `SVE_SELECT` option flag is set.)
- The `selectorObjectName` option gives the name of the object which should be used instead of the "ray" selector object (which "shoots" from the pointer object), when the `SVE_SELECT` option flag is set.
- The `selectorObjectFile` option gives the file name of the object description which should be used for the selector object if it is different from the default ray. This object is used for selection when the `SVE_SELECT` option flag is set.

DEVICE-SPECIFICATION-ENTRY:

```
{
    tracker <n> <stringID> <stringID> <stringID> <n> <string>↵,
    trackerHemi <n> <f> <f> <f>↵,
    glove <n> <stringID> <stringID> <string>↵
}
```

The `tracker` option specifies the tracking device that is to be used to control an SVE object. The first number is a unique identifier (and integer < 32 not used by another tracker specification). The three following strings are the name of the machine to which the tracking device is connected (and on which the tracker server is running), the type of tracker, and the serial port identifier. The next number is the receiver identifier (in a range 1 to the maximum number of receivers attached to the tracker). The last string is the name of the SVE object that the tracker controls (by affecting the object's local position matrix). The object does not need to be created at the time at which the program is begun. An example tracker specification would look like this:

```
tracker 1 mymachine fastrak /dev/ttyd2 2 SVE HMD
```

This line gives the specifications of tracker device 1, which is a "fastrak" tracker attached to "mymachine" through the serial port "/dev/ttyd2". The "SVE HMD" object will be controlled by receiver 2 of that device.

The `trackerHemi` option allows for the specification of the valid tracking hemisphere for tracking devices that require such a specification. The number identifies which tracking device (given in the tracker option line). The three float values define a vector that points in the direction of the center of the hemisphere within which the tracker device is correct.

The `glove` option specifies the details of a hand input device in a similar way to how the tracking devices are specified. The first number is unique identifier (an interger less than 32). The strings are the machine to which the glove device is attached, the serial port of that machine to which the device is attached, and the name of the SVE object to which the graphical representation of the hand will be attached.

CONFIGURATION-ENTRY:

```
{
   defaultTextureMap <string>↵,
   FPSupperLimit <f>↵,
   FPSlowerLimit <f>↵,
   hmd <b>↵,
   gouraud <b>↵,
   audio <b>↵,
   showFrameRate <b>↵,
   verbose <b>↵,
   debug <b>↵
}
```

These options define the initial configuration of the SVE application. Since the initialization file is read in the `SVE_init()` routine, configurations given in the initialization file will override the configuration specified for the routine. Descriptions of each of these configuration options are described below:

- The `defaultTextureMap` option gives the file name of the texture that will be used on a textured polyhedron which has no texture defined for it (or the texture defined could not be successfully loaded).
- The `FPSupperLimit` and `FPSlowerLimit` options define the upper and lower limit to the frame rate, respectively, which the SVE system will try to maintain. Currently, only the upper limit is enforced, where the SVE application will not be rendered at a faster frame rate than the given value (in frames per second).
- The `hmd` option determines if the `SVE_HMD` configuration option will be set or not.
- The `gouraud` option determines if the `SVE_LIT_GOURAUD` configuration option will be set or not.
- The `audio` option determines if the `SVE_NOAUDIO` configuration option will be set (false) or not (true).
- The `showFrameRate` option determines if the SVE system will report to standard output the current frame rate (in frames per second), and other rendering information, at a regular interval.
- The `verbose` option determines if the SVE system will print informative messages about the files it has read and important stages it has passed.
- The `debug` option determines if debugging messages will be printed.

SERVER-LOCATION-ENTRY:

```
{
   eventServer <stringID>↵,
   audioServer <stringID>↵,
   VRmachine <stringID>↵,
   worldServer <stringID>↵
}
```

These options indicate which machines are running the given server type for the application that is executed. The `eventServer` and `audioServer` option specify which machine is running the event server and audio server respectively. The `VRmachine` option gives a default for the `eventServer` and `audioServer` options, if they are not used. The `worldServer` option indicates which machine is running a world server. The world server shares the SVE objects of all SVE applications that connect to it. It is a work in progress and is not guaranteed to work.

5. Formal Definition of the Display Configuration File

The options for the display configuration file are converted to lower-case before they are examined.

DISPLAY-FILE:

[
VIEW-PLANE-POSITION-ENTRY ,
VIEW-PLANE-SIZE-ENTRY ,
VIEW-VOLUME-ENTRY,
EYE-PLACEMENT-ENTRY
]+

VIEW-PLANE-POSITION-ENTRY:

```
[
    ViewPlanePosition <f> <f> <f>↵,
    ViewPlaneRotation <f> <f> <f>↵
]+
```

The `ViewPlanePosition` and `ViewPlaneRotation` options define the position of the view plane's origin (usually corresponds to the center of the screen), and the X, Y, and Z rotations applied to the view plane (in that order), respectively.

VIEW-PLANE-SIZE-ENTRY:

```
[
    ViewPlaneMinX <f>↵,
    ViewPlaneMinY <f>↵,
    ViewPlaneMaxX <f>↵,
    ViewPlaneMaxY <f>↵
]+
```

The `ViewPlaneMinX` and `ViewPlaneMinY` options define the bottom left corner of the view plane in relation to the view plane's origin. The `ViewPlaneMaxX` and `ViewPlaneMaxY` options define the top right corner of the view plane in relation to the view plane's origin.

VIEW-VOLUME-ENTRY:

```
[
    fovY <n>↵,
    AspectRatio <f>↵
]+
```

The `fovY` option defines the field of view of the display in degrees across the vertical axis. The `AspectRatio` defines the field of view of the display across the horizontal axis in terms of the field of view of the display across the vertical axis.

EYE-PLACEMENT-ENTRY:

```
[
   eyePosition <f> <f> <f>↵,
   eyeSeparation <f>↵,
]+
```

These parameters define the rendered view of the environment by defining the eyepoint. The meaning of each parameter is described below:

- The `eyePosition` option gives the (X, Y, Z) position of the display in relation to the 'SVE HMD' object, which usually follows the head tracking receiver. This value allows correction for the fact that the head tracking receiver is often above the head rather than at the eye's position. Note that the eye position and the view plane position should not be the same (as this would result in an undefined view direction). To avoid this problem, be sure to position the view plane a significant distance from the eye position (generally in the negative Z direction).
- The eyeSeparation option gives the distance (in meters) between the objects representing the user's eyes for stereo viewing. Note that this is not necessarily the actual distance between the user's eyes, as the technology may require different values for the two images to be fused by the user's eyes into a stereo view.

6. Formal Definition of the Glove Gesture File

GESTURE_FILE:

```
[
   [JOINT_DEFINITION]*↵
   [GESTURE_DEFINITION]*↵
]*
```

The gesture file consists of the joint definition (a list of states and a mapping from angle to state), and a list of gestures associated with that joint definition. There can be many gestures for a joint definition, and many joint definitions can be defined. Note that if no joint definition is given, the default definition or the last one defined will be used.

JOINT_STATE:

```
<f> <f> <f>
```

The joint state is the range (minimum and maximum) of angles and the value of that range.

JOINT_DEFINITION:

```
joint_positions: <n>↵
bounds:↵
[JOINT_STATE]+↵
[FINGER JOINT]+↵
```

The joint definition contains a break down for one or more joints of the angle range to a discrete number of states. These states and the ranges are equivalent to those defined in JOINT_STATE. The number of states for the joint are given after "joint_positions:". The joints that this definition applies to follow (as "FINGER JOINT").

FINGER:

```
{thmb, indx, midl, ring, pnky, wrst}
```

One of 5 fingers, or the wrist.

JOINT:

```
{
   {rot, mphl, ip, abdt},
   {mphl, pxip, dsip, abdt},
   {prot, ptch, yaw}
}
```

For the thumb, the joints are rotation, metacarpal, inter-phelangeal, and abduction. For the wrist, the joints are palm rotation, pitch, and yaw. For all other fingers, the joints are metacarpal, proximal inter-phelangeal, destal inter-phelangeal, and abduction. The joints are diagrammed in "SVE_gloveData" on page 212.

GESTURE_DEFINITION:

```
name: <stringID>↵
[priority: <n>↵]
[
   FINGER JOINT [<f>]+↵
]+
[
   transitions:↵
   [
      FINGER JOINT TRANSITION↵
   ]+
   end:↵
   [
      FINGER JOINT [<f>]+↵
   ]+
]+
```

The gesture definition consists of a name, an optional priority (lower numbers have priority), and then a beginning pose definition. An optional transition and end pose definition may follow. The pose definitions consist of a joint specification (FINGER JOINT), and a list of probability values (as many as the joint states defined for that joint). The probability values are used to recognize a pose. If a joint is not specified, the default is that all states of the joint have the probability of 1.0. This means that the joint can be in any state, and it will not preclude that gesture from being recognized.

TRANSITION:

```
{none, any, forward, backward, back&forth}
```

Defines a transition that a joint can go through. The default is "any".

APPENDIX C: Reference Manual

1. SVE Function Reference

1.1. Main SVE loop

- **boolean**
  SVE_init(char *programName, SVE_config config, int *argc, char *argv[]);

char *programName	String used to label the graphics window.
SVE_config config	Configuration of the SVE system. Defines the devices to be used and rendering style.
int *argc	Arguement count (parameter given to main()).
char *argv[]	Arguement list (parameter given to main()).

This function must be called before any other SVE function call, except those that tell the library where to find the configuration file or other types of files. It sets up the SVE system state. Note that any values set in the `.sve.init` file will override the flags in the `config` variable.

- **void**
  SVE_beginEventLoop(void);

This function begins the SVE system event loop. Execution will not return to this point until the application is exited or `SVE_stopEventLoop()` or `SVE_done()` is called. The application will only receives control during this loop via callback routines defined prior to executing `SVE_beginEventLoop()`.

- **void**
  SVE_stopEventLoop(void);

This function causes the SVE system to exit from its event loop after the current frame is rendered. The current state, however, is not distroyed. Therefore, the system can be resumed where it left off if the `SVE_beginEventLoop()` funcion is used.

- **void**
  SVE_done(void);

This function shuts down the SVE system, including any tracking and glove devices being used. If the screen is being sent through the scan converter for video output, the computer screen is returned to normal operation. If the system needs to be restarted, another call to `SVE_init()` is required.

- **void**
  SVE_abort(void);

This functoin shuts down the SVE system, cleans up, then exits with an `exit(0)`.

1.2. SVE Configuration Routines

- **void**
  SVE_setInitFilename(char *initFilename);

Sets the filename of the initialization file that is read when `SVE_init()` is called. By default, this is "`.sve.init`".

1.3. World/object utilities

1.3.1 Load/Save

- **boolean**
  SVE_loadWorld(char *filename);

```
char *filename              Filename of the SVE world file.
```

This function loads in an SVE world from the specified filename. This world is then rendered during the rendering phase. If the given file does not exist in the path given, the Default Object Directory is used. The function returned `FALSE` if a world file could not successfully be read.

- **list**
  SVE_loadObjects(char *filename, SVE_object parent, SVE_state state);

```
char *filename              Filename of an SVE world file.
SVE_object parent           Parent object of all objects to be
                            read from the given file.
SVE_state state             Current state.
```

Reads in the list of objects contained in the given world file (using the Default Object Directory if the file is not found in the given path) and attaches them as children of the given `parent` object. If `SVE_WORLD` is given as the `parent` object, the objects read are children of the root (not attached to anything). Returns an unordered linked list with elments of `SVE_object`. If the world file sets an initial origin or flight speed, then those values will be used to set the appropriate fields of the given `state` structure. The `config` field of the `state` structure is used to determine if the textures encountered need to read from disk.

- **list**
  SVE_addObjects(char *filename, SVE_state state);

```
char *filename              Filename of an SVE world file.
SVE_state state             Current state.
```

Reads in the list of objects contained in the given world file and places them in the world as unattached objects. The world used is the object tree of the `state` given. If the given world file contains a user

position or user speed, then the appropriate fields of the `state` structure will be set using those values. Returns a linked list of `SVE_object`'s added (which may be the middle of a longer list if there where already objects in the world).

SVE_object
- **SVE_loadObject(char *filename, char *name, boolean *posInitialized);**

`char *filename`	`SVE object file name.`
`char *name`	`Name to be associated with the object.`
`boolean *posInitialized`	`This is set to TRUE if object's position has been initialized.`

This function allocates memory for an SVE object, reads the object definition from the given file (using the Default Object Directory if the file is not found in the given path), and returns the `SVE_object`. Note that the object is *not* placed in the world object tree, and therefore will not be rendered until it *is* added to the world object tree. If the initialization file initialized the object's position, then `posInitialized` is set to `TRUE`, so that the application can decide whether to override that initialization or not.

---

boolean
- **SVE_saveWorld(char *filename);**

`char *filename`	`SVE world file name.`

The current SVE world object hierarchy is saved in an SVE world file format in the file given. Objects which have changed will be saved to a SVE object file. Returns `FALSE if` the function was unable to save every file it needed to save.

---

boolean
- **SVE_saveObjects(list objectList, char *filename);**

`list objectList`	`Linked list of SVE objects to save.`
`char *filename`	`File name of the file to save to.`

Saves the list of objects in an SVE world file format to the filename given. If SVE_WORLD is given as the object list `(objectList)`, the world state object tree will be saved. Objects which have changed will be saved to a SVE object file. Returns `FALSE` if the function was unable to save every file it needed to save.

---

boolean
- **SVE_saveObject(SVE_object o, char *filename);**

`SVE_object o`	`SVE object to save.`
`char *filename`	`File name of the file to save to.`

Saves the object given in an SVE object file format to the specified file. Returns `FALSE` if the function was unable to save the object to the given file.

1.3.2 Information

SVE_object
- **SVE_findObject(char *name, list objectTree);**

`char *name`	`Name of desired object.`
`list objectTree`	`Object tree to search.`

Searches each object in the linked list of objects given in `objectTree` and their children, in a depth-first search. The object with the same name as the given `name` is returned, if found. `NULL` is returned if no object of that name exists in the object tree. If `SVE_WORLD` is given as the `objectTree`, the world state object tree is searched, which is equivalent to the `SVE_findWorldObject()` function below.

SVE_object
- **SVE_findWorldObject(char *name);**

`char *name`	`Name of desired object.`

Searches the SVE world object tree using a depth-first search. If an object with a name that matches the given `name`, it is returned. If no object in the SVE world object tree has that name, `NULL` is returned.

---

list
- **SVE_findAllObjects(char *name, list objectTree, boolean includingChildren);**

Generates a list of objects which were created with the given `name` (which can include the colon-number suffix of an object that was created with that name, but had its name changed to prevent duplicate names) which are in the given `objectTree`. If `includingChildren` is `TRUE`, then the given object tree is searched depth first to its leaf nodes.

SVE_object
- **SVE_findObjectInRepository(char *name);**

Find the object with the given `name` even if it has not been added to the global world tree.

---

void
- **SVE_getWorldMatrix(SVE_object o, M_matrix m);**

`SVE_object o`	`The SVE object in question.`
`M_matrix m`	`The returned result.`

Returns (in `m`) the transformation matrix of the given object, `o`, in absolute world coordinates (as opposed to its position matrix, which is only in relation to its parent).

---

boolean
- **SVE_objectVisible(SVE_object o);**

Returns `TRUE` if the object is visible, which means its `visible` attribute is `TRUE`, and the `visible` attribute of its parent and ancestors are also `TRUE`.

---

void
- **SVE_printObjectList(list objectList, boolean printChildren);**

list objectList	SVE object tree.
boolean printChildren	Flag used to shorten/lengthen output.

Prints to `stderr` the names of the objects in the `objectList` SVE objectlist. Their children (and their children's children, etc.) are printed also if the `printChildren` flag is `TRUE`. If `SVE_WORLD` is given for `objectList`, the world state object tree is used.

---

float
- **SVE_getNearestPoint(SVE_point testPoint, SVE_pointPtr *result, SVE_object *pointObject, list objectList);**

SVE_point testPoint	Point in world coordinates to test.
SVE_pointPtr *result	Reference to pointer to found vertex.
SVE_object *pointObject	Object containing the point found.
list objectList	List of objects to search for point.

Searches the objects of the given list (using the world state object tree if `SVE_WORLD` is given for `objectList`) for the closest vertex to the `testPoint`. The pointer to the closest point is returned in `*result`, and the SVE_object that contains that point is returned in `*pointObject`. The function returns the distance from `testPoint` to the point found. If no point is found, the function returns `BIG_DISTANCE`.

1.3.3 Object Creation and Deletion

SVE_object
- **SVE_createEmptyObject(char *name);**

char *name	Name to be given to the object.

Allocates memory for an object with no geometry associated with it, basically an empty object. The attributes of the object are set to the following values: (See the Data Structures section for a description of the `SVE_object` structure.)

```
name = name
parent = NULL
geometry = NULL
geometryList = Empty list
visible = TRUE
selectable = FALSE
highlight = FALSE
highlightGeometry = NULL
hasVisibleSphere = FALSE
visibleSphere = 0
hascolor = FALSE
materialIndex = 0
children = Empty list
boundaries = NULL
cullable = CULLABLE_BOX
update = CALC_ALL
facingViewerUpright = FALSE
facingViewer = FALSE
```

```
frameCallback = Empty list
remoteObject = FALSE
widgetData = NULL
visibility = SVE_VISIBILITY_UNKNOWN
moreTextures = NULL
UserPtr = NULL
position = identity matrix
worldPosition = identity matrix
```

SVE_object
- **SVE_createObjectCopy(SVE_object source, char *newName, boolean copyChildren);**

Creates a copy of the given object, `source`, with the name given by `newName`. If `copyChildren` is `TRUE`, the entire object tree, of which the `source` object is the root, will be copied.

---

void
- **SVE_deleteObject(SVE_object object, boolean attachChildrenToParent);**

Deletes the given `object`. If `attachChildrenToParent` is `TRUE`, the object's child objects will be attached to the given object's parent (or the root of the tree, if the object is a root object). If `attachChildrenToParent` is `FALSE`, then the child objects are deleted, also, as well as their children, and so on.

1.3.4 Object Manipulation

These functions are the preferred method of changing the attributes of the object, as certain flags need to be set to insure that other affected attributes are updated when needed. For example, when an object moves, the flags that indicate that the object's world position matrix and boundaries need to be recomputed need to be set.

---

void
- **SVE_changeObjectName(SVE_object object, char *newName);**

Changes the given `object`'s name to the given name `newName`.

---

void
- **SVE_setNewObjectPosition(SVE_object o, M_matrix newPosition);**

`SVE_object o`	`The SVE object to move.`
`M_matrix newPosition`	`The new position matrix.`

Changes the object's `position` matrix (its position and orientation in relation to its parent) to the given matrix.

- **void**
  SVE_moveObject(SVE_object o, SVE_point newOrigin);

`SVE_object o`	`The SVE object to move.`
`M_matrix newOrigin`	`The new origin of the object.`

Changes the "position" part of the object's `position` matrix (the left three values of the bottom row of the 4x4 matrix) to the given origin. This will move the object to a new location using the x, y, z axis of its parent.

- **void**
  SVE_translateObjectGlobal(SVE_object o,float x,float y,float z);

`SVE_object o`	`The SVE object to move.`
`float x`	`Amount to move in the x direction.`
`float y`	`Amount to move in the y direction.`
`float z`	`Amount to move in the z direction.`

Moves the object by the specified amounts in the x, y, and z directions using the x, y and z axis of its parent object.

- **void**
  SVE_rotateObject(SVE_object o, float angle, char axis);

`SVE_object o`	`The SVE object to move.`
`float angle`	`Amount (in degrees) to rotate the object.`
`char axis`	`Axis to rotate around('x', 'y', or 'z').`

Rotates the object around the specified axis (clockwise) using the object's (NOT the parent's) x, y, and z axis.

- **void**
  SVE_scaleObject(SVE_object o, float xScale, float yScale, float zScale);

Scales the given object by `xScale`, `yScale`, and `zScale` in the X, Y, and Z directions of the object's coordinate system, respectively.

- **void**
  SVE_translateWRT(SVE_object obj, SVE_object coordObj, SVE_point newPos, boolean absolute);

Translates the given object to a new position (`newPos`) in the local coordinate system of the `coordObj` object if absolute is `TRUE`, or translates by the vector given in `newPos` in the local coordinate system of `coordObj` if absolute is `FALSE`.

- **void**
  SVE_rotateWRT (SVE_object obj, SVE_object coordObj, float theDegrees, char axis);

Rotates the given object by the number of degrees in `theDegrees` in the local coordinate system of the `coordObj` object. The `axis` character defines which axis to rotate around, 'x', 'y', or 'z'.

- **void**
  SVE_scaleWRT (SVE_object obj, SVE_object coordObj, float scaleVal, char axis);

Scales the given object by the scale factor `scaleVal` in the local coordinate system of the `coordObj` object. The `axis` character defines along which axis the scale takes effect, 'x', 'y', 'z', or 'a' (uniform scale along all axis).

- **void**
  SVE_moveTo(SVE_object obj, SVE_object coordObj, SVE_point newPos);

Moves the given object to the position given in `newPos` in the local coordinate system of the `coordObj` object.

- **void**
  SVE_moveBy(SVE_object obj, SVE_object coordObj, SVE_point moveVector);

Moves the given object by the vector given in `moveVector` in the local coordinate system of the `coordObj` object.

- **void**
  SVE_moveVertex(SVE_object o, SVE_pointPtr oldVertex, SVE_point newPoint);

`SVE_object o`	The SVE object containing the vertex to move.
`SVE_pointPtr oldVertex`	Pointer to the vertex to be moved.
`SVE_point newPoint`	Point in world coordinates of the new position of the vertex.

Moves a vertex of an SVE object to a new position in world coordinates. Any face that uses that vertex will be affected.

- **void**
  SVE_setVisibility(SVE_object o, boolean visible, boolean setAncestors);

Sets the given object's `visible` attribute to the value given in `visible`. If `setAncestors` is `TRUE`, then the `visible` attribute of the object's parent and ancestors is also set the that value.

---

void
- **SVE_setSelectable(SVE_object o, boolean selectable);**

Sets the `selectable` attribute of the given object to the value given in `selectable`. The `selectable` attribute is generally used to keep the number of objects that can be selected by an interaction technique or collided with other objects to a minimum.

---

void
- **SVE_selectObject(SVE_object object);**

Generate an `SVE_OBJECT_SELECTION` event for the given object. If defined, the appropriate event callbacks will be called.

---

void
- **SVE_highlightObject(SVE_object object, boolean highlight);**

Generate an `SVE_OBJECT_HIGHLIGHT` event for the given object. If defined, the appropriate event callbacks will be called. If `highlight` is `TRUE`, and if a highlight geometry or highlight material is defined for the object, then that will be used when rending the object.

---

void
- **SVE_turnOnObjectLights(SVE_object o);**

Turn the light definitions contained in the given object to the "on" position. If the `SVE_LIGHTING` configuration is set, and the object is rendered, then the lights (if there are any in the object) will affect the rendering of the scene.

---

void
- **SVE_turnOffObjectLights(SVE_object o);**

Turn the light definitions contained in the given object to the "off" position. This will ensure that the lights, if any, will not affected the rendering of the scene.

---

boolean
- **SVE_changeText(SVE_object o, char *newText);**

`SVE_object o`	The object containing the text.
`char *newText`	The new text to assign to the object.

Searches the object for a `TEXT` primitive. If one is not found, this function returns `FALSE`. If a `TEXT` primitive is found, its `text` string is changed to a copy of the given `newText` and the function returns `TRUE`. The old `text` string is freed from memory. Only the first `TEXT` primitive in the object's primitives list is affected.

---

void
- **SVE_reCalculateWorldMatrix(SVE_object o);**

  `SVE_object o` Object which has changed.

This function should be called only when the `position` matrix of the object `o` has been changed without using another SVE function. Generally, the function `SVE_reCalculateParentBoundaries()` should be called as well. After this function call is made, the next call to `SVE_getWorldMatrix()` will result in the world position matrix of the object to be recalculated from the position matrixes of its parent and ancestors.

1.3.5 Object Geometry

---

void
- **SVE_setObjectMaterial(SVE_object o, SVE_material material);**

Sets the object's material of the given object to `material`. The object's material, if set, will override any material specified in the geometry definition.

---

void
- **SVE_setObjectHighlightMaterial(SVE_object o, SVE_material material, boolean includeText);**

Sets the given object's highlight material to the given `material`. The object will be rendered entirely with this material if the object is "highlighted" (its `highlight` attribute is set to `TRUE`). If `includeText` is set to `FALSE`, then any text primitives of the material will not change color (allowing them to be read if the highlight material is not the same color as the text color).

---

void
- **SVE_addObjectGeometry(SVE_object o, SVE_geometry newGeometry, float minDistance, float maxDistance);**

Addes a new (perhaps the first) geometry to the object's definition. The given geometry may have a valid range, set in `minDistance` and `maxDistance`, which are distances in meters. If `minDistance` or `maxDistance` are -1, then there isn't a limit for the minimum or maximum distance at which the geometry is valid. If both are -1, then the goemetry is always valid. The first valid geometry is the one that is rendered for the object. The list is searched in the order with which the geometries were added.

---

void
- **SVE_removeObjectGeometry(SVE_object o, SVE_geometry geometry);**

The given geometry is removed from the list of geometries of the given object.

---

void
- **SVE_changeObjectGeometry(SVE_object o, SVE_geometry newGeometry);**

The given geometry replaces the current geometry of the given object.

---

void
- **SVE_addPrimitiveToObject(SVE_object o, SVE_primitive primitive);**

The given primitive is added to the current valid geometry of the object. The function `SVE_addPrimitiveToGeometry()` should be used if there is some doubt as to which geometry is valid.

---

void
- **SVE_addHighlightPrimitiveToObject(SVE_object o, SVE_primitive primitive);**

Addes the given primitive to the highlight geometry of the given object. The highlight geometry will be rendered instead of the object's regular, valid, geometry if the object's `highlight` attribute is `TRUE`. Note that if the highlight geometry exists, it will override the highlight material. If the highlight geometry does not yet exist, this function will create it if the given primitive is valid.

---

void
- **SVE_updateObjectPrimitives(SVE_object o);**

`SVE_object o`	`Object which has changed.`

This function should be called only when any graphical aspect of the object `o` has been changed, besides its position or material, without using another SVE function. This function should not be necessary if the `SVE_initGeometryChange()` function is called before any change to the object's geometry.

---

void
- **SVE_updateAllObjects(void);**

The effect of this function is that `SVE_updateObjectPrimitives()` is called on all SVE objects that have been created. Calling this function may be necessary if the configuration has been changed (although it is done automatically by the `SVE_changeConfig()` function).

1.3.6 Geometry Routines

SVE_geometry
- **SVE_createGeometry(char *filename);**

Creates a `SVE_geometry` structure, which is identified by the given name. If another geometry has been created with the same name, and it has not been altered, then the returned `SVE_geometry` will reference the same geometry. Note that this function does *not* attempt to read a file by the given name. The name does not necessarily need to be a filename, although geometries loaded from SVE primitives files will use the full path name of the file from which the geometry description came to identify the geometry structure.

---

boolean
- **SVE_emptyGeometry(SVE_geometry geometry);**

Indicates if the given geometry is empty (has no description for visible appearance).

- **void**
  SVE_freeGeometry(SVE_geometry geometry);

Removes the given reference to the geometry. If this was the only reference to the geometry, then all memory used to store the geometry is freed.

- **void**
  SVE_initGeometryChange(SVE_geometry *geometry, SVE_object object);

Indicates to the SVE system that a geometry has or will be changed. This hint will allow the SVE renderer to correctly render the geometry, as well as treat the geometry's boundaries correctly. If the geometry is part of an object, then the `object` should be given, so that its boundaries are recalculated. If the geometry is independent of the object, then the `object` parameter can be `NULL`.

- **SVE_geometry**
  SVE_findGeometryInRepository(char *filename);

Returns a reference to the geometry referred to by the given name (which does not, necessarily, represent a file name), even if the geometry has not been assigned to an SVE object.

- **void**
  SVE_addPrimitiveToGeometry(SVE_geometry *geometry, SVE_primitive primitive, SVE_object object);

Adds the given `primitive` to the geometry referred to by the `geometry` parameter. If the geometry is associated with an object (one of the geometries on its geometry list), then the object should be given, also, so that its boundaries are recalculated. If the geometry is independent of an object, then the `object` parameter can be `NULL`.

- **void**
  SVE_beginPrimitive(SVE_primitiveType type);

Begins the construction of a primitive of the given `type`. The `type` parameter should be one of `POLYHEDRON`, `TEXTURED_POLYHEDRON`, `LINE`, `TEXT`, or `LIGHT`. A call to `SVE_beginPrimitive()` should be acompanied by a call to `SVE_endPrimitive()` to complete the construction of the primitive

---

- **int**
  SVE_primitivePoint(int index, float x, float y, float z);

Stores a location (x, y, z) using the given `index` to be used later by a call to `SVE_primitiveVertexIndexes()`. Returns the index if successful, or -1 if an error occured. If the given `index` is -1, then an un-taken index will be generated and used, and returned by the function.

- **int**
  SVE_primitivePointWithColor(int index, float x, float y, float z, float r, float g, float b, float a);

Stores a location (x, y, z) and color at that location (r, g, b, a for red, grean, blue, and alpha) using the given `index` to be used later by a call to `SVE_primitiveVertexIndexes()`. Returns the index if successful, or -1 if an error occured. If the given `index` is -1, then an un-taken index will be generated and used, and returned by the function. The color of the point will only be used for gouraud shaded scenes (i.e. the `SVE_GOURAUD` configuration flag is set).

- **int**
  SVE_primitiveNormal(int index, float x, float y, float z);

Stores the normal vector (x, y, z) using the given `index`. The last normal set by this function is used to set the normal of a face that is created or vertex that is created by `SVE_primitiveVertex()`. The normal can also be used at a vertex location by using the normal's `index` as a parameter to `SVE_primtiveVertexIndexes()`. Vertex normals are used by the Gouraud shading algorithm to define a surface which include all of the faces that have the vertex. The function returns the `index` if successful, or -1 if an error occured. If the given `index` is -1, then an un-taken normal index will be generated and used, and returned by the function.

- **int**
  SVE_primitiveTexCoord(int index, float s, float t);

Stores the texture coordinate (s, t) using the given `index`. The last texture coordinate set by this function is used to set the texture coordinate of a vertex that is created by `SVE_primtiveVertex()`. The texture coordinate can also be used at a vertex location by using the texture coordinate's index as a parameter to `SVE_primtiveVertexIndexes()`. Vertex texture coordinates are used by the texturing algorithm to place a texture on a face. Texturing will only occur if the `SVE_TEXTURES` configuration flag is set. The function returns the `index` if successful, or -1 if an error occured. If the given `index` is -1, then an un-taken texture coordinate index will be generated and used, and returned by the function.

- **int**
  SVE_beginPrimitiveFace(void);

Begins the definition of a face. Returns a unique index that identifies the face.

- **int**
  SVE_beginPrimitiveLine(void);

Begins the definition of a line. Returns a unique index that identifies the line.

- **int**
  SVE_beginPrimitiveClosedLine(void);

Begins the definition of a closed line (where an additional line is drawn between the first and last vertexes given). Returns a unique index that identifies the line.

SVE_vertexPtr
- **SVE_primitiveVertexIndexes(int pointIndex, int normalIndex, int textureIndex);**

Adds a vertex to a face or line being defined. The `pointIndex`, `normalIndex`, and `textureIndex` values given are the index values used in the primitive's point, normal, and texture coordinate lists, respectively. The `normalIndex` and `textureIndex` values can be -1, if no normal or texture coordiate is to be associated with this vertex. (If no normal is given, the face normal will be used by default.)

SVE_vertexPtr
- **SVE_primitiveVertex(float x, float y, float z);**

Adds a vertex to a face or line being defined with the given (x, y, z) location, and no normal or texture coordinate.

---

void
- **SVE_primitiveMaterial(SVE_material material);**

Specifies the material of a primitive, face, or line being defined.

---

void
- **SVE_primitiveHighlightMaterial(SVE_material material);**

Specifies the highlight material of a primitive being defined.

SVE_facePtr
- **SVE_endPrimitiveFace(void);**

Completes the definition of a face (begun by a `SVE_beginPrimitiveFace()` call). Returns the face definition.

SVE_facePtr
- **SVE_endPrimitiveLine(void);**

Completes the definition of a line (begun by a `SVE_beginPrimitiveLine()` or `SVE_beginPrimitiveClosedLine()` call). Returns the line definition.

---

void
- **SVE_primitiveText(char *text);**

Sets the character text (which can contain many lines separated by a '\n' character) of a `TEXT` primitive.

---

void
- **SVE_primitiveTextScale(float scale);**

Sets the scale of a `TEXT` primitive.

---

void
- **SVE_primitiveTextPosition(float x, float y, float z);**

Sets the position of the `TEXT` primitive. This is the position of the bottom left corner of the first letter of the text.

---

void
- **SVE_primitiveTextRotation(float xrot, float yrot, float zrot);**

Sets the rotation of the `TEXT` primitive. Rotation is applied in the following order: `xrot`, `yrot`, then `zrot`.

---

void
- **SVE_primitiveLineWidth(int lineWidth);**

Sets the line width for line primitives.

SVE_primitive
- **SVE_endPrimitive(void);**

Completes the construction of a primitive, which was begun by a call to `SVE_beginPrimitive`, and returns the constructed primitive.

---

void
- **SVE_getTextExtent(char *text, float *originX, float *originY, float *height, float *width);**

Given a text string, returns the origin (x, y), `height`, and `width` of the text which would be rendered if the text matrix was an identity matrix (i.e. the text geometry was unaltered). This routine effectively reports the boundaries of the text, except that the origin of the text is at the bottom left corner of the first character in the text. If the text has more than one line, the text will fall below the origin.

SVE_primitive
- **SVE_createEmptyPrimitive(int tag);**

Creates an empty primitive with the given id `tag`. For primitive construction, though, the `SVE_beginPrimitive()` is prefered.

SVE_primitive
- **SVE_getPrimitiveCopy(SVE_primitive primitive);**

Creates a copy of the given primitive.

1.3.7 Low Level Geometry Routines

SVE_pointPtr
- **SVE_createPoint(int index, float x, float y, float z);**

Create the internal represenation of a 3D location in space, which can be stored in the `pointList` of a primitive. The `index` should be a unique id (for the primitive), and can be used by the `pointIndex` record in a `SVE_vertexPtr` of the primitive.

SVE_pointPtr
- **SVE_createPointWithColor(int index, float x, float y, float z, float r, float g, float b, float a, float bw);**

Create the internal represenation of a 3D location in space and its color, which can be stored in the `pointList` of a primitive. The `index` should be a unique id (for the primitive), and can be used by the `pointIndex` record in a `SVE_vertexPtr` of the primitive. The color will be used when the SVE_GOURAUD configuration is specified.

SVE_normalPtr
- **SVE_createNormal(int index, float x, float y, float z);**

Create the internal represenation of a normal vector, which can be stored in the `normalList` of a primitive. The `index` should be a unique id (for the primitive), and can be used by the `normalIndex` record in a `SVE_vertexPtr` of the primitive.

SVE_textureCoordPtr
- **SVE_createTexCoord(int index, float s, float t);**

Create the internal represenation of a texture coordinate on a polygon, which can be stored in the `txtvertices` list of a primitive. The `index` should be a unique id (for the primitive), and can be used by the `texCoordIndex` record in a `SVE_vertexPtr` of the primitive.

SVE_vertexPtr
- **SVE_createVertex(int pointIndex, SVE_pointPtr point, int normalIndex, SVE_normalPtr normal, int texCoordIndex, SVE_textureCoordPtr texCoord);**

Create the internal representation of a polygon vertex. The `pointIndex`, `normalIndex`, and **texCoordIndex** values identify the appropriate point, normal, and texture coordinate in the primitive's lists. `normalIndex` and `texCoordIndex` can be -1 to indicate no normal or texture coordinate, respectively. if `point`, `normal`, and/or `texCoord` are `NULL`, and their respective indexes are not -1, then their values will be optained using the indexes and appropriate lists when needed.

SVE_facePtr
- **SVE_createFace(int index);**

Create the internal representation of a face (with no verticies), using the given index (which should be unique for the primitive).

SVE_facePtr
- **SVE_createLine(int index);**

Create the internal representation of a line (with no verticies), using the given index (which should be unique for the primitive).

SVE_lightPtr
- **SVE_createLight(void);**

Create the internal representation of a light. It must be added to the geometry of an object which is rendered to have any effect over the scene.

---

void
- **SVE_turnOnLight(SVE_lightPtr light);**

Given a light definition, turn it "on" (i.e. use it while rendering the scene if the light is contained in the geometry of an object in the scene).

---

void
- **SVE_turnOffLight(SVE_lightPtr light);**

Given a light definition, turn it "off" (i.e. don't use it while rendering a scene).

1.3.8 Material Routines

SVE_material
- **SVE_getColorMaterial(float color[3]);**

Given the red, green, blue color value, returns a material of that color, creating it if it does not already exist.

SVE_material
- **SVE_getMaterialByName(char *name);**

Given the `name` of a material, returns the material created with that name, or `NULL` if no material has been created with the name.

SVE_material
- **SVE_getMaterialByIndex(int index);**

Given the unique `index` of a material, returns the material created with that index, or NULL if no material has been created with that index.

SVE_material
- **SVE_createMaterial(char *name);**

Creates a material with the given name. If a material of the name already exists, then that material will be returned by the function. (Only one material for a particular name can exist.)

---

void
- **SVE_setMaterialAmbient(SVE_material material, float ambient[3]);**

Sets the ambient color of a material (color given no lights, or in shadow).

---

void
- **SVE_setMaterialDiffuse(SVE_material material, float diffuse[3]);**

Sets the diffuse color of a material (color response to light).

---

void
- **SVE_setMaterialSpecular(SVE_material material, float specular[3]);**

Sets the specular color of a material (color response to light at the viewer angle where the light provides specular reflection).

---

void
- **SVE_setMaterialEmission(SVE_material material, float emission[3]);**

Sets the emission color of a material.

---

void
- **SVE_setMaterialShininess(SVE_material material, float shininess);**

Sets the shininess value of a material.

---

void
- **SVE_setMaterialAlpha(SVE_material material, float alpha);**

Sets the alpha component of a material. By default, this indicates a level of transparency of a polygon with the given material, where 0.0 is fully transparent, and 1.0 is fully opaque.

---

void
- **SVE_setMaterialTexture(SVE_material material, char *texture, int textureMode, long textureEnv);**

Sets the file name of the texture image to map on polygons with the given material. The `textureMode` parameter can be `SVE_TEXTURE_REPEAT`, `SVE_TEXTURE_CLAMP`, or `SVE_TEXTURE_GREYSCALE`. The

textureEnv (texture environment) parameter can be SVE_TEXTURE_DEFAULT, SVE_TEXTURE_INTENSITY, SVE_TEXTURE_INTENSITY_ALPHA, SVE_TEXTURE_RGB, SVE_TEXTURE_RGB_LIGHTING, SVE_TEXTURE_RGB_ALPHA, or SVE_TEXTURE_RGB_ALPHA_LIGHTING.

- **void**
  SVE_setMaterial3Color(SVE_material material, float color[3]);

Sets the ambient and diffuse colors of the material to the given color.

- **void**
  SVE_setMaterial4Color(SVE_material material, float color[4]);

Sets the ambient and diffuse colors of the material to the first three values of the given color, and the alpha value of the material to the last value of the given color.

1.3.9 Texture swapping

- **int**
  SVE_defineObjectTextures(SVE_object o, int textureMode, int n, ...);

Defines a list of texture files to be used to texture map the given object. The n parameter indicates the number of texture files to be given. The textureMode parameter indicates the mode of the given texture files (SVE_TEXTURE_REPEAT, SVE_TEXTURE_CLAMP, or SVE_TEXTURE_GREYSCALE). The texture swap callback indicates which of the given textures is used for a particular frame.

- **int**
  SVE_defineObjectTextureList(SVE_object o, int textureMode, int n, char* textureNames[]);

Defines a list of texture files, in the textureNames array, to be used for all textures of the given object. The n parameter indicates the number of texture files to be given. The textureMode parameter indicates the mode of the given texture files (SVE_TEXTURE_REPEAT, SVE_TEXTURE_CLAMP, or SVE_TEXTURE_GREYSCALE). The texture swap callback indicates which of the given textures is used for a particular frame.

- **void**
  SVE_setTextureSwapCallback(SVE_object o, SVE_objectFunctionPtr f);

Sets the texture swap callback for a given object. The texture swap callback should return the index of the texture to use for the object in the list of textures given (with 0 being the index of the first texture).

- **long**
  SVE_defaultTextureSwap(SVE_object o, SVE_state state);

This is the default texture swap callback, which changes textures at the rate of 60 times per second, cycling through the list of textures.

1.3.10 Automatic object animation

```
void
• SVE_initAnimation()
```

Initializes SVE's animation routines.

```
void
• SVE_setObjectAnimation(SVE_object obj, boolean val)
```

Sets the animation flag for *obj* to be equal to *val*. A value of TRUE turns animation on for the object, FALSE turns it off.

```
void
• SVE_setAnimationVar(SVE_object obj, int var, double val)
```

Sets the animation variable(s) specified by *var*, for *obj*, to the value specified by *val*. The valid variables are X_VELOCITY, Y_VELOCITY, Z_VELOCITY, X_ACCELERATION, Y_ACCELERATION, Z_ACCELERATION, X_ROTATION, Y_ROTATION, and Z_ROTATION. See "Animation Variable" on page 42 for a description of the animation variables.

```
void
• SVE_setAnimationFunc(SVE_object obj,
       SVE_animateObjectFunctionPtr func)
```

Sets the object's animation function for *obj* to point to *func*.

```
void
• SVE_setDefaultAnimationFunc(SVE_animateObjectFunctionPtr func)
```

Sets the default animation function to be *func*.

```
void
• SVE_setUserDefinedData(SVE_object obj, void *data)
```

Sets the user defined data pointer for *obj* to point to *data*.

1.3.11 Animation Callbacks

```
void
• SVE_addAnimationCallback(SVE_functionPtr function);
```

Adds an animation callback function to the list of animation callbacks. The new function will be called before all animation callbacks added before it.

- **`void`**
  `SVE_removeAnimationCallback(SVE_functionPtr function);`

Removes the given animation callback, `function`, from the animation callback list.

- **`void`**
  `SVE_removeAllAnimationCallbacks(void);`

Removes all animation callbacks from the system.

- **`list`**
  `SVE_getAnimationCallbacks(void);`

Returns the linked list of animation callbacks (linked list of `SVE_functionPtr`).

1.3.12 Object Tree Manipulation

- **`void`**
  `SVE_clearWorld(void);`

Initializes the world back to how is set up when `SVE_init()` is called. All objects except the default object tree which represents the user are deleted. The default object tree is restored to its original tree structure.

- **`void`**
  `SVE_attachToObject(SVE_object child, SVE_object parent);`

`SVE_object child`	`The soon-to-be child object.`
`SVE_object parent`	`The soon-to-be parent object.`

Removes the SVE object `child` from the tree, saving its position and orientation in world coordinates, and then attaches it to the SVE object `parent` as one of it's children, calculating its `position` matrix so that it does not "move" in world. Although the object does not move in the world as a result of this function, afterwards it is attached to the `parent` object, so that if the `parent` object moves, the `child` object will follow as if rigidly connected to it. This is the preferred function to use for changing links in the object tree.

- **`SVE_object`**
  `SVE_removeObject(char *name, list *objectTree);`

`char *name`	`Name of object to be removed.`
`list *objectTree`	`Reference to an object tree (list).`

Searches the object tree given for an object with the name, `name` (depth first search). When it is found, it is removed (including its children) from the given object tree, and a reference to it is returned. If SVE_WORLD is given for `objectTree` (i.e. `SVE_removeObject("myobject", SVE_WORLD)`, the world state object tree will be searched. Note that the object is *not* deleted.

---

void
- **SVE_removeObjectEntry(SVE_object object, boolean attachChildrenToParent);**

`SVE_object object`	`Object to be removed.`
`boolean attachChildrenToParent`	

Removes the object (including its children) from the object tree it currently resides in. Note that the object is *not* delected. If `attachChildrenToParent` is `TRUE`, then the object's children are attached to the object's parent. Otherwise, the object children remain attached to the object.

---

void
- **SVE_addToWorldTree(SVE_object o);**

Adds the given object to the root of the current world object tree.

---

void
- **SVE_addChildToObject(SVE_object child, SVE_object parent);**

`SVE_object child`	`The soon-to-be child object.`
`SVE_object parent`	`The soon-to-be parent object.`

Adds the object `child` to the front of the linked list of children objects of the object `parent`.

---

void
- **SVE_addToObjectList(SVE_object o, list *objectTree);**

`SVE_object o`	`An object to add to an object tree.`
`list *objectTree`	`Reference to an object tree (list).`

Adds the given object `o` to the list of objects referenced by `objectTree`. (Added to the front of the linked list of objects referenced by `objectTree`.) If `SVE_WORLD` is given for `objectTree`, the object will be added to the world state object tree.

SVE_object
- **SVE_removeFirstObject(list *objectList);**

Removes the first object of the list of objects given as `objectList`, and returns it. If the list contains no objects, this function returns `NULL`.

SVE_object
- **SVE_getFirstObject(list *objectList);**

Returns the first object of the objects listed in `objectList` (or of the world object tree if `objectList` is `SVE_WORLD`). The object is *not* removed from the list.

1.3.13 Object Boundaries

- **SVE_boundaries *
  SVE_getObjectBoundaries(SVE_object o);**

SVE_object o	SVE object.

Returns the boundaries of the object o, which include its children (and their children, etc.). If the object has moved or changed, the boundaries will be recalculated.

- **SVE_boundaries *
  SVE_getPrimitiveBoundaries(SVE_object o);**

SVE_object o	SVE object.

Returns the primitive boundaries of the object o, which just bound its primitive components (faces, text, etc.). If the object's primitives have changed, the boundaries will be recalculated.

- **void
  SVE_getObjectCenter(SVE_object obj, SVE_point center);**

SVE_object obj	SVE object.
SVE_point center	Point to contain center point of object.

The center of the object `obj`'s boundaries (including its children) in world coordinates is returned in `center`.

- **SVE_object
  SVE_objectMatrixHit(list objectlist, M_matrix m, float margin, boolean onlySelectable);**

list objectlist	Object list to search.
M_matrix m	Matrix describing the position tested.
float margin	Margin of error.
boolean onlySelectable	Indicate if only "selectable" objects are to be considered.

Tests each object in the object tree `objectlist`, traversing the tree depth first, to see if the point in world coordinates indicated by the matrix `m` is contained in the object's boundary, within the given `margin` of error (in meters). If the `onlySelectable` flag is true, only objects which are "selectable" (`object->selectable == TRUE`) are considered. The first object found is returned. If no objects are found, then `NULL` is returned. If `SVE_WORLD` is given for `objectlist`, the world state object tree is searched.

SVE_object
- **SVE_objectPointHit(list objectlist, SVE_point pt, float margin, boolean onlySelectable);**

list objectlist	Object list to search.
SVE_point pt	Point in world coordinates to be tested.
float margin	Margin of error.
boolean onlySelectable	Indicate if only "selectable" objects are to be considered.

Tests each object in the object tree `objectlist`, traversing the tree depth first, to see if the point in world coordinates indicated by `pt` is contained in the object's boundary, within the given `margin` of error (in meters). If the `onlySelectable` flag is true, only objects which are "selectable" (`object->selectable == TRUE`) are considered. The first object found is returned. If and object is not found, then `NULL` is returned. If `SVE_WORLD` is given for `list`, the world state object tree is searched.

---

boolean
- **SVE_matrixHitObject(SVE_object obj, M_matrix m, float margin);**

SVE_object obj	Object to be tested.
M_matrix m	Matrix describing the position tested.
float margin	Margin of error.

If the point in world coordinates described by the matrix `m` is in the object `obj`'s boundaries within the given `margin` of error, this function returns `TRUE`, otherwise it returns `FALSE`.

---

boolean
- **SVE_pointHitObject(SVE_object obj, SVE_point pt, float margin);**

SVE_object obj	Object to be tested.
SVE_point pt	Point describing the position tested.
float margin	Margin of error.

If the position in world coordinates described by the point `pt` is in the object `obj`'s boundaries within the given `margin` of error, this function returns `TRUE`, otherwise it returns `FALSE`.

---

boolean
- **SVE_matrixHitObjectPrimitives(SVE_object obj, M_matrix m, float margin);**

SVE_object obj	Object to be tested.
M_matrix m	Matrix describing the position tested.
float margin	Margin of error.

If the point in world coordinates described by the matrix `m` is in the object `obj`'s primitive boundaries (which do not include its children) within the given `margin` of error, this function returns `TRUE`, otherwise it returns `FALSE`.

- **boolean**
  SVE_pointHitObjectPrimitives(SVE_object obj, SVE_point pt, float margin);

SVE_object obj	Object to be tested.
SVE_point pt	Point describing the position tested.
float margin	Margin of error.

If the position in world coordinates described by the point `pt` is in the object `obj`'s primitive boundaries (which do not include its children) within the given `margin` of error, this function returns `TRUE`, otherwise it returns `FALSE`.

- **boolean**
  SVE_objectBoundsCollide(SVE_object o1, SVE_object o2, boolean includeChildren);

Returns `TRUE` if the boundaries of the given objects intersect. If `includeChildren` is `TRUE`, than the boundaries of each object includes the boundaries of all their children. Note that sphere boundaries are approximated by box boundaries for this function.

- **boolean**
  SVE_objectBoundsInBounds(SVE_object o1, SVE_object o2, boolean includeChildren);

Returns `TRUE` if the boundaries of object `o1` lie completely in the boundaries of object `o2`. If `includeChildren` is `TRUE`, then the boundaries of each object includes the boundaries of all of their children. Note that sphere boundaries are approximated by box boundaries for this function.

- **SVE_object**
  SVE_pickObject(SVE_object pointer, list objectList, boolean pickChildren, boolean canPickLines, float *intersectDist, SVE_objectFunctionPtr rejectFunc);

This function returns the closest SVE object to the origin along the negative Z axis of the given `pointer` object (in the `pointer` object's coordinate system) of the objects given in `objectList`. If `pickChildren` is `TRUE`, the child objects of each object is checked recursively. If `canPickLines` is `TRUE`, then geometries containing lines (including text) are picked if the negative Z axis of the pointer intersects the geometry's bounding volume. The distance (in `pointer` coordinates) to the picked object (if any) is returned in `interesectDist`. If no object is intersected, the function returns `NULL`. A `rejectFunc` function can be provieded to reject objects that may be picked (intersects the negative Z axis, but may not be the closest). This function takes the object pointer and the world state, and should return `SVE_REJECT` or `SVE_OK`.

- **SVE_boundaries ***
  SVE_createEmptyBoundaries(void);

Creates a boundary structure, but does not set any boundary values. This is done automatically when ever a SVE function creates an object or a boundary.

SVE_boundaries *
- **SVE_createSphereBoundaries(float radius, SVE_point origin);**

`float radius`	`Radius of sphere boundary.`
`SVE_point origin`	`Origin (in object coords) of sphere.`

Creates a boundary structure which describes a sphere boundary at the `origin` given (in object coordinates) with the `radius` given.

---

void
- **SVE_setSphereBoundaries(SVE_boundaries **bounds, float radius, SVE_point origin);**

Sets the given boundaries to be a sphere with the given `radius` around the given `origin`.

SVE_boundaries *
- **SVE_createBoxBoundaries(SVE_point x1, SVE_point x2);**

`SVE_point x1`	`Bottom left point of box boundary.`
`SVE_point x2`	`Top right point of box boundary.`

Creates a box boundary described by two opposite points, the bottom left (`x1`) and the top right (`x2`). The points are in object coordinates.

---

void
- **SVE_setBoxBoundaries(SVE_boundaries **bounds, SVE_point x1, SVE_point x2);**

Sets the given boundaries to be a box with two opposite corners `x1` and `x2`.

SVE_boundaries *
- **SVE_copyBoundaries(SVE_boundaries *source, SVE_boundaries *dest);**

`SVE_boundaries *source`	`Boundary to be copied.`
`SVE_boundaries *dest`	`Boundary to copy to.`

Copies the boundary structure given in `source` to the structure of `dest`. Returns the boundaries that have been copied. If dest is `NULL`, a new boundary structure is allocated and returned.

---

void
- **SVE_calculateBounds(list objectTree);**

`list objectTree`	`SVE object tree to calculate bounds.`

This function can be used if the default behavior of calculating boundaries only when needed is not desired. It forces the object and primitive boundaries of the objects in `objectTree` and their children to recalculate their boundaries at the time the function is called.

---

boolean
- **SVE_getPrimitiveExtents(SVE_primitive primitive, SVE_point minBounds, SVE_point maxBounds);**

Given a `primitive` definition, calculates the corners of a box which would contain all components of the primitive.

---

void
- **SVE_reCalculateObjectBoundaries(SVE_object o);**

`SVE_object o` Object which has changed.

When an object's geometry or children list is changed OUTSIDE of an SVE function, this function should be called to insure that the object's boundaries are recalculates when needed.

---

void
- **SVE_reCalculateParentBoundaries(SVE_object o);**

`SVE_object o` Object which has moved.

When an object's position matrix is changed OUTSIDE of an SVE function, this function should be called to insure that the object `o`'s parent (and its parent, etc.) recalculates its boundaries when needed. This function is usually called at the same time as `SVE_reCalculateWorldMatrix()`.

---

void
- **SVE_drawBoundaries(list objects);**

`list objects` Object list to display boundaries.

This function will render the object and primitive boundaries of each object in the object list and their children. This function is usually called in a frame callback routine, as the viewing matrix must be on the top of the matrix stack for the boundaries to be rendered correctly.

1.3.14 Widgets

---

void
- **SVE_registerWidgetType(char *type, SVE_createWidgetFunctionPtr createFunc, SVE_readWidgetFunctionPtr fileCreateFunc, SVE_widgetFunctionPtr deleteFunc, SVE_widgetFunctionPtr eventFunc, list eventList);**

Registers a widget type using the given name. Instances of this widget type are created using the `SVE_makeWidget()` function. When a widget instance is created, the given `createFunc` funciton is called (if it is not `NULL`). If the widget is given in a file, then the `fileCreateFunc` function is called. If the widget instance is deleted, then the `deleteFunc` function is called before the widget is removed from the scene and its storage deleted. When an event in the given `eventList` occurs, the given `eventFunc` function is called.

- **`SVE_object`**
  `SVE_makeWidget(char *type, char *name, void *data);`

Creates an instance of a widget of the given `type`, using the given `name` and `data` pointer.

- **`void`**
  `*SVE_getWidgetData(char *type, char *name);`

Given a widget type name, and a name of a widget instance, returns the widget data of that widget instance.

1.4. Callback utilities

1.4.1 Event callbacks

- **`void`**
  `SVE_ResetCallbacks(SVE_config config);`

`SVE_config config`	`The current configuration of the SVE system.`

Resets all user-defined event callbacks to `NULL`, and restores the default event callbacks to their original value, and enables them. (See "Responding to Events" on page 36.)

- **`void`**
  `SVE_registerCallback(int event, SVE_functionPtr function);`

`int event`	`SVE or user defined event value.`
`SVE_functionPtr function`	`user function to be called.`

Registers a user defined function (`function`), which will be called every time an event (`event`) reaches the front of the event queue. The function will be added to th front of a list of event callback routines for the event specified. If the function returns `EVENT_CONSUMED`, then the callback routines further down the list (added before this one) will not be called. If the function returns `EVENT_IGNORED`, then the next callback function on the list will be called. See `include/event.h` for list of event types.

- **`void`**
  `SVE_removeCallback(int event, SVE_functionPtr function);`

`int event`	`SVE or user defined event.`
`SVE_functionPtr function`	`Event callback function to be removed.`

Removes the given callback function from the list of callback functions for the given event..

- **`void`**
  `SVE_removeAllCallbacks(int event);`

Removes all of the event callbacks for the given event type.

```
list
• SVE_getEventCallback(int event);
  int event                    SVE or user defined event.
```

Returns the linked list of the callback functions for the given event

---

```
void
• SVE_getMouseState(long int *mouseButton, long int *xPos, long int
      *yPos, long int *xWinPos, long int *yWinPos);
```

This routine can be used to find the current position of the mouse, and the buttons that are currently pressed. The **mouseButton** parameter is a bit flag, containing the current state of the mouse buttons and modifier keys (control, alt, etc.), 1 for down, 0 for up. See `include/event.h` for the appropriate bit masks. The `xPos` and `yPos` parameters are set to the position of the mouse on the screen. The `xWinPos` and `yWinPos` parameters are set to the position of the mouse in the SVE window.

```
void
• SVE_enterEvent(SVE_eventType eventType, void *eventData);
```

Enters an event of type `eventType` onto the event queue using the appropriate `eventData` information about the event. The event will be processed after all events that already have been entered in the event queue by the application or SVE system. If an event callback for the event exists, it will be called when the the event is processed.

1.4.2 Frame callback

```
void
• SVE_setFrameCallback(SVE_functionPtr function);
  SVE_functionPtr function   User defined function.
```

Adds the given function to the list of functions which will be called by the SVE system just after it clears the back buffer and loads the current viewing matrix, and just before the SVE system renders all of the objects in its object tree (given in the `SVE_state` structure). The functions are called in the opposite order from which they were added (last one added is called first).

```
void
• SVE_removeFrameCallback(SVE_functionPtr function);
```

Removes the given frame callback from the list of frame callbacks.

```
void
• SVE_removeAllFrameCallbacks(void);
```

Removes all frame callbacks from the list of frame callbacks.

```
    list
•   SVE_getFrameCallback(void);
```

Gets the linked list of frame callback functions

1.4.3 Object Frame Callbacks

```
    void
•   SVE_setObjectFrameCallback(SVE_object object,
          SVE_objectFunctionPtr function);
```

Adds the given `function` to the given `object`'s frame callback list. The function will be called after all other frame callbacks previously added to the object's list. These functions are called just before the object's geometry is rendered, after the graphics library has been set up to render in the object's local coordinate system.

```
    void
•   SVE_removeObjectFrameCallback(SVE_object object,
          SVE_objectFunctionPtr function);
```

Removes the given `function` from the list of frame callbacks of the given `object`.

```
    void
•   SVE_removeAllObjectFrameCallbacks(SVE_object object);
```

Removes all of the frame callback functions of the given given `object`.

```
    list
•   SVE_getObjectFrameCallback(SVE_object object);
```

Gets the list of frame callback functions for the given `object` (linked list of `SVE_objectFunctionPtr`)

1.4.4 Frame End Callbacks

```
    void
•   SVE_setFrameEndCallback(SVE_functionPtr function);
```

Adds the given `function` to the list of frame callback functions that will be called immediately after all objects in the scene have been rendered by the SVE system (before the frame is displayed to the screen). The given function will be called before all other frame end callbacks given before it.

```
    void
•   SVE_removeFrameEndCallback(SVE_functionPtr function);
```

Removes the given `function` from the list of frame end callback functions.

```
void
```
- **`SVE_removeAllFrameEndCallbacks(void);`**

Removes all frame end callback functions.

```
list
```
- **`SVE_getFrameEndCallbacks(void);`**

Returns the list of frame end callback functions (linked list of `SVE_functionPtr`).

1.4.5 Object Culling Callbacks

```
void
```
- **`SVE_addCullingFunction(SVE_cullFunctionPtr function);`**

This function allow the application to add to the list of functions called for each object to determine if the object should be "culled", or not drawn, given the current viewing parameters. The default list of culling functions contains one function, which is `SVE_getObjectVisibility()`. If the application would like to provide its own culling function which performs the default behavior and some additional, application specific, work, then its culling function should call `SVE_objectCull()` to determine if the object's bounds are in the viewing frustrum. The functions on the list are called one at a time until one returns SVE_NONE_VISIBLE or SVE_CHILDREN_ONLY_VISIBLE. If no function returns one of these values, the object's visibility is set to the value returned by the last culling function. The functions are called in the reverse order from which they were added (last added is first to be called).

```
void
```
- **`SVE_removeCullingFunction(SVE_cullFunctionPtr function);`**

Removes the given function from the list of culling functions.

```
void
```
- **`SVE_removeAllCullingFunctions(void);`**

Removes all culling functions, include the default function which is added automatically by the SVE system when it is initialized.

```
list
```
- **`SVE_getCullingFunctions(void);`**

Returns the list of culling functions that have been added.

```
SVE_vistype
```
- **`SVE_getObjectVisibility(SVE_object obj, M_matrix worldToEye);`**

Determines the visibility of the given object to the given the `worldToEye` matrix (which is the perspective matrix premultiplied by the viewing matrix). The function returns one of the following predefined values:

SVE_ALL_VISIBLE (if the object and its children are complete visible. The SVE renderer will assume the object's children are also completely visible), SVE_PART_VISIBLE (if the only part of the object is visible. The SVE renderer will also check each of the object's children for visibility), SVE_CHILDREN_ONLY_VISIBLE (if the object's geometry is not visible, but its children may be visible), SVE_NONE_VISIBLE (if neither the object nor the object's children are visible. The SVE renderer will not render the object or its children), and SVE_VISIBILITY_UNKNOWN (the SVE renderer will render the object, and will check its children for visibility).

---

unsigned short
- **SVE_objectCull(SVE_object o, M_matrix worldToEye, boolean earlyExit);**

This function determines if the given object is in the viewing frustrum determined by the worldToEye transformation matrix (which is the perspective matrix premultiplied by the viewing matrix, and is what is provided when the culling functions are called). The field earlyExit should be TRUE if it is not important to know whether the object is fully or partially visible. This function returns one of the following values: CULL_NOT_VISIBLE, CULL_ALL_VISIBLE, CULL_DONT_KNOW. If the function returns a value other than these, then the object is partially visible.

1.5. User Oriented Utilities

SVE_status
- **SVE_fly(SVE_state state);**

SVE_state state	SVE world state.

This function can be used as an animation callback, or called from another function, to have the user (specifically the userObject, usually named "USER") move in the direction of the view (or the viewingObject). Each call moves the user in the direction of the view in an amount proportional to the flightSpeed value in the state structure.

---

void
- **SVE_flyWithDirection(SVE_state state, M_matrix direction);**

SVE_state state	SVE world state.
M_matrix direction	Direction to fly in.

This function causes the user (specifically the userObject, usually named "USER") to move in a given direction. If the position matrix of an object is given as the direction, the user will fly in the negative Z direction in the object's coordinate system. Each call moves the user in an amount proportional to the flightSpeed value in the state structure.

---

void
- **SVE_flyInObject(SVE_state state, SVE_object o);**

SVE_state state	SVE world state.
SVE_object o	Object within which the user can fly.

This function causes the user (specifically the userObject, usually named "USER") to move in the direction of the view (or the viewingObject). Each call moves the user in the direction of the view in an amount proportional to the flightSpeed value in the state structure, but does not allow the user to move

outside of the object `o`'s boundaries. An error value is used to insure that the user cannot move outside the object.

- **void**
  SVE_flyInObjectWithDirection(SVE_state state, SVE_object o, M_matrix direction);

`SVE_state state`	`SVE world state.`
`SVE_object o`	`Object within which the user can fly.`
`M_matrix direction`	`Direction to fly in.`

This function causes the user (specifically the `userObject`, usually named "`USER`") to move in the direction given. If the `position` matrix of an object is given as the `direction`, the user will fly in the negative Z direction in the object's coordinate system. Each call moves the user in an amount proportional to the `flightSpeed` value in the `state` structure., but does not allow the user to move outside of the object `o`'s boundaries. An error value is used to insure that the user cannot move outside the object.

1.6. Rendering Functions

The rendering of the object tree in the world state structure is done automatically each frame. The frame callback allows the application to render anything not contained in the tree. Standard GL or OpenGL calls can be used to render GL or OpenGL primitives not supported by the SVE primitive structure, or SVE objects not in the world object tree can be rendered using the following routines:

- **void**
  SVE_renderObjectList(list objectTree);

`list objectTree`	`Object tree to render.`

Renders each object and their children (and their children, etc.) in the object list `objectTree`. If `objectTree` is `SVE_WORLD`, the world state object tree will be rendered (which is already done automatically for each frame).

- **void**
  SVE_renderObject(SVE_object obj);

`SVE_object obj`

Renders one object given by `obj`, and its children (and its children, etc.).

- **void**
  SVE_renderWorld(SVE_state worldState);

`SVE_state worldState`	`SVE world state with an object tree which is rendered.`

This is equivalent to `SVE_renderObjectList(worldState->objectTree);`

---

void
- **SVE_renderNow(SVE_state worldState, SVE_functionPtr frameCallback);**

Renders a complete frame of the object tree stored in the given `worldState`. The given `frameCallack`, if not `NULL`, will be called before the object tree is rendered. Note that this frame will be independent of any other frame rendered by the SVE system or other calls to `SVE_renderNow()` or `SVE_renderNowWithFrameCallbacks()`.

---

void
- **SVE_renderNowWithFrameCallbacks(SVE_state worldState, boolean newFrame);**

Renders a complete frame of the object tree stored in the given `worldState`. Note that, if `newFrame` is TRUE, this frame will be independent of any other frame rendered by the SVE system or other calls to `SVE_renderNow()` or `SVE_renderNowWithFrameCallbacks()`. All frame callbacks are called with this routine. Therefore if you call `SVE_renderNowWithFrameCallbacks()` in a frame callback (or a function that could be called from a frame callback) you must take some action that will prevent the function from calling `SVE_renderNowWithFrameCallbacks()` again! For example, this will work:

```
void frameCallback(SVE_state state)
{
   static boolean inFrameCallback = FALSE;

   if (!inFrameCallback) {
      inFrameCallback = TRUE;
      SVE_renderNowWithFrameCallbacks(state);
      inFrameCallback = FALSE;
   }
}
```

---

void
- **SVE_getViewingAndPerspectiveMatrix(SVE_state worldState, M_matrix viewing, M_matrix perspective);**

Returns the viewing and perspective matrixes used to set up the view given objects in world coordinates, and the user's world eye point and view plane position and window extents.

---

void
- **SVE_drawTrackerExtent(float radius);**

`float radius`	Anticipated tracker range.

This function will render a sphere represented by 3 line circles (around each axis at the) surrounding the tracker transmitter. It might be useful to provide feedback as to whether the user is approaching the limits of the tracker. It usually is called in a frame callback routine, as the viewing matrix must be on the top of the matrix stack for the sphere to be rendered correctly.

---

void
- **SVE_enableBackfaceCulling(void);**

Enable back face culling, which indicates that faces whose normals do not face the viewer should not be rendered. This is the default.

---

void
- **SVE_disableBackfaceCulling(void);**

Disables the rendering optimization that removes faces whose normals do not face the viewer from the faces that are rendered. Back face culling is enabled by default.

1.7. General utilities

1.7.1 State functions

SVE_config
- **SVE_getConfig(void);**

Returns the current configuration.

---

void
- **SVE_changeConfig(SVE_config newConfig);**

Changes the current configuration to the given `newConfig`.

SVE_state
- **SVE_getWorldState(void);**

Returns the current world state.

SVE_state
- **SVE_setWorldState(SVE_state newState);**

Changes the current world state to the given `newState`. The fields of the new state must be set correctly to avoid many potential problems. Of critical importance is the references to the user objects, such as the eye and view plane, as well as the various resource lists.

---

void
- **SVE_setBackgroundColor(float r, float g, float b);**

`float r, g, b`	Red, green, and blue values. Ranging from 0 to 1.

Sets the color to be used as the background for each frame rendered by the SVE system.

- **void**
  SVE_setFlightSpeed(float speed);

  `float speed` New flight speed.

Sets the flight speed value (in meters per second) used for the flying functions.

- **void**
  SVE_setViewPlaneExtents(float minX, float minY, float maxX, float maxY);

Sets the extents of the window on the view plane which defines the viewing volume of the viewer, using the viewer's eye point and planes that pass through the eye point and each edge of the window. Given values are ofsets from the view plane's origin in its X-Y plane.

- **void**
  SVE_setFieldOfView(int fieldOfViewY, float aspectRatio);

Sets the field of view (in 0.1's of degrees, i.e. 100 = 10 degrees) and aspect ratio of the view. From these values, the extents of the window on the view plane are calculated.

- **void**
  SVE_setDefaultObjectDirectory(char *directory);

Sets the directory path list that is searched when object files are to be loaded. The directory path list is given as paths separated by a colon (':'). The previous object directory path list is lost.

- **void**
  SVE_setDefaultWorldDirectory(char *directory);

Sets the directory path list that is searched when world files are to be loaded. The directory path list is given as paths separated by a colon (':'). The previous world directory path list is lost.

- **void**
  SVE_setDefaultMaterialDirectory(char *directory);

Sets the directory path list that is searched when material files are to be loaded. The directory path list is given as paths separated by a colon (':'). The previous material directory path list is lost.

- **void**
  SVE_setDefaultTextureDirectory(char *directory);

Sets the directory path list that is searched when texture files are to be loaded. The directory path list is given as paths separated by a colon (':'). The previous texture directory path list is lost.

1.7.2 Matrix functions

- **void**
  SVE_invertMatrix(M_matrix mat, M_matrix inv);

`M_matrix mat`	`Operand.`
`M_matrix inv`	`Result.`

Inverts the given matrix `mat` and returns the result in the matrix `inv`.

- **void**
  SVE_getRelativeMatrix(M_matrix a, M_matrix b, M_matrix result);

`M_matrix a`	`Source position matrix.`
`M_matrix b`	`Destination position matrix.`
`M_matrix result`	`Transformation from "a" to "b".`

Calculates the relative transformation from a position defined in the matrix `a` to a position defined in the matrix `b`, and stores the result in `result`. Thus [result][a] = [b]. For example, the following code sequence calculates an object's coordinate transformation from one world coordinate position (`parentWorldPos`) given its own world coordinate position (`childWorldPos`), which is used when attaching the child object to the parent object without changing the child object's world coordinates position. (An object's world coordinate position is calculated by pre-multiplying its coodinate transform from its parent by the parent's world coordinate position.)

```
SVE_getWorldMatrix(parent, parentWorldPos);
SVE_getWorldMatrix(child, childWorldPos);
SVE_getRelativeMatrix(parentWorldPos, childWorldPos, newPos);
SVE_copyMatrix(newPos, child->position);
```

- **double**
  SVE_getMatrixDist(M_matrix a, M_matrix b);

`M_matrix a`	`Operand.`
`M_matrix b`	`Operand.`

Calculates the euclid distance of the translational components of the matrix `a` and the matrix `b`. Returns the result.

- **void**
  SVE_copyMatrix(M_matrix s, M_matrix d);

`M_matrix s`	`Source matrix.`
`M_matrix d`	`Destination matrix.`

Copies the 4X4 matrix `s` to the matrix `d`.

- **void**
  SVE_printMatrix(M_matrix m);

This function will print the matrix on *stdout*. This function can be used for debugging.

1.8. Default User Tree Information

1.8.1 Cursor Information

- **void**
  SVE_getCursorPosition(M_matrix pos);

  `M_matrix pos` Result.

Stores in `pos` the position and orientation matrix of the `cursorObject`, which is usually the "`SVE cursor`" object.

- **SVE_object**
  SVE_getCursorObject(void);

Returns the `cursor`Object object, which usually is the "`SVE cursor`" object. The "`SVE cursor`" object begins as an empty object, but can be given primitives and/or children objects which will be rendered at the location and with the orientation of the "`SVE cursor`" object.

1.8.2 HMD Information

- **void**
  SVE_getHMDPosition(M_matrix pos);

  `M_matrix pos` Result.

Stores in `pos` the world position and orientation matrix of the `hmdObject`, which is usually the "`SVE HMD`" object.

- **SVE_object**
  SVE_getHMDObject(void);

Returns the `hmdObject` object, which usually is the "`SVE HMD`" object. The "`SVE HMD`" object begins as an empty object, but can be given primitives and/or children objects which will be rendered at the location and with the orientation of the "`SVE HMD`" object.

1.9. Polling Device Routines

- **SVE_pollDevice**
  SVE_createPollingDevice(int type, int deviceId,
  SVE_pollFunctionPtr openFunction, SVE_pollFunctionPtr pollFunction, SVE_pollFunctionPtr closeFunction, char *attachTo, void *data);

Creates a polling device instance of the given `type` and `deviceId`. After the returned device is added to the system, using `SVE_addPollingDevice()`, the device is openned with the given `openFunction`, polled each frame with the given `pollFunction` (which will be given the returned `SVE_pollDevice` structure), and closed with the given `closeFunction`. If given, the `attachTo` parameter will be used to set the `attachTo` field of the returned `SVE_pollDevice`, which the object with the `attachTo` name. The given `data` pointer will also be stored in the returned `SVE_pollDevice` structure.

SVE_pollDevice
- **SVE_findPollingDevice(int type, int id);**

Returns a polling device given the `type` and `id` it was defined with.

---

void
- **SVE_addPollingDevice(SVE_pollDevice device);**

Adds the given polling device to the SVE system, which will open it, call the polling function once each frame, and close it with the sysetem is done.

1.9.1 Tracker Device Routines

---

boolean
- **SVE_initTracker(int trackerId, char *machine, int type, char *port, int receiver, char *attachTo, float hemiVector[3], SVE_pollFunctionPtr pollFunction);**

Creates a new tracking device interface, using the given unique `trackerID`. The interface uses a tracker on the given `machine`, of the given `type`, `port`, and `receiver`. The `attachTo` string idenfies the name of an object that should follow the tracker. The `hemiVector` defines a vector pointed from the origin of the tracking reference frame to the center of the hemisphere within which the tracker is to be accurate. The `pollFunction` is called each frame to obtain the tracking information. The `SVE_updateTracker()` function can be used for the pollFunction.

---

void
- **SVE_attachTracker(int trackerId, char *objectName, SVE_pollFunctionPtr pollFunction);**

Attaches the tracking device interface identified with the given `trackerId` to the object with the name `objectName`. The polling function of the tracker interface, which is called to optain the current tracking information, is given in the `pollFunction` parameter.

SVE_status
- **SVE_updateTracker(SVE_pollDevice device, SVE_state state);**

Polls a tracking device for the current tracker information. Can be used as a polling function for the `SVE_initTracker()` and `SVE_attachTracker()` routines.

---

boolean
- **SVE_trackerExists(int trackerId);**

Returns `TRUE` if a tracker device interface has been created, through `SVE_initTracker()`, using the given `trackerId`.

1.9.2 Glove Utilities

- **boolean**
  SVE_initGlove(int gloveId, char *machine, char *port, char *attachTo);

`int gloveId`	`Unique id number of the glove.`
`char *machine`	`Machine to which the glove device is attached.`
`char *port`	`Port to which the glove device is attached ("/dev/ttyd2", for example).`
`char *attachTo`	`String name of an object to attach the glove objects that make up the hand representation.`

Initializes the glove input device connected to the given `machine` through the given `port`, and sets up an object structure for the hand's representation in the SVE environment. The hand object structure is retrieved from the "glove" sub-directory of the Default Object Directory, and is attached to the SVE object identified by the `attachTo` name.

- **void**
  SVE_readHandFile(char *filename, int gloveId);

`char *filename`	`Name of hand data file to read.`
`int gloveId`	`Reference to glove device.`

Reads a hand calibration file that contains the information needed to calibrate a particular hand to the sensors of the CyberGlove input device. The `gloveId` parameter refers to the same number that was used to initialize the glove device.

- **void**
  SVE_saveHandFile(char *filename, int gloveId);

`char *filename`	`Name of hand data file to save.`
`int gloveId`	`Reference to glove device.`

Saves a hand calibration file that contains the information needed to calibrate a particular hand to the sensors of the CyberGlove input device. The `gloveId` parameter refers to the number given when the glove device was initialized.

- **void**
  SVE_readGestureFile(char *filename, int gloveId);

`char *filename`	`Name of gesture file to read.`
`int gloveId`	`Reference to glove device.`

Reads a gesture file, and adds the gesture's defined there to the gesture list associated with the glove device identified by the `gloveId` parameter.

---

void
- **SVE_resetGestureList(int gloveId);**

int gloveId	Reference to glove device.

Removes all gestures memorized and read from files for the glove device identified by the gloveId parameter..

---

void
- **SVE_saveCurrentGesture(int gloveId, int priority, boolean replaceOldGesture);**

int gloveId	Reference to glove device.
int priority	Priority value.
boolean replaceOldGesture	Indicates if the gestures containing the same priority should be replaced.

Saves the angles of each joint made by the hand in the glove at the last time it was polled (once per frame) on a list of gestures maintained for the given glove device. The gesture list is keyed by the given priority. There can be many gestures per priority. If this gesture should be the only one at this priority, the replaceOldGesture flag should be TRUE. When recognizing gestures (see SVE_recognizeGestures()), the gesture with the lowest priority number that matches each angle of the current gesture will be considered a SVE_GESTURE event. The SVE_GESTURE event, with the priority of the matching gesture begin the event value, will be placed on the event queue.

---

void
- **SVE_recognizeGestures(int gloveId, boolean flag);**

int gloveId	Reference to glove device.
boolean flag	Enable flag.

If flag is TRUE, the current gesture made by the hand in the glove input device identified by gloveId will be checked with the list of gestures which have been saved for a match. If a match is found, an SVE_GESTURE event will be placed on the event queue (see SVE_saveCurrentGesture()) with an event value corresponding to the gesture index. If no match is found, an SVE_GESTURE event will occur with the value NULL_GESTURE. If flag is FALSE, hand gestures made by the device will not be recognized.

SVE_object
- **SVE_getPalmObject(SVE_object attachedTo);**

SVE_object attachedTo	Object to which the hand representation is attached.

Returns the object that is the palm object of the hand structure (see SVE_initGlove()).

1.10. Sound utilities

There are currently two different kinds of audio support. The first one will load in *AIFF* files and the second one will create spatial sound coming from a SUN workstation.

1.10.1 Audio commands

- **int**
  SVE_audioOpenSound(char *filename);

  `char *filename` audio file to load.

This function will load the audio sample into memory and return the a handle for this sample. If repeat is set to `TRUE`, then the sample begins playing immediately, otherwise it is just loaded.

- **void**
  SVE_audioReplaySound(int SampleNo, int repeat);

  `int SampleNo` handle of the sample.

  `int repeat` play the sound continuously?

This function will check whether the sample with handle `SampleNo` is not playing and play it if it isn't. To have the sound play continuously without user interaction make the `repeat` flag equal to `TRUE`.

- **void**
  SVE_audioStopSound(int SampleNo);

  `int SampleNo` handle of the sample.

If the sample with `SampleNo` is currently playing, it is stopped. The sample will not be deleted from memory.

- **void**
  SVE_audioCloseSound(int SampleNo);

  `int SampleNo` handle of the sample.

This function will stop playing the sample corresponding to `SampleNo` and remove it from memory.

- **boolean**
  SVE_audioCheckSound(int SampleNo);

  `int SampleNo` handle of the sample.

This functions checks to see whether or not the sample corresponding to SampleNo is currently playing. `TRUE` is returned if it is. Otherwise `FALSE` is returned.

- **void**
  SVE_audioSetVolume(int left, int right);

  `int left` volume left earphone.

  `int right` volume right earphone.

With this function the volume of both left and right earphone can be changed. The volume should be between 0 and 255 (the scale of change is logarithmic).

- **voids**
  SVE_audioGetVolume(int *left, int *right);

int *left	volume left earphone.
int *right	volume right earphone.

This function will return the volume of both the left and the right ear phone. The number will be between 0 and 255.

1.10.2 Spatial sound utilities

- **void**
  SVE_attachSoundToObject(SVE_object object);

SVE_object object	The new sound source object.

This call will activate a mechanism that automatically updates the spatial sound system running on the SUN station (nagel.cc.gatech.edu). The sound location will be coupled to the transformation of the object. This function can only be used when the system was initialized with the constant *SVE_SPATIALSOUND*.

NOTE: Spacial sound is currently not available.

- **void**
  SVE_changeSoundUpdateRate(int rate);

int rate	The update rate (in frames).

This call acts on the mechanism that automatically updates the spatial sound system running on the SUN station (nagel.cc.gatech.edu). The update rate determines the number of frames between sending sound updates, e.g. 1 = each frame, 10 = each tenth frame (default value is 1).Because too fast update rates will cause digitial noise ("clicks"), this update rate should be lowered to about 10-20 frames per second. This function can only be used when the system was initialized with the constant *SVE_SPATIALSOUND*.

1.11. Text Output Routines

- **void**
  SVE_printVerbose(char *string, ...);

Outputs the given format `string`, and the value of any additional parameters as indicated by the format string. Use in the same way that `printf()` is used. The string "SVE:" will be prepended to the output.

- **void**
  SVE_printError(char *string, ...);

Outputs the given format `string`, and the value of any additional parameters as indicated by the format string, to `STDERR`. Use in the same way that `printf()` is used. The string "SVE ERROR:" will be prepended to the output.

---

void
- **SVE_printFileError(char *filename, char *string, ...);**

Outputs the given format `string`, and the value of any additional parameters as indicated by the format string. Use in the same way that `printf()` is used. An error message, indicating that this is a file error message and including the given `filename`, will be prepended to the output.

---

void
- **SVE_printDebug(char *string, ...);**

Outputs the given format `string`, and the value of any additional parameters as indicated by the format string. Use in the same way that `printf()` is used. The given output will not appear if debugging has been turned "off" (i.e. the `SVE_debug` global variable is `FALSE`) and `DEBUG` is not defined (when compiling).

2. Data Structure Utilities

The SVE library uses two data structures and their utilities to store various parts of its data. The linked list data structure is used to store an ordered group of items, the number of which is unbounded. The dynaminc array data structure is used to store a group of items, each using a index key, the number of which is unbounded. The search time for any item in the linked list is worst case N, where N is the number of items in the list. The worst case search time for any item in the dynamic array is worst case 1 array reference.

The SVE library uses another utility to parse files. This utility supports directory path lists, and commented lines.

2.1. Linked List

This is a simple linked list data structure, where any item of type `list` is a pointer to the first node in a linked list. The pointer from one node to the next is also of type `list`. The following routines are defined for this data structure.

---

list
- **createList();**

Creates an empty linked list, and returns it.

---

void
- **addToList(list *listHead, void *data);**

`list *listHead`	`Reference to the list front.`
`void *data`	`Data to store (cast to void *)`

Stores the given `data` at the front of the given list.

---

void
- **addToListEnd(list *listHead, void *data);**

`list *listHead`	`Reference to the list front.`
`void *data`	`Data to store (cast to void *)`

Searches for the end of the given list, and creates a new node containing the given `data` at the end.

---

void
- **addToListFront(list *listHead, void *data);**

`list *listHead`	`Reference to the list head`
`void *data`	`Data to store (cast to void *)`

Adds a new node containing the given `data` to the front of the given list.

- **void**
  addToListSorted(list *listHead, void *data, int (*compareFunc)(void *a, void *b));

```
list *listHead              Reference to the list head
void *data                  Data to store (cast to void *)
int (*compareFunc)(void *a, void *b)
                            Function that compares two data values
```

Adds a new node containing the given `data` to the list, placing it so that the nodes up to the inserted node are in order. The order is determined by the given compare function. The compare function is given two data values that have been stored, and it should return an integer less than zero if the first value is less than the second, zero if the values are equal, and greater than zero if the first value is greater than the second. As an example, if the data values being stored are pointers to character strings (char *), then the `strcmp` function could be used as the compare function.

- **void ***
  getData(list listHead);

```
list listHead               The head of the list
```

Returns the data value stored at the node which is at the head of the given list. If the data value is not a void*, it should be cast to the correct type.

- **list**
  getNext(list listHead);

```
list listHead               The head of the list
```

Returns the list which begins after the first node in the given list. If the list only has one node in it, this function will return an empty list.

- **list**
  findData(list listHead, void *data, int (*equalFunc)(void *data, void *b));

```
list listHead               Head of the linked list
void *data                  Data to search for
int (*equalFunc)(void *data, void *b)
                            Function used to search
```

This function searches the given function, looking for a data value in the list which is equal to the given data value, based on the equal function provided. The equal function should return TRUE (1) if the two data values given to it should be considered equal (the first parameter to the equal function is the data value given), or FALSE(0) if they should be considered not equal. (Note that this is different from the compare function given to `addToListSorted()`). If the given equal function is `NULL`, then the data values will be directly compared.

This funciton returns a list, where the first node in the list contains the data value which is "equal" to the given data value, or it returns an empty list if no matches where found.

- **void ***
  removeData(list *listHead, void *data, int (*equalFunc)(void *data, void *b));

```
list *listHead              Head of the linked list
void *data                  Data to search for
int (*equalFunc)(void *data, void *b)
                            Function used to search
```

This function is similar to the `findData()` function, except that if a match is found (based on the given equal function), then the node containing the match is removed from the given list, and the data value contained in that node is returned. This function returns `NULL` if no match is found.

- **void ***
  removeFirst(list *listHead);

```
list *listHead              Reference to the linked list head
```

Removes the first node in the given list and returns the data value stored in that node. If the given list is empty, then this function returns `NULL`.

- **boolean**
  listEmpty(list listHead);

```
list listHead               Head of the linked list
```

Returns `TRUE` if the list is empty, `FALSE` if it is not.

- **void**
  sortList(list *listHead, int (*compareFunc)(void *a, void *b));

```
list *listHead              Reference to the list head
int (*compareFunc)(void *a, void *b)
                            Function used to compare data values
```

This function sorts the given linked list using the given compare function to determine relative ordering between two data values. The compare function should return an integer less than zero if the first parameter is less than the second parameter, zero if the two parameters are equal, or an integer greater than zero if the first parameter is greater than the second parameter.

---

void
- **copyList(list *dest, list source, void (*copyFunc)(void **destData, void *sourceData));**

`list *dest`	`List to copy to`
`list source`	`List to copy`
`void (*copyFunc)(void **destData, void *sourceData)`	`Function to copy data values`

This function copies the `source` list, using the given copy function to copy the data values, and returns the copy in `dest`. The copy function should make a copy of the `sourceData` parameter, set the `destData` parameter to the copy, and return the copy. The `strcpy` function is an example which will copy string data values.

---

void
- **appendList(list *dest, list source);**

`list *dest`	`List to append to`
`list source`	`List to append`

Appends the list `source` to the list refered to by `dest`. This function does not make a copy.

---

void
- **updateSortedList(list *listHead, list insertList, int (*compareFunc)(void *a, void *b), void (*freeFunc)(void *data));**

`list *listHead`	`Reference to the list head`
`list insertList`	`List to insert`
`int (*compareFunc)(void *a, void *b)`	`Function used to compare data values`
`void (*freeFunc)(void *data)`	`Function used to free duplicate values`

This function inserts the list `insertList` into the list refered to by `listHead`, inserting each item so that the resulting list is in order (if the list refered to by `listHead` was in order to begin with). The compare function is used to compare data values in the list (see `sortList()`). The free function is used to free any data values that are duplicates (the compare function returns 0). If the free function is `NULL`, then duplicates are not freed.

---

void
- **freeList(list *listHead, void (*freeFunc)(void *data));**

`list *listHead`	`Reference to the list head`
`void (*freeFunc)(void *data)`	`Function which frees data values`

This function frees the given list (setting it to be empty afterwords). The data values in the list are freed using the given function. If the function is `NULL`, no function will be called to free the data values stored in the list.

---

void
- **printList(list listHead, void (*printFunc)(void *data));**

`list listHead`	`Reference to the list head`
`void (*printFunc)(void *data)`	`Function used to print the data values`

This function prints to `stdout` the list given, using the given print function to print out the data values at each node.

2.2. Dynamic Array

The dynamic array utilities maintain a dynamic array data type, DA_array. This structure can be considered an implementation of an array which is unbounded. The array grows as more items are inserted into it, but does so in a way that preserves (almost) direct access to the items in the array.

DA_array
- **DA_createArray(void);**

Creates an empty dynamic array.

---

int
- **DA_empty(DA_array d);**

`DA_array d`	`The dynamic array`

Returns `TRUE`(1) if the given array is empty, `FALSE`(0) if it is not.

---

int
- **DA_store(DA_array *sp, void *v_ptr, int index);**

`DA_array *sp`	`Reference to the dynamic array`
`void *v_ptr`	`Data item to store (cast to void *)`
`int index`	`Index used to store the data item`

This function stores the given data item into the given dynamic array at the location identified by the index value. Any data already at that location will be lost. The function returns `FALSE` if there is not enough memory to store the data.

---

void *
- **DA_get(DA_array sp, int index);**

`DA_array sp`	`The dynamic array`
`int index`	`Index of the desired item`

This function finds the data item stored in the given dynamic array at the location identified by the index value. If there is no data item stored there, the function returns `NULL`.

---

int
- **DA_arrayMin(DA_array p);**

`DA_array p` — The dynamic array

Returns the smallest index used to store a data item. Returns -1 if the array is empty.

---

int
- **DA_arrayMax(DA_array p);**

`DA_array p` — The dynamic array

Returns the largest index used to store a data item. Returns -1 if the array is empty.

---

void
- **DA_merge(DA_array *A, DA_array *B);**

`DA_array *A` — Reference to a dynamic array

`DA_array *B` — Reference to a dynamic array

This function merges the data values contained in the dynamic array `B` into the dynamic array `A`. Items in `A` will be overwritten by any value in `B` with the same index value. The dynamic array `B` is destroyed by this function (it is returned empty).

DA_array
- **DA_makeCopy(DA_array s, void (*copyFunc)(void **dest, void *source));**

`DA_array s` — The dynamic array to copy

`void (*copyFunc)(void **dest, void *source)` — Function used to copy data items

This function copies the given dynamic array, using the given copy function to copy the stored data items, and returns the copy. If the copy function is `NULL`, then the items are copied directly.

---

void
- **DA_free(DA_array *f, void (*freeFunc)(void *data));**

`DA_array *f` — The dynamic array to free

`void (*freeFunc)(void *data)` — Function used to free data items

This function frees the storage used by a dynamic array, freeing the data items stored in it using the given free function. If the free function is `NULL`, then the items are not freed. The dynamic array is set to be empty.

- **void**
  DA_print(DA_array d);
  `DA_array d`

Prints the items of the given dynamic array out to `stdout`.

2.3. File Parser Utility

- **void**
  FP_prependToPath(char **path, char *directories);

Add directories given in `directories` (separated by colon) to the front of the given `path`.

- **void**
  FP_appendToPath(char **path, char *directories);

Add directories given in `directories` (separated by colon) to the front of the given `path`.

- **int**
  FP_openFile(char *filename, char *defaultPath);

Opens the file with the given filename if found in the given search path. The current directory is searched last unless the search path contains ".". The return value should be used for other file parser functions on this file.

- **void**
  FP_closeFile(int No);

Close the file refered to by the given file id (`No`).

- **char ***
  FP_getFilename(int No);

Returns the filename of the file refered to by the given file id (`No`), including the path were it was found.

- **void**
  FP_reportFileError(int No, char *message, ...);

Prints the given `message` including the full pathname of the file refered to by the given file id (`No`). This function should be used as `fprintf()` is used.

- **void**
  FP_setCommentToken(int No, char *comment);

Sets the `comment` token (which defines the beginning of a comment, which ends at the end of the line) for the file refered to by the file id (`No`). Comments are ignored. The default comment is "#".

---

void
- **FP_setSeparateToken(int No, char *separate);**

Sets the set of characters which are considered "white space" for the file refered to by the given file id (`No`). If any of these characters are encountered, then the current token is ended, and the next token begins at the next character which is not in the `separate` character set. The default separate character set contains space, tab, and carrage return.

---

int
- **FP_nextLine(int No);**

This function will tell whether the next token in the file refered to by the file id given (`No`) is on a new line.

---

int
- **FP_getNextBoolean(int No);**

Returns the next token of the file refered to by the file id given (`No`) if it is of type boolean (`TRUE` or `FALSE`).

---

float
- **FP_getNextFloat(int No);**

Returns the next token of the file refered to by the file id given (`No`) if it is of type float.

---

int
- **FP_getNextInt(int No);**

Returns the next token of the file refered to by the file id given (`No`) if it is of type int.

---

char *
- **FP_getNextToken(int No);**

Returns the next token of the file refered to by the file id given (`No`) as a character string.

---

int
- **FP_getCurrentLine(int No);**

Returns the line number on which the last token of the file refered to by the file id given (No) was read.

---

char *
- **FP_getRemainingLine(int No);**

Returns all characters from the beginning of the next token (thus skipping the "white space" characters after the last token) until the end of the current line, not including the carrage return, of the file refered to by the file id given (`No`).

```
/char *
```
- **FP_findFile(char *filename, char *defaultPath);**

Returns the full filename with its path of the file by the given `filename` if it is found using the given `defaultPath` path list. The file is *not* openned.

2.4. Vector Routines

The following routines provide homogeneous coordinates and vectors which can be used to represent locations or directions in three dimensions. The vector's (or location's) fourth component is usually 1. However, when a vector is put through a non-linear transformation, the fourth component may be given a value other than 1. In this case, the 3D vector or coordinate is obtained by dividing the first three components by the forth component. Note that most of the vector routines given here do not alter the forth component of the vector(s) they use, nor do they check to make sure that the component is 1 before performing their function.

The following types are defined for the vector routines:

```
typedef float V_scalar;
typedef V_scalar V_vector[4];
```

In addition, the constants `X`, `Y`, `Z`, and `W` are defined to represent the first, second, third and forth components of the vector, respectively.

```
void
```
- **V_zero(V_vector);**

Initializes vector to (0, 0, 0, 1).

```
void
```
- **V_neg(V_vector);**

Negates the `X`, `Y`, and `Z` components of the given vector. Thus (x, y, z, w) becomes (-x, -y, -z, w).

```
void
```
- **V_norm(V_vector);**

Normalizes the given vector based on the first three components (`X`, `Y`, `Z`). The length of the (`X`, `Y`, `Z`) vector result will be 1.

```
V_scalar
```
- **V_len(V_vector);**

Returns the length of the given (`X`, `Y`, `Z`) vector.

```
V_scalar
```
- **V_sqrLen(V_vector);**

Returns the square of the length of the given (`X`, `Y`, `Z`) vector, which can be computed faster than `V_len()`.

---

void
- **V_copy(V_vector dest, V_vector source);**

Copies the four component from the given `source` vector to the `dest` vector.

---

void
- **V_mult(V_vector, V_scalar);**

Multiplies the (`X`, `Y`, `Z`) components of the given vector by the given scalar value.

---

void
- **V_add(V_vector accum, V_vector operand);**

Adds the (`X`, `Y`, `Z`) components of the given `operand` vector to the (`X`, `Y`, `Z`) components of the given `accum` vector, and stores the (`X`, `Y`, `Z`) result in the `accum` vector.

---

void
- **V_sub(V_vector accum, V_vector operand);**

Subtracts the (`X`, `Y`, `Z`) components of the given `operand` vector from the (`X`, `Y`, `Z`) components of the given `accum` vector, and stores the (`X`, `Y`, `Z`) result in the `accum` vector.

V_scalar
- **V_dot(V_vector, V_vector);**

Returns the dot product of (X, Y, Z) components of the given vectors.

---

void
- **V_cross(V_vector accum, V_vector operand);**

Performs the cross product of the (`X`, `Y`, `Z`) components of the given `accum` and `operand` vectors, and places the (`X`, `Y`, `Z`) vector result in the `accum` vector.

---

void
- **V_move(V_vector v, V_scalar x, V_scalar y, V_scalar z)**

Sets the (`X`, `Y`, `Z`) components of the vector `v` to (`x`, `y`, `z`).

---

void
- **V_move4(V_vector v, V_scalar x, V_scalar y, V_scalar z,V_scalar w)**

Sets the vector `v` to (`x`, `y`, `z`, `w`).

2.5. Matrix Routines

The matrix routines deal with 4x4 transformation matrixes, which, when multiplied by a 4 component (3D) vector, can transform a location (represented by the vector) to a new location in homogeneous coordinents. If the transformation is non-linear, then the forth component may be a value other than 1, in which case the actual transformed location can be obtained by dividing the first, second, and third components by the forth component.

The following type definition is used to represent the 4x4 transformation matrix:

```
typedef V_vector M_matrix[4];
```

It should be noted that matrix transformations in SVE pre-multiply the current accumilated transformation, and vectors are transformed by post-multiplying the accumilated transformation. Therefore, a translational matrix will contain the X, Y, and Z translations in the (3, 0), (3, 1), and (3, 2)locations of the matrix (which is 0 based). Thus, the translation for matrix `t` is contained in `t[3][0]`, `t[3][1]`, and `t[3][2]`.

---

void
- **M_loadID(M_matrix);**

Initializes the given matrix to the identity matrix.

---

int
- **M_invert(M_matrix);**

Inverts the given matrix and stores the result in it. The multiplication of the inverted and non-inverted forms of the same matrix results in an identity matrix.

---

void
- **M_copy(M_matrix dest, M_matrix source);**

Copies the `source` matrix values to the `dest` matrix.

---

void
- **M_mult(M_matrix accum, M_matrix operand);**

Multiplies the given `accum` matrix with the `operand` matrix, and stores the result in the `accum` matrix.

---

void
- **M_vectMatMult(V_vector, M_matrix);**

Multiplies the given vector by the given matrix, and stores the result in the given vector.

---

void
- **M_matVectMult(M_matrix, V_vector);**

Multiplies the given matrix by the given vector, and stores the result in the given vector.

3. SVE Global Variables.

This section provides descriptions of the different variables the user can observe or change in an SVE program. These variables all have default values which in most cases will not need to be changed.

3.1. Information

The following variables are provided to the application for information only. For the most part, changing the variables will have at best no effect, and at worst will have negative consequences. (Just don't do it.) Generally, a function exists in the SVE library that will allow the application to make the desired change.

---

int
- **SVE_minX,**
- **SVE_minY,**
- **SVE_sizeX,**
- **SVE_sizeY;**

The origin and size of the window on the screen where the scene is rendered.

---

int
- **SVE_vofY;**

float
- **SVE_aspectRatio;**

The field of view (in tenths of degrees) and aspect ratio used to define the view. See `SVE_setFieldOfView()`.

---

float
- **SVE_viewPlaneMinX,**
- **SVE_viewPlaneMinY,**
- **SVE_viewPlaneMaxX,**
- **SVE_viewPlaneMaxY;**

Defines the extents of the viewplane (on a XY plane whose position in relationship to the eyepoint is given in `SVE_viewPlanePosition`, and whose rotation from parallel to the XY plane of the coordinate system in which the viewplane is defined is given in `SVE_viewPlaneRotation`). See `SVE_setViewPlaneExtents()` and the initialization file description.

---

float
- **SVE_viewPlanePosition[3],**
- **SVE_viewPlaneRotation[3];**

Defines the position of the viewplane and the rotation of the view plane from parallel to the XY plane (given as a rotation about the X axis, then a rotation about the Y axis, then a rotation about the Z axis, in that order). See the initialization file description.

---

int
- **SVE_facesRendered;**

Reports the number of faces rendered for the last frame.

---

int
- **SVE_linesRendered;**

Reports the number of lines rendered for the last frame.

---

long int
- **SVE_texturesRead;**

Reports the total size (in bytes) of texture images that have been read.

SH_table
- **SVE_objectRepository;**

A hash table that contains all SVE objects (of type `SVE_object`) that have been created, even if they haven't been added to the rendered object tree. See `SVE_findObjectInRepository()`.

---

list
- **SVE_geometryRepository;**

A linked list of SVE geometries (of type `SVE_geometry`) that have been created, even if they haven't been used by an object as the object's appearance. See `SVE_findGeometryInRepository()`.

---

long int
- **SVE_availableHardware;**

Bit array that reports the hardware identified by SVE. The current hardware abilities that are identified are SGI Reality Engine graphics (`SVE_REALITY_ENGINE`) and texture mapping (`SVE_TEXTURE_MAPPING`).

---

long int
- **SVE_textureMemory;**

Reports the maximum amount of texture memory that the hardware claims it has (if it makes any claim). If the textures that are loaded exceed the amount of texture memory, than previous textures will be swapped out for new textures, which can cause dramatic rendering slow-downs.

---

char *
- **SVE_hostname;**

The name of the machine on which the SVE application is running, including the domain name.

```
    int
•   SVE_facesRead;
```

This variable contains the total number of faces loaded in the world by the file readers.

3.2. Setup

The following global variables define information that the application may wish to access and/or change to affect how the application functions. Most of these variables have a corresponding attribute in the initialization file that can be set to acheive the same result.

```
    SVE_state
•   SVE_worldState;
```

This is the current state structure that the SVE system uses to define the environment the user is in. This structure contains the rendered world object tree, environment attributes (such as background color), callback function stacks, the previous event, time information for syncronizing animation, and the identification of which objects perform special functions, among other things. See the `SVE_state` description.

```
    float
•   SVE_near,
•   SVE_far;
```

Defines the distance (in meters) to the near and far planes of the viewing volume. The distance is given in eye coordinates (the coordinate system of the "SVE eye" object).

```
    float
•   SVE_textDashDistance;
•   SVE_textInvisibleDistance;
```

These variables define the distance (in meters) at which text turns into dashes or becomes invisible, respectively.

```
    float
•   SVE_backgroundColor[3];
```

Defines the red, green, and blue components of the background of the rendered scene. See `SVE_setBackgroundColor()`.

```
    long
•   SVE_pdExtX,
•   SVE_pdExtY;
```

Predistortion paramters. See "SVE_PREDISTORT" on page 27.

---

float
- **SVE_FPSupperLimit,**
- **SVE_FPSlowerLimit;**

Frame control: upper and lower limits. If -1, not a limit. Currently, only the upper limit is enforced. If the frame rate is too fast, then the SVE system will introduce a delay that will maintain the upper frame rate limit.

---

char *
- **SVE_defaultTextureDirectory,**
- **SVE_defaultMaterialDirectory,**
- **SVE_defaultObjectDirectory,**
- **SVE_defaultWorldDirectory,**
- **SVE_defaultConfigDirectory,**
- **SVE_defaultAudioDirectory;**

Defines a directory list (directories separated by a ':' character) through which SVE will search for texture files, material files, object files, world files, display configuration files, and audio files, respectively.

---

char *
- **SVE_pointerObjectName;**

Defines the name of the pointer object (used for picking). The default is "FliteStik".

---

char *
- **SVE_pointerObjectFile;**

Defines the name of the object file used for the pointer object (used for picking). The default is "FliteStik.obj".

---

char *
- **SVE_selectorObjectName;**

Defines the name of the object that selects (during picking, usually a ray cast out into the environment). The default is "ray".

---

char *
- **SVE_selectorObjectFile;**

Defines the name of the object file used for the selector object. The default is "ray.obj".

---

char *
- **SVE_initFilename;**

Defines the file name of the initialization file. See `SVE_setInitFilename()`.

---

boolean
- **SVE_verbose;**

Determines if general information about what is occuring with the SVE system is printed (`TRUE`) or not (`FALSE`).

---

boolean
- **SVE_debug;**

Determines if debugging information on the SVE system as it is running is printed (`TRUE`) or not (`FALSE`).

---

char *
- **EVENT_SERVER_MACHINE,**
- **AUDIO_SERVER_MACHINE,**
- **TRACKER_SERVER_MACHINE;**

These character strings identify the machines that are running the event server, the audio server, or the tracker server, respectively.

---

char *
- **SVE_serverDirectory;**

Defines the directory in which the server programs can be found.

---

boolean
- **SVE_showFrameRate;**

If this variable is `TRUE` (which is the default), SVE will continuously print the current frame rate, as well as faces and lines rendered. If this variable is `FALSE` then SVE will calculate the frame rate but won't print it.

---

char *
- **SVE_defaultTextureMap;**

When loading a wavefront object it, it may contain texture vertices when no texture is specified. In this case, the material pointed to by this variable is loaded. By default this variable will point to the file `gvu_logo.rgb`. Changing this to `NULL` will tell SVE not to use any default textures.

4. SVE Data Structures

This is a brief description of all data structures in the SVE environment which can be used and manipulated by an application using the SVE system.

4.1. State information

```
typedef struct SVE_stateStruct {
    char              *programName;
    short int         windowType;
    SVE_window        windowData;
    list              objectTree;
    list              lightList;
    int               lastMaterial;
    list              materialList;
    list              pollingDevices;
    list              worlds;
    SVE_object        viewingObject;
    SVE_object        viewingObject2;
    M_matrix          viewingMatrix;
    M_matrix          perspectiveMatrix;
    char              currentEye;
    SVE_object        originObject;
    SVE_object        userObject;
    SVE_object        hmdObject;
    SVE_object        cursorObject;
    SVE_object        viewPlaneObject;
    SVE_point         origin;
    SVE_eventType     eventType;
    void              *eventData;
    boolean           ntscOn;
    SVE_config        config;
    float             flightSpeed;
    int               network;
    struct timeval    *beginTime;
    struct timeval    *frameTime;
    struct timeval    *lastFrameTime;
    struct timezone   *timeZone;
    list              userFrameCallback;
    list              userFrameEndCallback;
    list              animationCallback;
    list              cullingFunction;
    float             framesPerSecond;
} SVE_stateStruct;

typedef struct SVE_stateStruct *SVE_state;
```

This data structure is passed on to each callback routine defined in an SVE application. The global state structure can also be obtained using the function `SVE_getWorldState()`. This structure is the life blood of the SVE system. Every piece of information about the current state of the world is contained within it, including the current world definition, the current position and orientation of each tracker, and glove information (if activated). Most fields in the state structure are intended to be a source of information (`viewingMatrix`, `glove`, `eventType`, `eventData`, `config`), although some fields are intended to be altered for desired effects.

Here is a short description of each field:

---

char *
- **programName;**

Given name of the application. This is what appears in the application window's title bar.

---

short int
- **windowType;**

Defines the type of windowing system used. The current supported window types are `SVE_NO_WINDOW`, `SVE_IRIX_GL`, and `SVE_X`. The window type used is determined at compile time by which SVE library is used.

SVE_window
- **windowData;**

Stores the information on the window, which can be used by the application. If the window type is `SVE_NO_WINDOW`, then this field is `NULL`. If the window type is `SVE_IRIX_GL`, then this field points to a structure with this form:

```
typedef struct SVE_windowStruct {
   long int          windowID;
} SVE_windowStruct, *SVE_window;
```

where `windowID` is the id returned by `winopen()`. If the window type is `SVE_X`, then the field points to a structure with this form:

```
typedef struct SVE_windowStruct {
   long int          windowID;
   Display           *display;
   Window            window;
   Widget            widget;
   XtAppContext      app_context;
   GLXContext        glx_context;
} SVE_windowStruct, *SVE_window;
```

where `windowID` will be 1 if the window was successfully opened (or -1 if an error occured), and the X widget in which the rendering occurs is referenced by the `widget` field and the `glx_context`.

---

list
- **objectTree;**

This references another data structure that defines the objects in the SVE world (the objects SVE will render for each frame). The variable for the current world state is also known as `SVE_WORLD` in most functions.

---

list
- **lightList;**

List of light primitives (`SVE_lightRenderPtr`) that have been defined and will be rendered. Lights are kept in a separate list, which is generated at each frame, because they need to be "rendered" before anything else. This field points to a structure containing the light definition, the light id (used by the renderer), and the object which contains the light (and, in some cases, defines the light's position):

```
typedef struct SVE_lightRenderStruct {
   int              id;
   SVE_lightPtr     light;
   SVE_object       object;
} SVE_lightRenderStruct;
typedef SVE_lightRenderStruct *SVE_lightRenderPtr;
```

---

int
- **lastMaterial;**

This field holds the index to the last material created. See `SVE_getMaterialByIndex()`.

---

list
- **materialList;**

This is a linked list of materials (`SVE_material`) which have been defined for the object primitives.

---

list
- **pollingDevices;**

This is a linked list of defined polling devices (`SVE_pollDevice`). Polling devices are things such as trackers, gloves, or other continuous input devices or routines.

---

list
- **worlds;**

This contains a linked list of world server connections. (Warning...experimental!) The list is of `SVE_worldStruct *` pointers.

```
typedef struct SVE_worldStruct {
   char      *name;
   char      *address;
   int       outport;
   boolean   openned;
   int       pendingActions;
   list      actionBackLog;
   list      nameTranslation;
} SVE_worldStruct;
```

SVE_object
- **viewingObject;**

This is object used to generate the viewing matrix. The view will be towards the view plane object. This is initially set to the "SVE eye" object.

SVE_object
- **viewingObject2;**

This is object used to generate the viewing matrix for a second eyepoint (for stereo views). The view will be towards the view plane object. This is initially set to the "SVE other eye" object.

M_matrix
- **viewingMatrix;**

This is the viewing transformation placed on the modeling transformation stack which allows objects to be rendered from the point of view of the viewing object (looking at the view plane object) rather than from the world origin. This matrix is updated just before any rendering is performed. It is updated each frame just before rendering the scene (before frame callbacks are called, but after animation callbacks are called).

M_matrix
- **perspectiveMatrix;**

This is the perspective transformation placed on the perspective transformation stack to define the viewing volume from the eyepoint. It is updated each frame just before rendering the scene (before frame callbacks are called, but after animation callbacks are called).

---

char
- **currentEye;**

This value is `SVE_LEFT_EYE` or `SVE_RIGHT_EYE` depending on which eye point is currently being rendered. For monoscopic displays, this value will always be `SVE_LEFT_EYE`.

SVE_object
- **originObject;**

This references an empty object whose position matrix defines the position of the "origin" of the user's coordinate system, which is understood to be at the user's feet when she is standing in the center of the movement range. It is named "ORIGIN". The object that defines the tracking reference coordinate system, usually called "USER", is, by default, a child of this object. Moving the origin object effectively moves the user about in the environment.

SVE_object
- **userObject;**

This references an empty object whose position matrix defines either the tracker's reference coordinate system, which is the transmitter's position in relation to the "origin" (if the trackers are being used), or an average person's height from the "origin". It is named "USER". The two objects that often hold the tracker information ("SVE HMD" and "SVE cursor") are children of this object.

SVE_object
- **hmdObject;**

This references, by default, the "SVE HMD" object.

SVE_object
- **cursorObject;**

This references, by default, the "SVE cursor" object.

SVE_object
- **viewPlaneObject;**

This references, by default, the "SVE view plane" object, whose world location and orientation defines the location and orientation of the view plane, through which the user sees the scene (from the eye point(s)).

SVE_point
- **origin;**

OBSOLETE. Move the `originObject` SVE object (see above) to move the user around.

SVE_eventType
- **eventType;**

This is the event type. See "Events" on page 36. for a list of possible event types.

---

void *
- **eventData;**

This field points to the event data. It needs to be cast to the appropriate pointer based on the value in `eventType` (above). See "Events" on page 36. for more details.

---

boolean
- **ntscOn;**

This flag determines if the application window is on the computer screen (`FALSE`), or being sent to the scan converter for video output (`TRUE`).

---

int
- **config;**

This defines the configuration of the SVE system by bit-wise `OR`ing a combination of the option values. The options are listed in "Configuration Flags" on page 23.

---

float
- **flightSpeed;**

Determines the speed at which a user moves through the SVE world when "flying". Its is in meters per second.

---

int
- **network;**

This variable is used by SVE to see if it's getting tracker information, sending audio commands, and/or getting events over the network. It is a bit array. If the `NET_TRACKER` bit is set, then the tracking is done via the network (which is always true if tracking is done at all). If the `NET_EVENTS` bit is set, then events are sent to the application from a remote event server. If the `NET_SOUNDS` bit is set, then audio is performed on a remote machine.

---

struct timeval *
- **beginTime;**

This is the time at which the `SVE_init()` function call was made.

---

struct timeval *
- **frameTime;**

This is the time at which the current frame rendering was begun.

---

struct timeval *
- **lastFrameTime;**

This is the time at which the last frame rendering was begun.

---

struct timezone *
- **timeZone;**

This is the time zone for all time values stored in the `SVE_state` structure.

---

list
- **userFrameCallback;**

Linked list of defined frame callbacks (of type `SVE_functionPtr`).

---

list
- **userFrameEndCallback;**

Linked list of defined frame callbacks (of type `SVE_functionPtr`) that occur after the SVE rendering has been done.

---

list
- **animationCallback;**

Linked list of animation callbacks (of type `SVE_functionPtr`).

---

list
- **cullingFunction;**

Linked list of culling functions (of type `SVE_cullFunctionPtr`) that are called for each object to determine if an object should be rendered or not.

---

float
- **framesPerSecond**

The number of frames generated each second. This number starts of as being -1, and is updated after each frame.

4.2. SVE_object

```
typedef struct SVE_objectStruct {
    SVE_object          parent;
    char                *name;
    SVE_geometry        geometry;
    list                geometryList;
    boolean             highlight;
    SVE_geometry        highlightGeometry;
    M_matrix            position;
    M_matrix            worldPosition;
    boolean             visible;
    boolean             hascolor;
    boolean             selectable;
    SVE_boundaries      *boundaries;
    int                 cullable;
    boolean             hasVisibleSphere;
    float               visibleSphere;
    int                 materialIndex;
    int                 update;
    list                children;
    AnimationStruct     animation_vars;
    boolean             facingViewerUpright;
    boolean             facingViewer;
    SVE_functionPtr     menu_callback;
    list                frameCallback;
    SVE_widgetData      widgetData;
    SVE_vistype         visibility;
    boolean             remoteObject;
    list                worldIdList;
    SVE_multipleTexturesPtr moreTextures;
    void                *UserPtr;
} SVE_objectStruct;

typedef struct SVE_objectStruct *SVE_object;
```

`SVE_object` is a reference to a larger structure (`SVE_objectStruct`) which defines every detail about an object in the SVE environment.

What follows is a short description of each field in the SVE object structure:

SVE_object
- **parent;**

A reference to an object's parent. The children of the root of an object tree have `SVE_WORLD` for a parent.

---

char *
- **name;**

The string identifier of an object. The SVE library ensures that this name is unique for each object.

SVE_geometry
- **geometry;**

This references the geometry assigned to the object. If the object has more than one possible geometry (see below), then this is the geometry last chosen. It is possible that more than one object has the same geometry, in which case this many be one of many references to an object's geometry. It is important, therefore, to indicate to the SVE system if you change an object's geometry by hand by calling `SVE_initGeometryChange()` before making any changes.

---

list
- **geometryList;**

List of posible geometries for an object. When an object is to be rendering, this list is traversed, and each geometry is tested to see if it is "valid". The first valid geometry is chosen, and is placed in the geometry field. A geometry is valid if it has no distance constraints, or if the user's eye and the object's origin are within the given distance constrants. This field is a linked list of `SVE_geometryEntry` pointers, which point to the following structure:

```
typedef struct SVE_geometryEntryStruct {
    SVE_geometry geometry;
    boolean      minRange;
    float        minDist;
    boolean      maxRange;
    float        maxDist;
} SVE_geometryEntryStruct;

typedef SVE_geometryEntryStruct *SVE_geometryEntry;
```

---

boolean
- **highlight;**

The value indicates if the object is currently being highlighted or not. An object may have a defined color change or geometry change when this value changes. The best method to "highlight" and "un-highlight" an object, though, is to use the `SVE_highlightObject()` function.

SVE_geometry
- **highlightGeometry;**

This is a geometry which, if defined, will replace the object's usual geometry when the `highlight` field of the object is `TRUE`.

M_matrix
- **position;**

This 4X4 matrix defines the position and orientation of the object in relation to its parent.

M_matrix
- **worldPosition;**

This 4X4 matrix defines the position and orientation of the object in world coordinates. It is not guaranteed to be correct, as it is not updated regularly. Use the `SVE_getWorldMatrix()` function to find the correct world position.

---

boolean
- **visible;**

This flag determines if the object is rendered by SVE while rendering the current frame. If `TRUE`, the object and its children are rendered, if `FALSE` the object and its children are not rendered.

---

boolean
- **hascolor;**

Determines if an object has one global color value, which is defined in the `materialIndex` field. This color will override the materials given in the object's geometry descriptions.

---

boolean
- **selectable;**

Determines if an object is considered for selection during and intersection or picking algorithm the SVE system uses to determine which object has been selected.

SVE_boundaries *
- **boundaries;**

This references a structure that determines the bounding volume of the object (used for intersection algorithms). These boundaries include the boundaries of the child objects.

---

int
- **cullable**

This value is one of `NOT_CULLABLE`, `CULLABLE_BOX`, or `CULLABLE_SPHERE`. It determines if the object can be considered for culling from the view frustrum (checking to see if the object can be seen, and if not, not rendering it), or whether the object should be considered for culling using a bounding box or sphere (from the object's boundary definition).

---

boolean
- **hasVisibleSphere;**

Determines if an object has a given range within which it is visible.

---

float
- **visibleSphere;**

This is the range at which an object is visible. An object is visible if it is within this distance from the viewing object (usually the "SVE HMD" object) in the SVE world, and not visible if the viewing object is outside of this range.

---

int
- **materialIndex;**

This is an index to the material used to color the object when `hasColor` is `TRUE`.

---

int
- **update;**

These are single bit flags that indicate if certain attributes of an object need to be updated. It is usually set by SVE functions and need not be touched by an application. If, however, an attribute changes without using an SVE function to change it, the appropriate bit should be set by doing a bit wise OR of this with the appropriate set of these:

```
CALC_ALL
CALC_WORLD_MATRIX
CALC_OBJECT_BOUNDS
```

To simplify matters, the `SVE_reCalculateWorldMatrix()` and `SVE_reCalculateObjectBoundaries()` functions are provided to set the `CALC_WORLD_MATRIX` and `CALC_OBJECT_BOUNDS` flags, respectively, and to take care of possible problems.

AnimationStruct
- **animation_vars;**

This is a structure containing state variables for the animation of the object. The structure has the following form: (See "Animation Routines" on page 41 for more details.)

```
typedef struct
{
   double   xvel, yvel, zvel,
            xacc, yacc, zacc,
```

```
                 xrot, yrot, zrot;
    double       lasttime;
    SVE_animateObjectFunctionPtr UserFunc;
    boolean      animate;
    void         *userdata;
} AnimationStruct;
```

---

list
- **children;**

This is a linked list of the child objects for this object. It is unordered and unbounded in number.

---

boolean
- **facingViewerUpright;**

This flag determines if the SVE system rotates the object so that it is always facing the user, but remaining upright (i.e. the Z axis is always pointing towards the user on the X-Y plane).

---

boolean
- **facingViewer;**

This flag determines if the SVE system rotates the object so that it's negative Z axis is always pointing towards the user.

---

list
- **frameCallback;**

This is the list of frame callbacks (of type `SVE_functionPtr`) associated with the object. If the object is about to be rendered, these callback functions will be called first. Anything drawn will be drawn in the local coordinate system of the object.

SVE_widgetData
- **widgetData;**

This structure contains the data associated with objects which are also 3D interactors. See "3D interactors" on page 91.

SVE_vistype
- **visibility;**

Indicates the object's visibility, as determined when SVE prepares to render a frame (before the frame callbacks, but after the animation callbacks). The possible values for this field are `SVE_ALL_VISIBLE`, `SVE_PART_VISIBLE`, `SVE_CHILDREN_ONLY_VISIBLE`, `SVE_NONE_VISIBLE`, and `SVE_VISIBILITY_UNKNOWN`.

```
boolean
```
- **remoteObject;**

This field is `TRUE` if the object was obtained from a remote world server, not created by the SVE application itself.

```
list
```
- **worldIdList;**

This is a linked list of world servers in which the object belongs. (Warning... Experimental!)

```
SVE_multipleTexturesPtr
```
- **moreTextures;**

If this field is not NULL, then it points to the following structure, which contains a list of textures, and a function which can switch through the texture list to define the textures of the object's materials.

```
typedef struct SVE_multipleTexturesStruct {
   int    nTextures;
   int    textureMode;
   int    currentIndex;
   char** textureNames;
   long*  textureIDs;
   SVE_objectFunctionPtr swapCallback;
} SVE_multipleTexturesStruct;

typedef SVE_multipleTexturesStruct *SVE_multipleTexturesPtr;
```

```
void *
```
- **UserPtr;**

This pointer is not used in SVE. The programmer can use it to attach its own data structures to an object.

4.3. SVE_geometry

The geometry of any object is stored in a separate structure so that many object can use the same geometry if they wish to. Each geometry is uniquely identified by the `primitivesFilename` field, which is usually the name of the file, uncluding its complete path, from which the geometry was obtained. This would be an object file.

```
typedef struct SVE_geometryStruct {
   list            primitives;   /* Linked list of SVE_primitive */
   int             lastPrimitive;
   SVE_primitive   label; /* for hypertexts */
   char           *text;  /* for hypertexts */
   char           *textfile;
   boolean         primitivesChanged;
   char           *primitivesFilename;
   long            glID;
   int             firstMaterial;
   int             lastMaterial;
   int             facesRendered;
   int             linesRendered;
   SVE_boundaries *primitiveBounds;
```

```
    boolean         containsLight;
    list            transparentFaces;  /* Linked list of SVE_transparentFace */
    int             update; /* Bit flags to indicate if certain attributes
                                 need to be updated when needed (lazy update) */
    int             objectLinks;
} SVE_geometryStruct;

typedef SVE_geometryStruct *SVE_geometry;
```

- **`list`**
  `primitives;`

This is a linked list of primitives which make up the geometry. The list contains items of type `SVE_primitive`.

- **`int`**
  `lastPrimitive;`

This is the index (or tag) of the last primitive created.

- **`SVE_primitive`**
  `label;`

Hypertext that will be displayed at the object's position and orientation (given in the object's world position matrix).

- **`char *`**
  `text;`

Obsolete.

- **`char *`**
  `textfile;`

User to save the hypertext if it has been changed.

- **`boolean`**
  `primitivesChanged;`

This flag indicates if a geometry has changed, which means that it needs to be saved if the object is saved to file.

- **`char *`**
  `primitivesFilename;`

This is the unique character string that identifies the geometry. It is usually the name of the object file from which the geometry was loaded.

Object
- **glID;**

This is the handle for a GL object which represents the geometry. It is computed each time the geometry changes. It does not include any text primitives.

SVE_boundaries *
- **primitiveBounds;**

This references a structure that stores the bounding volume of the group of primitives that make up the geometry.

---

int
- **update;**

These are bit flags which indicate if certain attributes of an object need to be updated. It is usually set by SVE functions that alter the geometry. The bits used are defined as:

```
CALC_ALL
CALC_PRIMITIVE_BOUNDS
CALC_GL_OBJECT
```

---

int
- **objectLinks;**

This is the number of objects which reference this particular geometry structure.

4.4. SVE_geometryEntry

```
typedef struct SVE_geometryEntryStruct {
   SVE_geometry geometry;
   boolean      minRange;
   float        minDist; /* if (minRange), minimum distance from user
                             to geometry origin */
   boolean      maxRange;
   float        maxDist; /* if (maxRange), maximum distance from user
                             to geometry origin */
} SVE_geometryEntryStruct;

typedef SVE_geometryEntryStruct *SVE_geometryEntry;
```

4.5. SVE_primitive

```
typedef struct SVE_primitiveStruct {
   int               tag;
   SVE_primitiveType type;
   DA_array          pointList;    /* Dynamic array of SVE_pointPtr */
   DA_array          normalList;   /* Dynamic array of SVE_normalPtr */
   DA_array          txtvertices;  /* Dynamic array of SVE_textureCoordPtr */
   list              faceList;     /* Linked list of SVE_facePtr */
   int               materialIndex;
   char             *text;
```

```
    int                 lineWidth;
    M_matrix            matrix; /* only used for text primitives.. */
    SVE_lightPtr        light;
    boolean             isMenuEntry;  /* only ways to tell it's a menu obj. */
    boolean             isMenuHeader;
    int                 highlightMaterial;
} SVE_primitiveStruct;

typedef SVE_primitiveStruct *SVE_primitive;
```

A short description of each field of the SVE_primitive follows.

- **int**
  tag

Integer value used to distinguish a primitive from the other primitives of a geometry.

- **SVE_primitiveType**
  type;

This defines the type if primitive. Possible values are:

```
POLYHEDRON            Primitive consists of a list of faces.
LINE                  Primitive consists of one line with many points.
TEXT                  Primitive consists of a string of characters.
TEXTURED_POLYHEDRON   The same as POLYHEDRON but has a texture.
LIGHT                 Declares a light source. There is a maximum of 8
                      light sources.
```

- **DA_array**
  pointList;

Array of points defined for the primitive. Each point has a location and an optional color. The items of the array are of type `SVE_pointPtr`, which is defined below.

```
typedef struct SVE_pointStruct {
    int         index;
    SVE_point   point;
    float       color[4];  /* vertex diffuse color if different */
    float       bwColor;
} SVE_pointStruct;
typedef SVE_pointStruct *SVE_pointPtr;
```

- **DA_array**
  normalList;

Array of normal vectors defined for the primitive. The items of the array are of type `SVE_normalPtr`, which is defined below.

```
typedef struct SVE_normalStruct {
    int         index;
    SVE_point   normal;
} SVE_normalStruct;
typedef SVE_normalStruct *SVE_normalPtr;
```

DA_array
- **txtvertices;**

Array of texture vertices defined for the primitive. The items of the array are of type `SVE_textureCoordPtr`, which is defined below.

```
typedef struct SVE_textureCoordStruct {
    int    index;
    float  texcoord[2];
} SVE_textureCoordStruct;
typedef SVE_textureCoordStruct *SVE_textureCoordPtr;
```

---

list
- **faceList;**

Linked list of faces (for a `POLYHEDRON`) or lines (for a `LINE`). Each face has a list of vertices (which consists of a point, an optional normal vector, and an optional texture vertex), an index to a material, and a normal vector (which is used for flat shading). The linked list contains items of type `SVE_facePtr`, which is defined below.

```
typedef struct SVE_faceStruct {
    int   index;
    list  vertexList;  /* Linked list of SVE_vertexPtr */
    int   materialIndex;
    SVE_normalPtr   normal;  /* for flat shading */
} SVE_faceStruct;
typedef SVE_faceStruct *SVE_facePtr;
```

The list of vertexes of a face is a linked list of items of type `SVE_vertexPtr`, which is defined below.

```
typedef struct SVE_vertexStruct {
    SVE_pointPtr   point;
    int            pointIndex;
    SVE_normalPtr  normal;
    int            normalIndex;
    SVE_textureCoordPtr texCoord;
    int            texCoordIndex;
} SVE_vertexStruct;
typedef SVE_vertexStruct *SVE_vertexPtr;
```

---

int
- **materialIndex;**

An index to a material used as a global material to color the primitive.

---

char *
- **text;**

Text string used for a `TEXT` primitive.

---

int
- **lineWidth;**

Line width used for a `LINE` primitive.

M_matrix
- **matrix;**

Transformation matrix used for the `TEXT` primitive.

SVE_lightPtr
- **light;**

Refers to a structure which defines the parameters for a `LIGHT` primitive. The definition for `SVE_lightPtr` is given below.

```
typedef struct SVE_lightStruct {
    int         id;
    boolean     on;
    float       ambient[3];
    float       emission[3];
    SVE_point   position;
    boolean     atInfinity;
    float       spotExponent;
    float       spotSpread;
    SVE_point   spotDirection;
} SVE_lightStruct;

typedef SVE_lightStruct *SVE_lightPtr;
```

These parameters mirror the options given in an object file for a light primitive, which is discussed in "LIGHT" on page 120.

---

int
- **highlightMaterial**

This is the material index for the material used when the object which uses this geometry is highlighted. If this value is -1, then the primitive will not change materials when the goemetry is highlighted.

4.6. SVE_boundaries

This structure defines the bounding volume of an object. There are two possible bounding shapes, a sphere or a box. Only relevant fields are used for each object according to the type of bounding volume it has.

```
typedef struct SVE_boundaries{
    boolean           hasSphere;
    SVE_point         sphereOrigin;
    float             sphereRadius;
    boolean           hasBox;
    SVE_point         boxVertex1;
    SVE_point         boxVertex2;
} SVE_boundaries;
```

4.7. SVE_material

The SVE system maintains a list of materials which the application's objects can use, and to which the application can add newly created materials. The properties of a particular material are best changed using the appropriate SVE function (See "Colors and materials" on page 74.), however it may be useful to examine a material's current properties. The structure for material definitions is given below.

```
typedef struct SVE_materialStruct {
    char   *name;
```

```
    int     index;
    long    changed;
    float   ambient[3];
    float   bwAmbient;
    float   diffuse[3];
    float   bwDiffuse;
    float   specular[3];
    float   bwSpecular;
    float   emission[3];
    float   bwEmission;
    float   shininess;
    float   alpha;
    int     textureMode;
    long    textureEnv;
    long    textureID;
    char   *texture;
    char   *textureFilename;
    boolean transparentTexture;
} SVE_materialStruct;

typedef struct SVE_materialStruct *SVE_material;
```

4.8. SVE_point

The SVE_point data type is simply an array of three floats that define a location or vector.

```
typedef float SVE_point[3];
```

4.9. SVE_status

This type is returned by SVE callback functions. Currently, it is only significant for event callbacks, where the callback should return one of these values:

```
EVENT_IGNORED
EVENT_CONSUMED
```

The return value indicates whether the event was used and no further action should be taken on it, or whether it was ignored (and possibly a default handler should process the event).

Other types of callbacks can return `SVE_OK`.

4.10. SVE_widgetData

When an object is used as a 3D interactor, or widget, then some additional information is stored with the object in the `SVE_widgetData` structure.

```
typedef struct SVE_widgetDataStruct {
    char *type;
    void *data;
} SVE_widgetDataStruct, *SVE_widgetData;
```

The items stored are the type of widget (a character string), and the data associated with the widget, which is dependent on the type of widget it is.

4.11. SVE_pollDevice

The data associated with a polling device is stored in a structure of type `SVE_pollDevice`.

```
typedef struct SVE_pollDeviceStruct {
    void                 *deviceHandle;
    int                   type;
    int                   id;
    char                 *attachedToName;
```

```
    SVE_object            attachedTo;
    SVE_pollFunctionPtr   pollFunction;
    SVE_pollFunctionPtr   closeFunction;
    void                 *data;
} SVE_pollDeviceStruct;

typedef struct SVE_pollDeviceStruct *SVE_pollDevice;
```

The meaning of each of these fields is described below.

- **`void *`**
  `deviceHandle;`

This is usually the device handle obtained when initializing the device. Similar to a file descriptor handle.

- **`int`**
  `type;`

This is the type of the device. Two types are already defined: `SVE_TRACKER_DEVICE` and `SVE_GLOVE_DEVICE`. Other types can be defined.

- **`int`**
  `id;`

This is the unique integer number identifier of the particular device amoung the devices of a particular type.

- **`char *`**
  `attachedToName;`

This is the name of the SVE object with which the polling device is associated.

- **`SVE_object`**
  `attachedTo;`

This is the reference to the SVE object with which the polling device is associated. If the object identified by the `atttachedToName` string does not exist yet, then this field is `NULL`. When the object is created, then this field will be updated to refer to it.

- **`SVE_pollFunctionPtr`**
  `pollFunction;`

This is the function that is called periodically to update the device's state. The function should be of the following form.

```
SVE_status SVE_pollFunction(SVE_pollDevice device, SVE_state state);
```

- **`SVE_pollFunctionPtr`**
  `closeFunction;`

This is the function that is called when the device is to be closed.

- **`void *`**
  `data;`

This pointer refers to the data specific to the polling device, which was given to the SVE system when the device was created.

4.12. SVE_gloveData

Hand input devices are a special case of the generic polling device. The data associated with each hand input device is of type `SVE_gloveData`, and it is shown below.

```
typedef struct SVE_gloveData {
   boolean               gloveActive;
   char                  *port;
   char                  *handfilename;
   JointCalStruct        handCal[FNGRS+1][JNTS];
   boolean               gloveButton;
   float                 angle[FNGRS+1][JNTS];
   float                 tipPosition[FNGRS][3];
   CyberGlove            *rawGloveData;
   boolean               checkForGesture;
   gestureStateStruct    *gestureState;
} SVE_gloveData;
```

What follows is a short description of each field of this structure:

- **`boolean`**
  `gloveActive;`

Determines if the glove input device is active.

- **`char *`**
  `port;`

Interface port to which the glove device is attached to the machine.

- **`char *`**
  `handfilename;`

File name of the last hand calibration file used.

JointCalStruct
- **handCal[FNGRS+1][JNTS];**

Contains the offset and gain values for each joint which are used to determine an angle from the value given by the glove device. These values are unique for each person.

```
typedef struct JointCalStruct {
    int offset;
    float gain;
} JointCalStruct;
```

The formula used is `angle = gain * (device_value - offset)`.

---

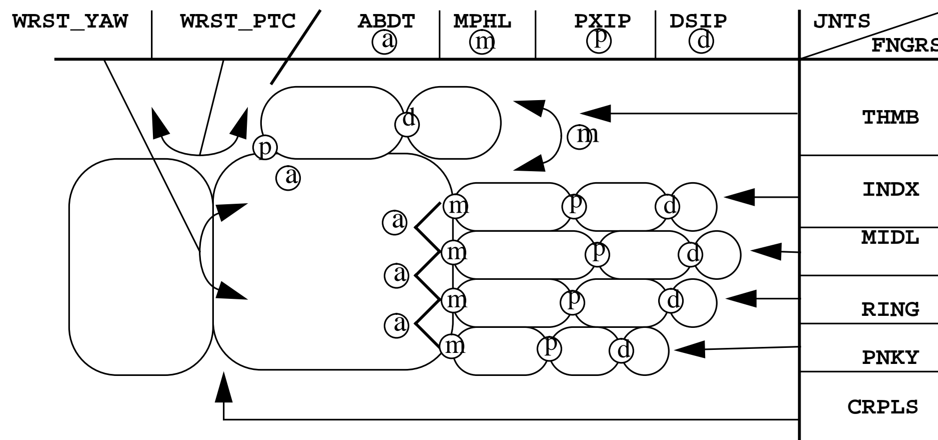
boolean
- **gloveButton;**

Indicateswhether or not the glove button is on (`TRUE`) or off (`FALSE`).

---

float
- **angle[FNGRS+1][JNTS];**

This 2D array stores the angle values of the joints of each finger which were polled at the last input poll stage of the SVE system. The `FNGRS` and `JNTS` values are defined in "`cg_glove.h`". A particular angle can be indexed as follows: (See the Cyberglove documentation for more detail.)



Mapping of angle indexes to bends in hand

SVE_point
- **tipPosition[FNGRS];**

Position in absolute world coordinates of the fingertips.

CyberGlove *
- **rawGloveData;**

If the glove input device is active, this references the data structure used by the CyberGlove device drivers. This should not be used if possible.

---

boolean
- ***checkForGesture;***

Flag to determine wether the current glove position should be compared to the list of possible gestures.

---

gestureStateStruct *
- **gestureState;**

This references a list of glove gestures, sorted by their given priority. If the `checkForGesture` flag is set, and the current glove position matches one of the gestures on this list, then a SVE_`GESTURE` event is generated. The event value will be `MIN_GESTURE` + (gesture index), where the gesture index is the priority of the gesture.

4.13. gestureStateStruct

```
typedef struct gestureStateStruct {
    char *filename;
    int numGestures;
    gestureDef *gestureList;
    jointDef currentJointValues[FNGRS+1][JNTS];
    gestureProbType *currentProb; /* Array of gesture probabilities */
    boolean *potentialGesture; /* True if a particular gesture is in
                                  transition from start to end */
    int lastGesture;
} gestureStateStruct;
```

4.14. gestureDefStruct

This defines a gesture used to match against the current configuration of the glove device. The priority determines the index value given to each gesture event that occurs when that gesture is matched. If a current configuration matches more than one gesture, the gesture with the highest priority (1) will be given as the match.

```
typedef float *probList;

typedef struct gestureDefStruct {
    char *name;
    short priority;
    jointDef jointValues[FNGRS+1][JNTS];
    int lastPosture[FNGRS+1][JNTS];
    int handPosture[FNGRS+1][JNTS]; /* contains the states of each sensor
                                       as index into probability list */
    probList startProb[FNGRS+1][JNTS];
    int allowedTrans[FNGRS+1][JNTS];
    probList endProb[FNGRS+1][JNTS];
} gestureDefStruct;
```

APPENDIX D: Tracker Library

The SVE library uses another library which handles the communication between the computer and the tracker device through the serial line. This library abstracts out the notion of tracker devices, and deals with tracking receivers on a one to one basis. In addition, it is possible to have many references to the same receiver at the same time.

1. Using a tracker receiver

A particular receiver is identified with a triplet of (tracker device type, which port the device is connected to, and which of the receivers attached to the device to use). For example, if one wished to use the second receiver of an Isotrak II, which is connected to the second serial port of an SGI computer, the receiver would be identified as (IsotrakII, "/dev/ttyd2", 2).

A tracker receiver is opened with the function, `TRK_open()`, which takes three values representing the triplet described above, and returns a reference to that tracker receiver of type `trkDevice`. That reference value is used to later retrieve the current position matrix of the tracker, using `TRK_getMatrix()`, and, if appropriate, the current state of the tracker's buttons using `TRK_getButtons()`. Most tracking devices do not have buttons associated with them, but if they do, the `TRK_getButtons()` function returns the state of the buttons, and which button(s) have changed since inquiring last. The tracker receiver can be "closed", indicating that it is no longer to be used, and structures associated with it can be freed if no one else is using them, by calling the function `TRK_close()`.

The `TRK_open()` function requires three values to indicate which receiver to use. The first value is the tracker device type, which can be one of the following values: `BIRD`, `ISOTRAKII`, `FASTRAK`, or `BOOM`; indicating an Ascension™ Bird, Polhemus™ Isotrak II, Polhemus™ Fastrak, and FakeSpace™ Boom tracking device respectively. The functions `TRK_getTypeId()` and `TRK_getTypeName()` translate between the integer value and character string identification value for each type.

The second value is the serial port identification, which should be a string indicating port the port to which the tracker it attached, such as "/dev/ttyd1".

The third and last value required by the `TRK_open()` function is the receiver value, which is a number greater than or equal to one. The maximum value of this parameter depends on the tracking device. For example, a Boom device only has one tracker receiver (the head position), while a Fastrak device has up to four receivers.

The table below outlines the configuration of each tracking device supported by the tracker library. The last column gives the orientation of the transmitter that will produce a Y up coordinate system within which the tracker matrix is given.

Table 12: Configuration of Supported Tracker Devices

Tracker	Tracker type	Tracker name	Max receivers	Baud	Buttons	Orientation
Ascension Bird	BIRD	bird	8	38400	none	Z down
Polhemus IsotrakII	ISOTRAKII	isotrackii	2	9600	none	Y up
Polhemus Fastrak	FASTRAK	fastrak	4	38400	Button 4 (receiver 1)	Y up
FakeSpace Boom2C	BOOM	boom	1	9600	4	NA
Virtual I/O iGlasses	IGLASSES	iglasses	1	9600	none	NA

Some tracking devices are only valid for a given hemisphere around a reference point, rather than the entire sphere around the point. If this is the case, then the hemisphere is defined by a vector pointing in the direction of the center of the hemisphere. This is often pointing down (towards the floor) when the reference point is above the user's head. For this devices, the hemisphere can be changed using the function `TRK_changeHemi()`. For newer devices, this call is effective only for the receiver indicated (although it affects all user's of that particular receiver), for old devices, this call affects all receivers attached to the device.

2. Tracker function definitions

An explanation for each function provided by the tracker library is given below.

---

trkDevice
- **TRK_open(int deviceType, char *port, int receiver);**

`int deviceType`	`Device type identifier.`
`char *portId`	`Port name.`
`int receiver`	`Receiver identifier.`

Opens a given receiver of a given tracker device which is connected to the computer using the given serial port. The `deviceType` parameter can be one of `BIRD`, `ISOTRAKII`, `FASTRAK`, or `BOOM`. The `port` parameter is the name of the serial port ("/dev/ttyd1", for example). The `receiver` parameter is a value greater than or equal to 1, and less than or equal to the number of receivers attached to the specified device. The value returned should be used in any other function calls which should use the receiver opened by this call.

This function returns `NULL` if the receiver could not be opened.

---

void
- **TRK_getMatrix(trkDevice receiver, float matrix[4][4]);**

`trkDevice receiver`	`Receiver identifier.`
`float matrix[4][4]`	`Position matrix.`

Sets the `matrix` parameter to the current position of the receiver identified by the `receiver` parameter. The position matrix includes the location and orientation of the tracker receiver.

---

void
- **TRK_getButtons(trkDevice receiver, int buttons[], int buttonChange[]);**

`trkDevice receiver`	`Receiver identifier.`
`int buttons[]`	`The current value of each button.`
`int buttonChange[]`	`Which button(s) have changed.`

This function returns the current state of the buttons of the given `receiver` in the `buttons` array. There is one value (0 for up, 1 for down) for each button associated with the tracker device which is stored in the `buttons` array, and a value indicating if a button's status has changed (1 for changed, 0 for no change), which is stored in the `buttonChange` array.

---

void
- **TRK_changeHemi(trkDevice receiver, float vector[]);**

`trkDevice receiver`	`Receiver identifier.`
`float vector[]`	`Hemisphere center vector.`

Changes the hemisphere for which the given `receiver` is valid using the `vector` parameter. The `vector` parameter defines the direction in which the center of the hemisphere lies. This function is only required for certain trackers that are only valid for one hemisphere, rather than the entire sphere around a reference point. An example of the type of tracker is an electromagnetic tracker.

---

void
- **TRK_close(trkDevice receiver);**

`trkDevice receiver`	`Receiver identifier.`

Closes the given `receiver`. If this is the last reference to the particular receiver, then the receiver's data structures are freed. If this is the last reference to any receiver of the device it is attached to, then that device is shut down by software.

---

int
- **TRK_getTypeId(char *typeName);**

`char *typeName`	`Character string representation of a tracker device type.`

Returns the integer value associated with a particular tracker device type representation.

---

char *
- **TRK_getTypeName(int typeId);**

`int typeId`	`Integer identification of a tracker device type.`

Returns the character string representation of a tracker device type identified by the given integer value.

3. Tracker data type

The `trkDevice` data type, which is returned by the `TRK_open()` function, and is used from then on to identify the tracker receiver that has been open, is defined as follows:

```
typedef struct receiverDefStruct {
    int trkType;
    int trkPort;
    int receiver;
} receiverDefStruct;

typedef receiverDefStruct *trkDevice;
```

The `trkType` field identifies the type of the receiver's tracker device. The `trkPort` field is an integer that differentiates this device from other tracker devices. The `receiver` field identifies the receiver number.

APPENDIX E: Inset Utility

The inset utility used to be integrated in SVE version 1.5. Because it is dependent on GL routines specific for the SGI Reality Engine graphics, and was developed to demonstrate a point (about perceptual requirements of VE images in HMD's), but does not provide a performance increase, it has been removed from the SVE library. Interested developers may still use the utility by including the inset.c code (found in the misc directory of the SVE source), and calling the following functions:

```
void INSET_init(SVE_config config);
```

This function should be called after `SVE_init()`. It sets up the inset configuration.

```
void SVE_insetReset(void);
```

This function can be called to reset the inset configuration after parameters have been changed.

This option has not been tested with many SVE configurations, including predistortion and stereo. (Insetting with predistortion could work, and the combination may be implemented at a later date).

To control insetting, users may set globally declared variables: `SVE_hdExtX`, `SVE_hdExtY`, `SVE_ldExtX`, and `SVE_ldExtY`, which control the horizontal and vertical extent of the source high and low detail windows, and `SVE_hdXPctg` and `SVE_hdYPctg`, which are percentages and control the size of the central image in the destination window. The extent of the destination window is controlled by the standard window globals `SVE_minX`, `SVE_minY`, `SVE_sizeX`, and `SVE_sizeY`. Note that large source extents are not really necessary HMD's with low resolution.

In order to set the source extent variables, users may want to know something about the insetting destination window's configuration. They can get this information with the routine `SVE_getInsetParams()`. This routine returns two sets of four variables. The first set describes only the non-overlapping (see below) part of the inset. The second describes the full inset, including overlapping portions. The first two variables of each set describe the horizontal/vertical ratios of inset to insetting window. The second two variables of each set are the field of view and aspect ratios as used by the GL command `perspective`. These latter two variables may be used quite easily with the `SVE_objectVisible()` routine.

```
void SVE_getInsetParams(SVE_config config,
                        float *hd_inHRatio, float *hd_inVRatio,
                        int *hd_innerFovY, float *hd_innerARatio,
                        float *hd_outHRatio, float *hd_outVRatio,
                        int *hd_outerFovY, float *hd_outerARatio);
```

Insets of more or less than 100% in either dimension are allowed.

The routine `SVE_insetReset()` allows users to reset the inset configuration after the extent and percentage variables have been changed. It is not necessary to call this routine before calling `SVE_getInsetParams()`. `SVE_insetReset()` takes a window title as its parameter.

void SVE_insetReset(void);

The insetting composition is performed with the SGI's dynamic texturing hardware. The source images are treated as a textures. In the destination window, a point grid is precalculated, and the source images are textured onto this grid. There is a trade-off between texel interpolation and grid precision (the number of grid points). This may be controlled with the constants `NUM_CELLSX` and `NUM_CELLSY` (internal variables). The central inset is rendered in front of the low detail surround texture, and it overlaps that texture at its edges. Transparency increases at the edges to achieve a gradual fade from high to low detail.

Note that image generation and insetting must be synchronized. If not, the insetting routines will load incomplete frames as textures. In SVE, this is accomplished by used the same process for rendering and insetting.

Index of Example Applications

"Hello World" Example, My First SVE Program. 8
`example1.world`, the World Description File 10
`hello_world.object`, the Object Geometry File 10
`.sve.init`, the initial configuration file. 11
example2, application using a callback 18
.sve.init, an Example Initialization File 21
Example Initialization File 28
Animation Example 42
Example 3, Animation and Event Callback Demonstration 44
Example5, Manipulating the Object Tree in the Environment. 54
Modified Example 3, Rotating Many Cubes 56
all_primitives.object, an Example of Each Primitive Used in the SVE Library. 64
Primitive Construction Example 69
Example SVE object with lights 76
example4, Application Using Light. 76
World File Used in Example4. 77
Sound Example 79
Gesture File Example 85
glove_example 87
Simple Button Widget Example 92
Object File Example 112
Initialization File Example 113
Example Display Configuration File, Head Mounted 114
Example Display Configuration File, Monitor 114
Gesture File example 114

Index of Figures

Configurations Provided by the SVE System 7
snapshot of the first example. 9
Global control flow. 13
Example Object Tree 14
SVE data files 15
The interaction loop. 16
Concepts of a notifier mechanism and the event lookup table 17
The control flow of the vr-application. 18
Objects Inserted From Initialization File in Source vi: 21
SVE System Overview 35
Object Tree Example 53
Default World Tree and Example World Tree 59
Inserting Tracker Object as Parent of Attached Object ("MoveMe") 61
The Propagation of Transformations From Parent to Child Objects. 63
Snap Shot of an Object Containing All Possible Primitive Types. 65
The Normal Resulting From Different Vertex Orderings. 69
Hand SVE Object Tree 84
Widget Example Screen Shot. 91
Depth first notation of a tree hierarchy. 118
Mapping s,t Coordinates to Polygons 123

Index

A
addToList 177
addToListEnd 177
addToListFront 177
addToListSorted 178
AnimationStruct 203
appendList 180
AUDIO_SERVER_MACHINE 192

C
copyList 180
createList 177

D
DA_arrayMax 182
DA_arrayMin 182
DA_createArray 181
DA_empty 181
DA_free 182
DA_get 181
DA_makeCopy 182
DA_merge 182
DA_print 183
DA_store 181

E
Event Data Structures 39
Event Types 37
EVENT_CONSUMED 16
EVENT_IGNORED 16, 19
EVENT_SERVER_MACHINE 192

F
findData 178
FP_appendToPath 183
FP_closeFile 183
FP_findFile 185
FP_getCurrentLine 184
FP_getFilename 183
FP_getNextBoolean 184
FP_getNextFloat 184
FP_getNextInt 184

FP_getNextToken 184
FP_getRemainingLine 184
FP_nextLine 184
FP_openFile 183
FP_prependToPath 183
FP_reportFileError 183
FP_setCommentToken 183
FP_setSeparateToken 184
freeList 180

G
GESTURE 87
getData 178
getNext 178

L
listEmpty 179

M
M_copy 187
M_invert 187
M_loadID 187
M_matrix 187
M_matVectMult 187
M_mult 187
M_vectMatMult 187

O
ORIGIN 60

P
printList 181

R
removeData 179
removeFirst 179

S
server-audio 99
server-event 99
server-tracker 99
sortList 179
SVE cursor 60
SVE eye 60
SVE HMD 60
SVE_abort 134

SVE_addAnimationCallback 16, 43, 152
SVE_addChildToObject 54, 154
SVE_addCullingFunction 49, 163
SVE_addHighlightPrimitiveToObject 71, 143
SVE_addObjectGeometry 67, 142
SVE_addObjects 62, 134
SVE_addPollingDevice 89, 171
SVE_addPrimitiveToGeometry 71, 144
SVE_addPrimitiveToObject 71, 143
SVE_addToObjectList 58, 154
SVE_addToWorldTree 58, 154
SVE_aspectRatio 188
SVE_attachSoundToObject 81, 175
SVE_attachToObject 54, 153
SVE_attachTracker 83, 171
SVE_audioCheckSound 80, 174
SVE_audioCloseSound 174
SVE_audioGetVolume 80, 175
SVE_audioOpenSound 80, 174
SVE_audioReplaySound 80, 174
SVE_audioSetVolume 80, 174
SVE_audioStopSound 80, 174
SVE_availableHardware 189
SVE_backgroundColor 190
SVE_beginEventLoop 15, 133
SVE_beginPrimitive 67, 144
SVE_beginPrimitiveClosedLine 68, 145
SVE_beginPrimitiveFace 68, 145
SVE_beginPrimitiveLine 68, 145
SVE_boundaries 73, 209
SVE_BW 24, 25
SVE_calculateBounds 158
SVE_changeConfig 23, 167
SVE_changeObjectGeometry 142
SVE_changeObjectName 51, 138
SVE_changeSoundUpdateRate 81, 175
SVE_changeText 72, 141
SVE_clearWorld 153
SVE_copyBoundaries 158
SVE_copyMatrix 169
SVE_createBoxBoundaries 158
SVE_createEmptyBoundaries 157
SVE_createEmptyObject 52, 137
SVE_createEmptyPrimitive 147
SVE_createFace 148
SVE_createGeometry 71, 143

SVE_createLight 149
SVE_createLine 149
SVE_createMaterial 74, 150
SVE_createNormal 148
SVE_createObjectCopy 52, 138
SVE_createPoint 148
SVE_createPointWithColor 148
SVE_createPollingDevice 89, 170
SVE_createSphereBoundaries 158
SVE_createTexCoord 148
SVE_createVertex 148
SVE_cullFunction 48
SVE_debug 192
SVE_defaultAudioDirectory 191
SVE_defaultConfigDirectory 191
SVE_defaultMaterialDirectory 191
SVE_defaultObjectDirectory 191
SVE_defaultTextureDirectory 191
SVE_defaultTextureMap 192
SVE_defaultTextureSwap 72, 151
SVE_defaultWorldDirectory 191
SVE_defineObjectTextureList 71, 151
SVE_defineObjectTextures 71, 151
SVE_deleteObject 54, 138
SVE_disableBackfaceCulling 167
SVE_done 18, 133
SVE_drawBoundaries 73, 159
SVE_drawTrackerExtent 166
SVE_emptyGeometry 143
SVE_enableBackfaceCulling 167
SVE_endPrimitive 67, 147
SVE_endPrimitiveFace 68, 146
SVE_endPrimitiveLine 68, 146
SVE_enterEvent 161
SVE_facePtr 208
SVE_facesRead 190
SVE_facesRendered 189
SVE_far 190
SVE_findAllObjects 56, 136
SVE_findGeometryInRepository 71, 144
SVE_findObject 56, 136
SVE_findObjectInRepository 52, 136
SVE_findPollingDevice 90, 171
SVE_findWorldObject 58, 136
SVE_fly 49, 164
SVE_flyInObject 49, 164

SVE_flyInObjectWithDirection 49, 165
SVE_flyWithDirection 49, 164
SVE_FPSlowerLimit 191
SVE_FPSupperLimit 191
SVE_freeGeometry 144
SVE_geometryEntry 200
SVE_geometryRepository 189
SVE_getAnimationCallbacks 43, 153
SVE_getColorMaterial 74, 149
SVE_getConfig 23, 167
SVE_getCullingFunctions 49, 163
SVE_getCursorObject 61, 170
SVE_getCursorPosition 61, 170
SVE_getEventCallback 36, 161
SVE_getFirstObject 56, 154
SVE_getFrameCallback 46, 162
SVE_getFrameEndCallbacks 46, 163
SVE_getHMDObject 61, 170
SVE_getHMDPosition 61, 170
SVE_getInsetParams 219
SVE_getMaterialByIndex 74, 149
SVE_getMaterialByName 74, 149
SVE_getMatrixDist 169
SVE_getMouseState 161
SVE_getNearestPoint 137
SVE_getObjectBoundaries 73, 155
SVE_getObjectCenter 155
SVE_getObjectFrameCallback 46, 162
SVE_getObjectVisibility 49, 163
SVE_getPalmObject 84, 173
SVE_getPrimitiveBoundaries 73, 155
SVE_getPrimitiveCopy 70, 147
SVE_getPrimitiveExtents 159
SVE_getRelativeMatrix 169
SVE_getTextExtent 69, 147
SVE_getViewingAndPerspectiveMatrix 166
SVE_getWidgetData 97, 160
SVE_getWorldMatrix 59, 136
SVE_getWorldState 33, 167
SVE_GLOVE 14, 24, 27
SVE_gloveData 212
SVE_GOURAUD 14, 23, 25
SVE_highlightObject 40, 141
SVE_HMD 13, 20, 24, 26
SVE_HMDMONO 24, 26
SVE_HMDSTEREO 24, 26

SVE_hostname 189
SVE_init 13, 23, 133
SVE_initAnimation 41, 152
SVE_initFilename 191
SVE_initGeometryChange 66, 144
SVE_initGlove 83, 172
SVE_initTracker 83, 171
SVE_INSET 28
SVE_insetReset 219
SVE_invertMatrix 169
SVE_LIGHTING 23, 25
SVE_lightRenderPtr 195
SVE_linesRendered 189
SVE_LIT_GOURAUD 24, 25
SVE_loadObject 52, 135
SVE_loadObjects 62, 134
SVE_loadWorld 14, 61, 134
SVE_makeWidget 96, 160
SVE_material definition 74
SVE_matrixHitObject 156
SVE_matrixHitObjectPrimitives 156
SVE_minX 188
SVE_minY 188
SVE_moveBy 64, 140
SVE_moveObject 63, 139
SVE_moveTo 64, 140
SVE_moveVertex 140
SVE_multipleTexturesPtr 204
SVE_near 190
SVE_NOAUDIO 24, 27
SVE_NODISPLAY 24, 28
SVE_NORMAL 13, 23, 24
SVE_object 51, 199
SVE_object field
  boundaries 201
  children 51, 203
  cullable 51, 202
  facingViewer 52, 203
  facingViewerUpright 51, 203
  frameCallback 203
  geometry 200
  hascolor 201
  hasVisibleSphere 202
  highlight 51, 200
  highlightGeometry 201
  materialIndex 202

name 51, 200
parent 51, 200
position 51, 201
selectable 51, 201
update 202
UserPtr 52, 204
visible 51, 201
visibleSphere 202
widgetData 203
worldPosition 201
SVE_OBJECT_HIGHLIGHT 27
SVE_OBJECT_SELECTION 27
SVE_objectBoundsCollide 73, 157
SVE_objectBoundsInBounds 73, 157
SVE_objectCull 164
SVE_objectMatrixHit 155
SVE_objectPointHit 73, 156
SVE_objectRepository 189
SVE_objectVisible 136, 219
SVE_pdExtX 190
SVE_pdExtY 190
SVE_pickObject 157
SVE_pointerObjectFile 191
SVE_pointerObjectName 191
SVE_pointHitObject 73, 156
SVE_pointHitObjectPrimitives 73, 157
SVE_pointPtr 207
SVE_pollFunction 89
SVE_PREDISTORT 24, 27
SVE_primitiveHighlightMaterial 68, 146
SVE_primitiveLineWidth 147
SVE_primitiveMaterial 68, 146
SVE_primitiveNormal 69, 145
SVE_primitivePoint 70, 144
SVE_primitivePointWithColor 145
SVE_primitiveTexCoord 69, 145
SVE_primitiveText 68, 146
SVE_primitiveTextPosition 68, 147
SVE_primitiveTextRotation 68, 147
SVE_primitiveTextScale 68, 146
SVE_primitiveVertex 68, 146
SVE_primitiveVertexIndexes 70, 146
SVE_printDebug 176
SVE_printError 175
SVE_printFileError 176
SVE_printMatrix 169

SVE_printObjectList 58, 137
SVE_printVerbose 175
SVE_RBSTEREO 24, 26
SVE_readGestureFile 85, 172
SVE_readHandFile 85, 172
SVE_reCalculateObjectBoundaries 159
SVE_reCalculateParentBoundaries 159
SVE_reCalculateWorldMatrix 64, 142
SVE_recognizeGestures 87, 173
SVE_registerCallback 16, 36, 160
SVE_registerWidgetType 91, 159
SVE_removeAllAnimationCallbacks 16, 43, 153
SVE_removeAllCallbacks 16, 36, 160
SVE_removeAllCullingFunctions 49, 163
SVE_removeAllFrameCallbacks 18, 46, 161
SVE_removeAllFrameEndCallbacks 46, 163
SVE_removeAllObjectFrameCallbacks 46, 162
SVE_removeAnimationCallback 16, 43, 153
SVE_removeCallback 16, 36, 160
SVE_removeCullingFunction 49, 163
SVE_removeFirstObject 58, 154
SVE_removeFrameCallback 18, 46, 161
SVE_removeFrameEndCallback 46, 162
SVE_removeFromObjectList 58
SVE_removeObject 54, 153
SVE_removeObjectEntry 54, 154
SVE_removeObjectFrameCallback 46, 162
SVE_removeObjectGeometry 67, 142
SVE_renderNow 48, 166
SVE_renderNowWithFrameCallbacks 48, 166
SVE_renderObject 47, 165
SVE_renderObjectList 47, 165
SVE_renderWorld 47, 165
SVE_ResetCallbacks 160
SVE_resetGestureList 87, 173
SVE_RIGHTEYE 24, 26
SVE_rotateObject 63, 139
SVE_rotateWRT 64, 140
SVE_saveCurrentGesture 87, 173
SVE_saveHandFile 85, 172
SVE_saveObject 52, 135
SVE_saveObjects 62, 135
SVE_saveWorld 62, 135
SVE_scaleObject 63, 139
SVE_scaleWRT 64, 140
SVE_SELECT 24, 27

SVE_selectObject 40, 141
SVE_selectorObjectFile 191
SVE_selectorObjectName 191
SVE_serverDirectory 192
SVE_setAnimationFunc 42, 152
SVE_setAnimationVar 152
SVE_setBackgroundColor 167
SVE_setBoxBoundaries 158
SVE_setDefaultAnimationFunc 42, 152
SVE_setDefaultMaterialDirectory 32, 168
SVE_setDefaultObjectDirectory 32, 168
SVE_setDefaultTextureDirectory 32, 168
SVE_setDefaultWorldDirectory 32, 168
SVE_setFieldOfView 168
SVE_setFlightSpeed 50, 168
SVE_setFrameCallback 17, 46, 161
SVE_setFrameEndCallback 46, 162
SVE_setInitFilename 28, 134
SVE_setMaterial3Color 75, 151
SVE_setMaterial4Color 75, 151
SVE_setMaterialAlpha 74, 150
SVE_setMaterialAmbient 74, 150
SVE_setMaterialDiffuse 74, 150
SVE_setMaterialEmission 74, 150
SVE_setMaterialShininess 74, 150
SVE_setMaterialSpecular 74, 150
SVE_setMaterialTexture 75, 150
SVE_setNewObjectPosition 63, 138
SVE_setObjectAnimation 41, 152
SVE_setObjectFrameCallback 46, 162
SVE_setObjectHighlightMaterial 72, 142
SVE_setObjectMaterial 72, 142
SVE_setSelectable 141
SVE_setSphereBoundaries 158
SVE_setTextureSwapCallback 72, 151
SVE_setUserDefinedData 42, 152
SVE_setViewPlaneExtents 168
SVE_setVisibility 140
SVE_setWorldState 33, 167
SVE_showFrameRate 192
SVE_sizeX 188
SVE_sizeY 188
SVE_SPATIALSOUND 24, 28
SVE_state 15, 33, 193
SVE_state field
  animationCallback 198

beginTime 198
config 197
cullingFunction 199
currentEye 196
cursorObject 197
eventType 197
eventVal 197
flightSpeed 197
framesPerSecond 199
frameTime 198
hmdObject 196
lastFrameTime 198
lastMaterial 195
lightList 194
materialList 195
network 198
ntscOn 197
objectTree 194
origin 197
originObject 196
perspectiveMatrix 196
pollingDevices 195
programName 194
timeZone 198
userFrameCallback 198
userFrameEndCallback 198
userObject 196
viewingMatrix 196
viewingObject 195
viewingObject2 195
viewPlaneObject 197
windowData 194
windowType 194

SVE_status 16
SVE_STEREO 14, 24, 26
SVE_stopEventLoop 15, 133
SVE_textDashDistance 190
SVE_textInvisibleDistance 190
SVE_TEXTURE_DEFAULT 75
SVE_TEXTURE_INTENSITY 75
SVE_TEXTURE_INTENSITY_ALPHA 75
SVE_TEXTURE_RGB 75
SVE_TEXTURE_RGB_ALPHA 75
SVE_TEXTURE_RGB_ALPHA_LIGHTING 75
SVE_TEXTURE_RGB_LIGHTING 75
SVE_textureCoordPtr 208

SVE_textureMemory 189
SVE_TEXTURES 14, 24, 25
SVE_texturesRead 189
SVE_TRACKER 24, 26
SVE_trackerExists 171
SVE_translateObjectGlobal 63, 139
SVE_translateWRT 64, 139
SVE_turnOffLight 149
SVE_turnOffObjectLights 141
SVE_turnOnLight 149
SVE_turnOnObjectLights 141
SVE_updateAllObjects 143
SVE_updateObjectPrimitives 143
SVE_updateTracker 83, 171
SVE_verbose 192
SVE_vertexPtr 208
SVE_viewPlaneMaxX 188
SVE_viewPlaneMaxY 188
SVE_viewPlaneMinX 188
SVE_viewPlaneMinY 188
SVE_viewPlanePosition 188
SVE_viewPlaneRotation 188
SVE_vofY 188
SVE_widgetData 210
SVE_window 194
SVE_WIREFRAME 24, 25
SVE_worldState 190
SVE_worldStruct 195

T
TEXT 68
TRACKER_SERVER_MACHINE 192
TRK_changeHemi 217
TRK_close 217
TRK_getButtons 216
TRK_getMatrix 216
TRK_getTypeId 217
TRK_getTypeName 217
TRK_open 216
trkDevice 217

U
updateSortedList 180
USER 60

V

V_add 186
V_copy 186
V_cross 186
V_dot 186
V_len 185
V_move 186
V_move4 186
V_mult 186
V_neg 185
V_norm 185
V_scalar 185
V_sqrLen 185
V_sub 186
V_vector 185
V_zero 185