THE SIMPLE VIRTUAL ENVIRONMENT LIBRARY

User's Guide

Version 1.2

Jouke C. Verlinden, Drew Kessler, Larry Hodges

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

Graphics, Visualization and Usability Center,

Georgia Institute of Technology, USA.

ABSTRACT

The *Simple Virtual Environments* C library (SVE) provides basic functions to create virtual reality applications on Silicon Graphics workstations, including load/save/ render routines of (hierarchical grouped) objects and an event-callback mechanism. It also provides several default callback routines to minimize the programming effort needed for an average VR application (like walkthroughs). The functionality of the current version is rather moderate, but is easy to use and extendable to personal needs.

June 28, 1993

Table of Contents

1. INTRODUCTION:

We, at the Graphics, Visualization and Usability center, have recently purchased special hardware to build a general-purpose virtual reality platform. Current equipment includes a Silicon Graphics Indigo Elan, a Silicon Graphics Reality Engine, a Virtual Research Helmet Mounted Display, a dual-receiver Ascension Bird 3D tracking system and a Virtual Research Cyberglove. Although it is possible to write virtual reality applications solely using the existing Silicon Graphics libraries and the examples of the cyberglove software, we felt the need for a comprehensive and small VR-library; our experience with similar Virtual Reality systems (Jouke at Delft, Drew at the University of Virginia) was that most applications use the same routines to render objects, to load and save worlds etc. Instead of cut-copying all the time, we wanted to have a kernel on top of the existing libraries.

We started this project in November 4th, 1992. Our objective was to create a library that provides easy mechanisms to create virtual environments and has sufficient flexibility to implement non-trivial applications. Everyone is encouraged to use the library, to give us feedback, and even to make modifications. We tried to make the system as device-independent as possible, it should be easy to add other tracker systems etc. And if no special devices are available, the keyboard and mouse are used to interact with the environment. The world/object ASCII file-formats allow easy conversion from/to wavefront, rend386, CADKEY and other polygon-based Virtual Reality libraries. We encourage you all to write such convertors and to make these publicly available. A major drawback on compatibility is the dependence on the Silicon Graphics graphics language (gl) to render the virtual scenes. However, Silicon Graphics has announced OpenGL to be available this year, enabling Silicon Graphics applications to run on SUN/HP/DEC workstations and PC's with Windows/NT.

This document is intended to be a comprehensive description of 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 33) of this document.

1.1. Example Application

Before discussing the SVE-basics, we present you a simple application:

Source i: "hello world" example

```
/* example1 (sve module)

   reads an object file and its primitives and displays it in a window
   on the bottom left corner of the screen. The standard SVE key commands
   are recognized: 'q' to quit'; 't' to switch to NTSC; 'x', 'X', 'y', 'Y', 'z',
   and 'Z' to move around in the world.

*/

#include "sve.h"

main(int argc, char *argv[])
{
   SVE_config config = SVE_NORMAL; /* No trackers, gloves, or other extras */

   printf("Starting application\n");
   SVE_init("Example1 (sve)", config);

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

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

```
    SVE_beginEventLoop();

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

The 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 on it. Neither the trackers nor the glove are used. The standard SVE key commands are recognized by this application. Just move the cursor 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	Quit application.
t	Toggle between NTSC (Head mounted display) and display on the monitor.
x and X, or left/right arrow keys	Move viewpoint right and left.
y and Y, or up/down arrow keys	Move viewpoint up and down.
z and Z	Move viewpoint backword and forward.
Left mouse button and drag.	Rotate the viewpoint

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

Figure 1. snapshot of the first example.

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

Source ii: the world description file *hello_world.world*

```
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
-0.5 0.5 -5 1
other attributes: 0
number of children: 0
```

The geometry of the objects is defined in separate files, so objects can be used more than once:

Source iii: the object geometry file *hello_world.object*

```
Simple Virtual Primitive File Format version 1.0
number of components: 2

component 1 type: polyhedron
Data of component 1:
# 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
# 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 1 5 4
30 0 0 4 1 2 6 5
30 0 0 4 2 3 7 6
30 0 0 4 3 0 4 7
30 0 0 4 4 5 6 7

component 2 type: text
Data of component 2:
transformation matrix:
0.4 0 0 0
0 0.4 0 0
0 0 0.3 0
0.1 0.6 0.36 1
# of lines:
1
Hello World!
```

1.2. On-line Help

On-line help is available through man pages that can be found in *~vrgroup/sve/v1.2/man*. If this directory is included in your MANPATH environment variable, you will be able to obtain a man page on the SVE functions with the *man* utility.

2. SVE BASICS

The control flow of a virtual environment, actually any interactive application can be characterized by the following diagram:

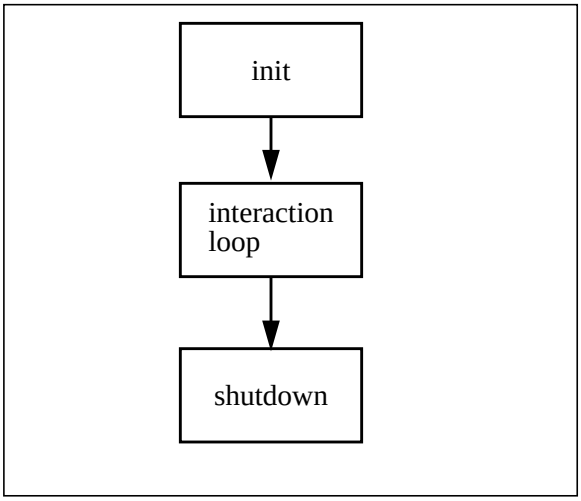


Figure 2. Global control flow.

This flow is used as a skeleton in our library: the library provides functions for initializing the VR application, processing the input, giving feedback, and to shut 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)

This function should be the first SVE-call, it allocates and initializes several data structures and opens a graphics window in the lower left corner of the screen[1]. The program name will appear in the title bar of that window, the configuration parameter specifies which graphics modes and hardware devices are used. Each additional input/output device is represented by an integer (defined in *sve.h*), the combination of multiple devices and preferences is defined by combining the integers with the binary "or" operation. *SVE_init* will call the initialization routines of the hardware devices, and will set callback-routines that define the interaction to a default use of the configuration. At the moment, we have the following configurations:

- *SVE_NORMAL*: The standard SGI configuration (represented by *NULL*). It will not use special hardware (as the glove or the tracker system), just the regular input devices (keyboard/mouse etc.).
- *SVE_HMD*: Use one bird tracker to change the viewpoint, the data of the two trackers will be saved in the global state-structure *SVE_worldState* (The Nuts and Bolts chapter, page 12, for a description of this data structure).
- *SVE_GLOVE*: the CyberGlove will be used, a graphical object will show the movements of the glove in the virtual screen and simple gesture recognizing can be enabled.
- *SVE_GOURAUD*: the world will be rendered with gouraud shading (flat shading is used by default, see gl manual chapter 9-4).

1. for NTSC-conversion with help of the VIDI/O box

- *SVE_TEXTURES*: textured polygons will be rendered with their textures (to insure a reasonable performance this flag should only be set when running on the Reality Engine, these textured polygons are rendered as ordinary ones by default).
- *SVE_NETWORKSLAVE*: expects input events to come from a remote machine. When the flag *SVE_HMD* is also set, both tracker matrices will be retrieved from the remote site as well (typically used to link the Bird tracking system at buckhead.gvu.gatech.edu with the Reality Engine). This requires a tracker/event server to run on another machine (see the Nuts and Bolts Chapter for more about this configuration).
- *SVE_SPATIALSOUND*: will initialize a network link with the spatialization host (nagel.cc.gatech.edu), and enables the function *SVE_attachSoundToObject(object)*. This facility is only in its preliminary stage[1]

New configurations should be added when other devices and graphics modes become available. It is possible to have different sets of interaction with one identical hardware configuration, e.g. there are several ways to navigate with the 3D tracker devices (flying in the direction you look at, flying in the direction you point at with the cursor, the “magic carpet” metaphor etc.). In such cases, new configurations should be included to select the appropriate callback sets.

2.1.1 Loading an Environment.

boolean SVE_loadWorld(filename)

This function loads objects from a file into an internal data structure. This world will be rendered automatically during the interaction loop. The function returns FALSE when an error during loading occurs. The file formats are described in appendix B, page 37. There are two distinctive file types: a world description file and object files. For a particular world (which defines everything that the SVE system will render), there is one world description file and many object files. The world description file defines a list of objects (and their hierarchy, which objects are children of other objects, etc.), their position in the world, and which object file to use for each object. The object file defines an object, which consists of many different types of primitives (polygons, lines, text, textured polygons, etc.).

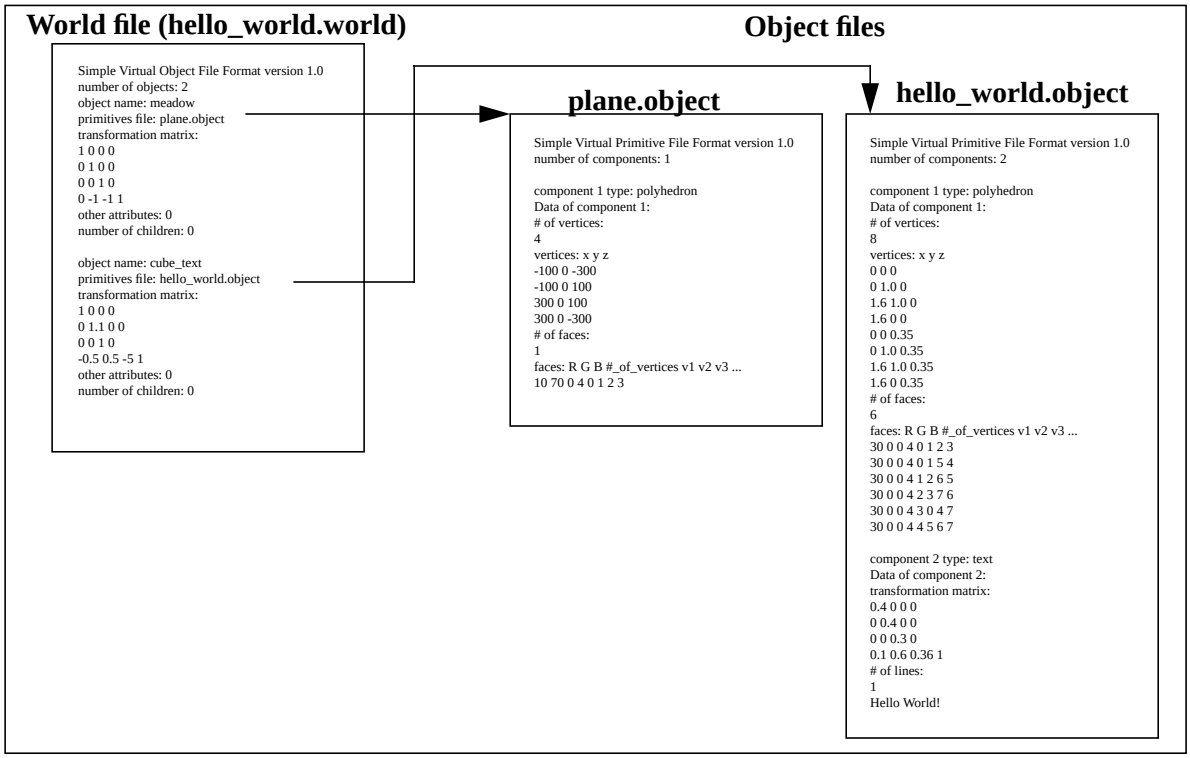


Figure 3. SVE data files

1. See “A First Experience with Spatial Audio in a Virtual Environment” (1993), David A. Burgess and Jouke C. Verlinden, Soon to be a GVU technical report.

2.2. The Interaction Loop.

If we look a bit more closely on the interaction loop, we get:

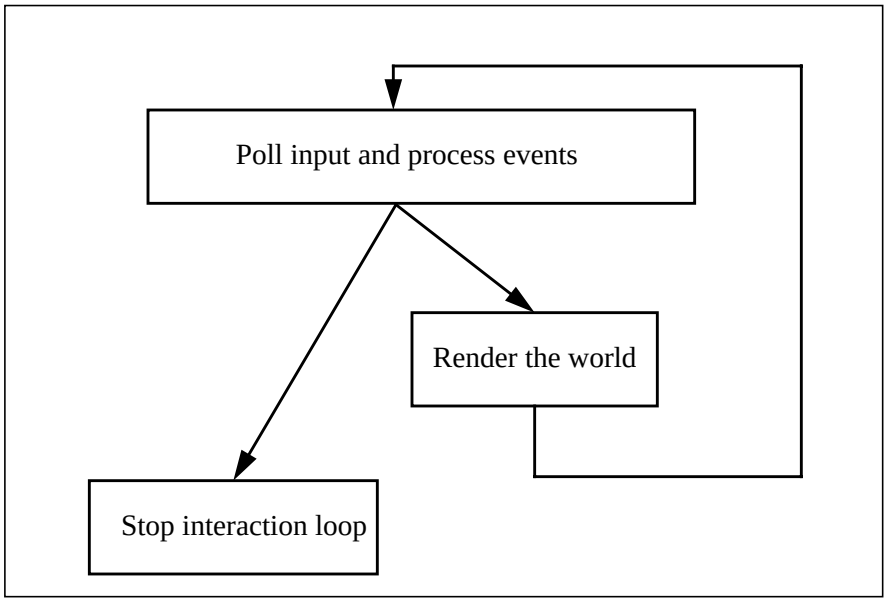


Figure 4. The interaction loop.

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(), SVE_stopEventLoop()* stops it.

2.2.1 Input Handlers

The data of the input devices is automatically put in a global *SVE_stateStruct* data structure. After this, the processing of the events is done by a notifier-callback mechanism (used on most of the 2D direct manipulation systems like SUNVIEW, etc.): the system notifies the application when an event occurs by calling an appropriate callback routine.

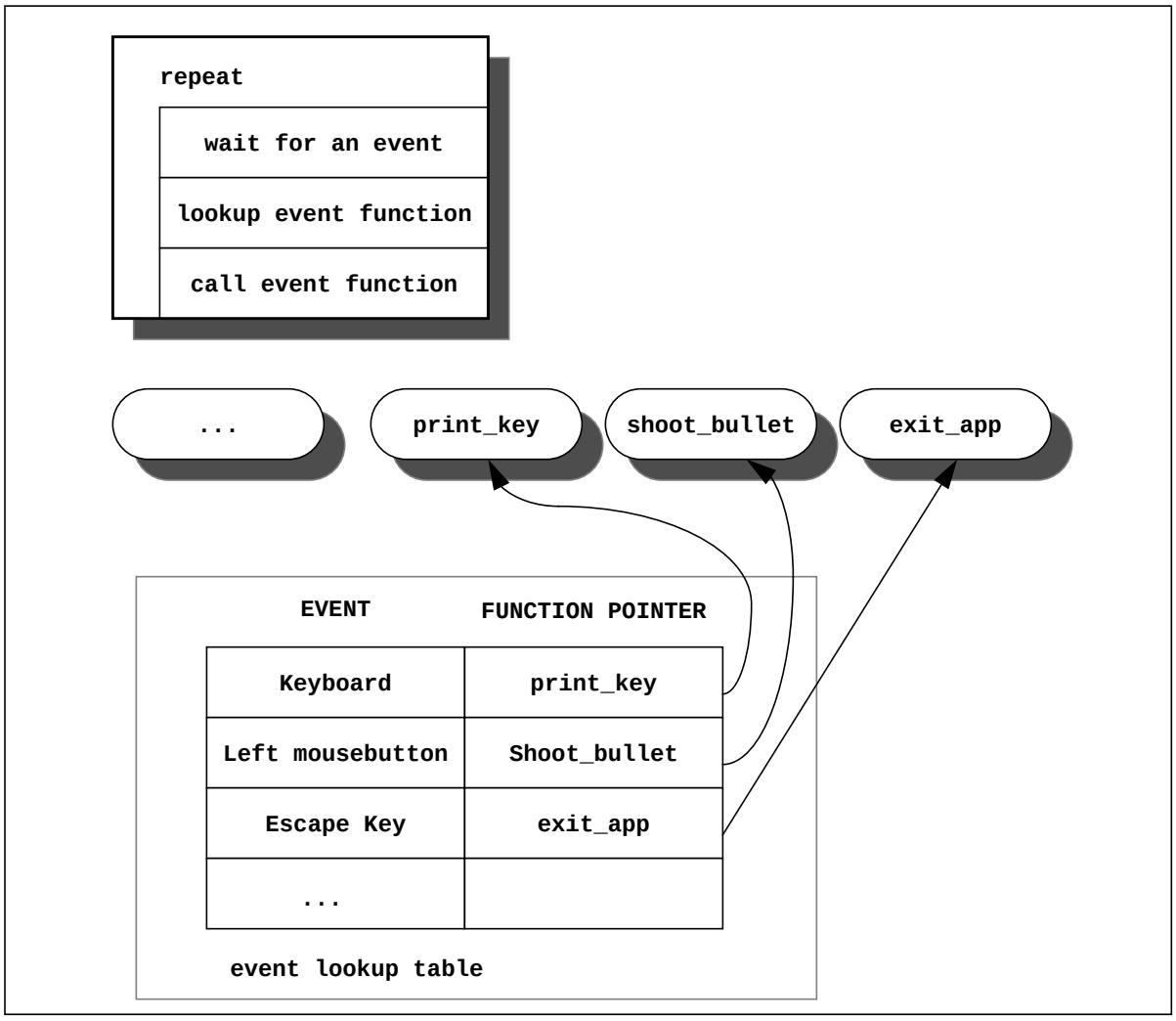


Figure 5. C**oncepts of a notifier mechanism and the event lookup table.**

Standard gl-events and additional ones (e.g. gestures) each have their own callback routines, selected by their event-identifiers (an integer constant). Callback procedures must have the prototype *void function(SVE_state state)*, they are set with the function *SVE_registerCallback(event, function)*, and disabled by calling *SVE_removeCallback(event)*. In the SVE-library, each event has only one callback routine, thus registering a new callback routine replaces an older one. As already mentioned, every event has its "default" callback routine depending on the configuration. No callback routines have to be registered if default behavior is sufficient. Default callback routines are disabled and enabled by *SVE_disableDefaultCallback(event)* and *SVE_enableDefaultCallback(event)* respectively. It is possible to use both user-defined and default callback routines for one event (e.g. to add keyboard controls). In that case the user-defined routine will be called prior to the default one.

2.2.2 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 the Nuts and Bolts section on Objects, page 17). 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 set with *SVE_setFrameCallback(function)*. Only one callback function can be set and *SVE_setFrameCallback(NULL)* will remove the entry.

2.3. Shutting Down.

void SVE_done(void)

This function deallocates the data structures and shuts down the I/O devices. It will automatically exit the application itself calling *exit(0)*, and thus should be the very last function call in the main routine.

2.4. Summary and Another Example

In short, SVE provides functions to initialize devices and rendering modes, functions to regulate the events and one function to stop the application. The interaction mechanism is based on event-callback functions, and objects are kept in an internal database.

The total control flow of a typical SVE-based application can be described as:

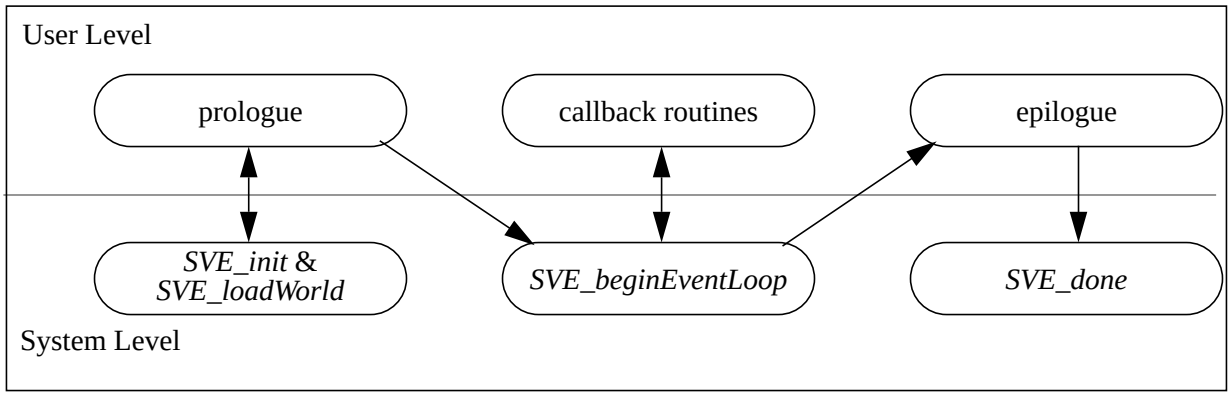


Figure 6. The control flow of the vr-application.

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

Source iv: example application 2

```
/* example2 (sve module)

   starts up with the bird trackers if 1st parameter is “t”,
   reads a world and registers an event-callback routine.
   The standard SVE key commands are recognized: ’q’ for quit; ’t’ to switch
   to NTSC; ’x’, ’X’, ’y’, ’Y’, ’z’, and ’Z’ to move around the world.

*/

#include “sve.h”


void printKey(SVE_state state) /* callback routine for KEYBD-events */
{
```

```
    printf("Key %c pressed\n",state->eventVal);
} /* printKey */


main(int argc, char *argv[])
{
    SVE_config config = SVE_NORMAL; /* Default configuration. No trackers. */

    printf("Starting application\n");

    /* Check command line for 't' */
    if ((argc > 1) && (strcmp(argv[1],"t") == 0))
    config = config | SVE_HMD; /* Add trackers to SVE configuration. */

    SVE_init("Example2 (sve)", config);

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

    printf("Registering an input callback\n");
    SVE_registerCallback(KEYBD, printKey);

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

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

This application presents the same virtual environment as the first one and also has the same controls to navigate. If the application is started with a 't' as parameter, the bird tracker system will be used and the viewpoint position will be updated with the position and orientation of bird receiver 1. This is how the command line would look to do this:

```
example2 t
```

Furthermore, one callback function is registered for KEYBD-events, which echoes the keys pressed to *stdout*.

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

3. NUTS AND BOLTS.

This chapter presents the SVE library in detail including the state model, event handling and object definition. The sections should provide sufficient information for being able to read, write and understand SVE-based applications.

3.1. State Structure/info.

During the execution of an application, the library keeps its variables in one large C-structure *SVE_worldState*. Every piece of information about the current state is contained in it, including the current configuration and samples of the hardware, rendering modes, the object database etc.

```
typedef struct SVE_stateStruct {
   char             *programName;/* a string for window title */
   SVE_objectList   *objectTree;/* World database */
   Matrix           viewingMatrix;/* the viewpoint orientation */
   Matrix           hmd;   /* the original non-inverted hmd sample */
   SVE_gloveData    *glove; /* glove related samples */
   SVE_point        origin; /* tracker to world coordinates vector */
   SVE_eventType    event; /* a gl event */
   short            eventVal;/* a gl event value */
   boolean          ntscOn;/* internal value for video mode */
   int              config;/* the config field */
   float            flightSpeed;/* sometimes used */
   boolean          checkForGesture;/* gesture recognition is called if TRUE */
   SVE_gestureList  *gestures;/* list with gestures for recognition */
} SVE_stateStruct;

typedef SVE_stateStruct *SVE_state;
```

Most fields in the state structure are intended to be a source of information (as the tracker information). The fields that are printed in italics also allow altering (Appendix C, page 45, presents a detailed description of this structure.) A pointer to this data structure is passed to the callback routines so that each can use and change the current state of the application.

3.2. More About Events & Callback Routines

Callback functions for event handling and drawing look similar, their prototypes are identical and each callback function has default and user defined classes. Yet their function is different, as can be predicted from this flow chart of the event loop:

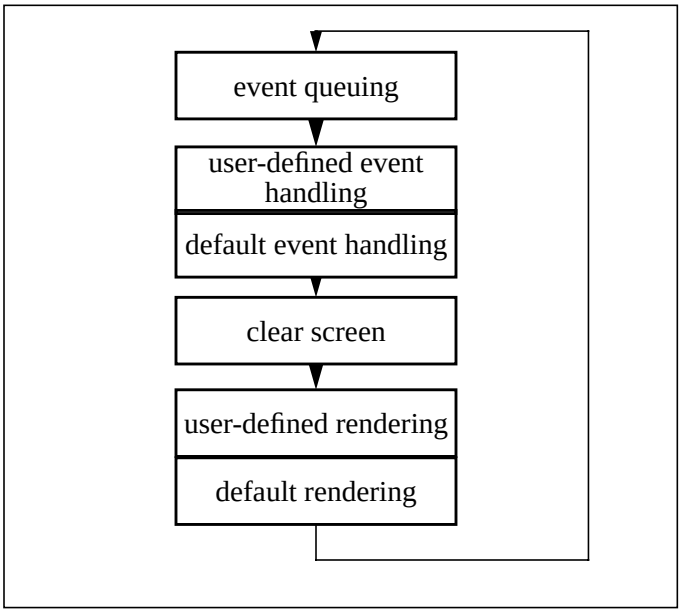


Figure 7. The event loop.

This section will focus on the use of events and the separation between input handling and drawing. A list of the default routines will be given, and yet another example application will illustrate the concepts.

3.2.1 Events

During each frame, the SVE event loop checks the gl event-queue. These events are read from the queue and put into the *SVE_worldState* fields *event* and *eventVal* (the first corresponds with the device type, the

second with the value), and the appropriate callback routines are called. More information about events and gl devices (including the SpaceBall and the Buttons & Dials boxes) can be found in chapter five of the gl-manual. Furthermore, the include file */usr/include/gl/device.h* has a full list of device constants. Only a few gl events are queued by default: KEYBD, LEFTMOUSE, MIDDLEMOUSE, RIGHTMOUSE, ESCKEY, BUT72 (left arrow), BUT73 (down arrow), BUT79 (right arrow) and BUT80 (up arrow). More events will be queued if the function *qdevice(deviceType)* is called before entering the event loop. For example, separate keys can be queued by opening devices for each of these:

```
qdevice(F1KEY);
SVE_registerCallback(F1KEY, help);

qdevice(HOMEKEY);
SVE_registerCallback(HOMEKEY, reset_position);
```

Just as mouse events, two events will be generated when these keys are entered: one for pressing (*eventVal* = 1) and one for releasing (*eventVal* = 0):

```
void help(SVE_state state)
{
   if(state->eventVal == 1)
      showHelp(); /* show helpmessage as long as the F1 key is pressed */
   else
      hideHelp(); /* the key is released, hide the message */
}
```

We have added some other events along with the standard gl ones. The most substantial additions are the “Mouse-drag” events, that are generated while a mousebutton is held down: LEFTMOUSEDRAG, MIDDLEMOUSEDRAG and RIGHTMOUSEDRAG.

Besides the elegancy of defining the user interface with events and callback procedures, the mechanism also proves to be a very effective message passer. Instead of directly calling other functions, subroutines can generate their own events with the standard gl function *qenter(DeviceType, value)*.

EXPERT INFORMATION:

Events are internally represented as numbers (gl manual chapter 5), the callback function-pointers are stored in arrays that use these events as indices. If you want to add other events to the system be aware that you may have to change the constant N_EVENTS (in events.c), which determines the size of these arrays (now set to 700). The numbers between 600 and 699 are free for use.

3.2.2 Event callback routines

As already mentioned in the first chapter, all callback procedures have the format *void function(SVE_state state)*. That means that these procedures cannot return values, but are able to change the current state directly.

The initial call *SVE_init()* will reset the event-callback routines depending on the configuration field in the state. The user-defined callback procedures are all linked to NULL, whereas the default callback procedures are set as displayed in the following table:

Table 2: Default event-callback procedures.

EVENT	SVE_NORMAL	SVE_HMD
KEYBD	handle_key	handle_key
REDRAW	redraw	redraw
ESCKEY	stop_app	stop_app

Table 2: Default event-callback procedures.

EVENT	SVE_NORMAL	SVE_HMD
LEFTMOUSE	start_orientation_change	reset_speed
LEFTMOUSEDRAG	none	fly
MIDDLEMOUSE	none	decelerate
MIDDLEMOUSEDRAG	none	none
RIGHTMOUSE	none	accelerate
RIGHMOUSEDRAG	none	none
BUT72	move_left	move_left
BUT73	move_down	move_down
BUT79	move_right	move_right
BUT80	move_up	move_up

Remark: Other configurations, such as *SVE_GLOVE*, *SVE_GOURAUD* and *SVE_TEXTURES*, have no effect on the interaction.

The default callback routines perform the following tasks:

handle_key -- Pressing ‘x’, ‘X’, ‘y’, ‘Y’, ‘z’ and ‘Z’ will change the world origin, ‘t’ toggles the video mode (normal <-> NTSC), ‘f’ fixes the camera position and ‘q’ quits the application.

start_orientation_change -- when the left mousebutton is pressed, this routine will link an event-callback routine to the LEFTMOUSEDRAG-event, that will change the camera orientation.

redraw -- This function calls reshapeviewport() which sets the viewport to the dimensions of the current graphics window (see the gl windowing and font library programming guide, section 1.3).

stop_app -- quits the event loop.

move_left, *move_down*, *move_right* and *move_up* -- will change the world origin relative to the current view.

reset_speed -- Sets the flying speed to a predefined value.

fly -- moves the origin, its direction is determined by the orientation of the current view, the speed by its initial value and an acceleration.

decelerate -- decreases the flying speed.

accelerate -- increases the flying speed.

EXPERT INFORMATION

The default callback routines are defined in *defaultcallbacks.c* (and their prototypes in *sve_local.h*), you can make your own ones and add these to the source. To link these new default routines during initialization, the function *SVE_resetCallbacks()* in *event.c* must be modified.

3.2.3 Frame-callback routines

Keep in mind that the event-callback procedures cannot draw directly, as the screen will be cleared just after processing the events; if user-defined drawing functions are needed, you must use the frame-callback function. This function must have exactly the same prototype format as the event-callback functions, *void*

function(SVE_state state), and is called every frame update just after the screen is cleared. Non-event based processes such as animations can only be updated with this frame-callback mechanism. For both event and frame callback routines, dynamic linking can be used to define complex interaction:

Source v: example application 3

```
/* example3 (sve module)

   Starts up with the bird trackers if 1st parameter is “t”,
   reads a world and registers an event-callback and a
   frame-callback routine.
   All of the standard SVE keys commands are recognized.
   In addition, the ’r’ key rotates the cube to the right, the ’l’ key
   rotates the cube to the left.

*/

#include “sve.h”

/* global variable object */
SVE_object globalObject;

/* two frame-callback routines, are invoked by pressing keys */
void rotate_right(SVE_state state)
{
   pushmatrix();
   loadmatrix(globalObject->position); /* put object matrix on the stack */
   rot(3, ’y’); /* apply rotation */
   getmatrix(globalObject->position); /* update the object matrix */
   popmatrix();
}

void rotate_left(SVE_state state)
{
   pushmatrix();
   loadmatrix(globalObject->position); /* put object matrix on the stack */
   rot(-3, ’y’); /* apply rotation */
   getmatrix(globalObject->position); /* update the object matrix */
   popmatrix();
}

/* callback routine for KEYBD-events */
void handleKey(SVE_state state)
{
   switch(state->eventVal) {
      case ’r’: SVE_setFrameCallback(rotate_right);
         break;
      case ’l’: SVE_setFrameCallback(rotate_left);
         break;
      case ’ ’: SVE_setFrameCallback(NULL);
   } /* switch */
} /* printKey */

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

   printf(“Starting application\n”);
```

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

    SVE_init("Example3 (sve)", config);

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

    globalObject = SVE_findWorldObject("cube_text");
    printf("Registering an input callback\n");
    SVE_registerCallback(KEYBD, handleKey);

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

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

This application is almost identical to the second example, the difference is that the cube will start to rotate when the ‘l’ or the ‘r’ key are pressed (‘l’ - left turns, ‘r’ - right). This rotation is stopped by entering a space.

3.3. Objects

One of the most important aspects of the library is the graphical representation of the virtual environment. An object database is managed by the library, and has functions to load, save and render itself. In this section, we will begin with discussing the high-level programming interface to this database (loading and saving the objects). After mentioning the current graphical primitives, we will focus on the object data structure and on manipulation of the virtual environment. Finally, object drawing routines and rendering modes are discussed.

3.3.1 Loading and saving objects

The object database is loaded by calling the function *SVE_loadWorld(<filename>)*. This function first tries to open the file in the current directory, if this file doesn’t exist it will search in the standard object-directory[1] as defined in *sve.h*. It will return the boolean FALSE if an error occurs. The ASCII files (see appendix B) are parsed and the world database is built on the fly. Although the parsing routines give clear warnings when syntax errors occur, some errors totally confuse the algorithm and generate segmentation faults. It is good practice to copy example files and to modify them rather than to create these from scratch.

The function *SVE_saveWorld(<filename>)* does the opposite. It saves the database in a worldfile, and the objects that are changed during execution of the program.[2] in their object files. Caution has to be taken, as existing files are replaced. In order to prevent losing well designed worlds and objects, copies should be preserved in a “save” directory.

The function *SVE_loadWorld()* replaces the existing world, to merge the worldfile with the existing one, you can you the call *SVE_addObjects()* (look in the appendix C for a complete description).

Furthermore, we have simple conversion routines to convert Wavefront objects to SVE-readable files (appendix B shows how).

---

1. Currently *~vrgroup/sve/v1.2/objects*
2. The library does not detect these changes by itself, it scans the object field *primitivesChanged*. Thus, the programmer is responsible for setting this boolean.

3.3.2 Primitives classes

Currently, three classes of primitives are supported: flat shaded polyhedrons, polylines and texts. Some of these were already used in the example files. Here is another example object file that incorporates all three:

Source vi: all_primitives.object

```
Simple Virtual Primitive File Format version 1.0
number of components: 4

component 1 type: polyhedron
Data of component 1:
# 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
# of faces:
6
faces: R G B #_of_vertices v1 v2 v3 ...
30 0 0 4 0 1 2 3
10 0 0 4 0 1 5 4
20 0 0 4 1 2 6 5
22 0 0 4 2 3 7 6
35 0 0 4 3 0 4 7
10 0 0 4 4 5 6 7

component 2 type: polyline
Data of component 2:
# 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
# of polylines:
6
polylines: R G B #_of_vertices v1 v2 v3 ...
100 100 100 4 0 1 2 3
100 100 100 4 0 1 5 4
100 100 100 4 1 2 6 5
100 100 100 4 2 3 7 6
100 100 100 4 3 0 4 7
100 100 100 4 4 5 6 7

component 3 type: text
Data of component 3:
transformation matrix:
0.4 0   0   0
```

```
0  0.40  0
0  0  0.3 0
0.1 1.5 0.3 6 1
# of lines:
1
***this is text***

component 4 type: textured_polyhedron
Data of component 4:
image file: testimage.rgb
repeat texture: FALSE
blending: TRUE
# of vertices:
4
vertices: x y z
-0.3 -1.2 -0.0 0 0
-0.3 -0.2 -0.0 0 1
 0.3 -0.2 -0.0 1 1
 0.3 -1.2 -0.0 1 0
# of faces:
1
faces: R G B #_of_vertices v1 v2 v3 ...
100 100 100 4 0 1 2 3
```

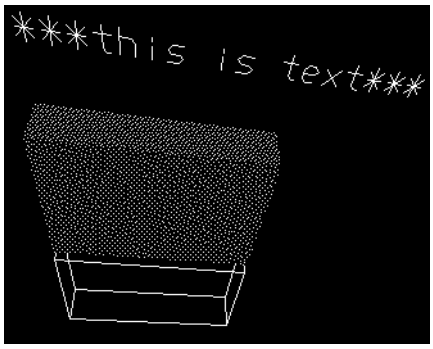


Figure 8. Snapshot of this object.

The appendix gives detailed information on file formats and data structures of these primitive types.

3.3.3 Data structure

In short, the environment is stored in a tree, each node representing an object. That means that the objects can be grouped hierarchically, for example, a table can be represented as a tabletop (parent) that has legs at each corner as children.

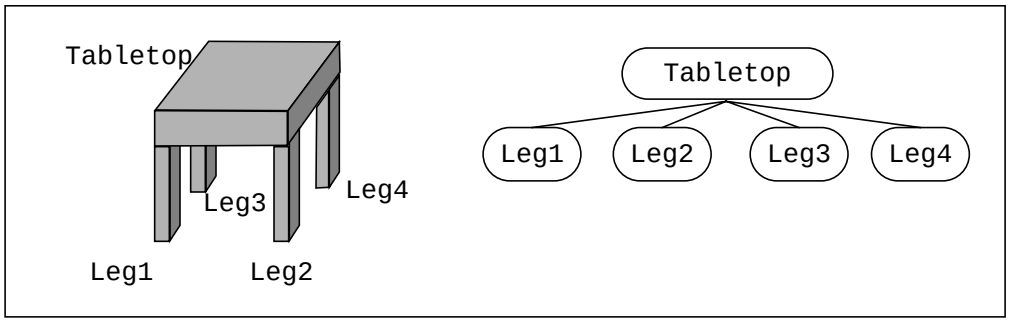


Figure 9. Hierarchical grouping.

This is the C-structure that defines an object:

```
typedef struct SVE_objectStruct {
 SVE_object parent;
 char *name;
 SVE_primitiveList *primitives;
 Matrix position;
 float color[4];/* red, green, blue and alpha (latter not used currently) */
 boolean visible, hascolor, selectable;
 boolean hasBoundaries, hasSphere; /* old vars (will be removed) */
 float radius; /* for boundary spheres */
 SVE_boundaries *boundaries;
 boolean hasVisibleSphere ; /* new vars for handling */
 float visibleSphere; /* level of detail */
 boolean primitivesChanged;
 char *primitivesFilename;
 Object glID;
 boolean needsUpdating;
/* the following attributes are specially added for hypertexts etc. */
 SVE_primitive *label; /* text-label for an object */
 int labelDrawMethod;/* attr. for pumping up performance in j-render */
 char *text;
 char *textfile;
 boolean textChanged;
 long class; /* classification of objects */
 boolean renderParentLink; /* currently not used */
 SVE_objectList *children;
} SVE_objectStruct;
```

(the complete list of data structures is presented in Appendix C, page 45)

The field *primitives* points to a linked list of primitive graphical representations (polyhedrons, lines, texts etc.) The object scale, position and orientation are determined by its transformation matrix *position*. Transformation matrices allow scaling, rotation, translation and other linear transformations (for information on matrices, we refer to a standard computer graphics book and section 7.4 of the gl-manual). Matrices are accumulated for hierarchal groups, so when the matrix of an object is modified, the same modification will be applied to all of its progeny, without updating their matrices. For example, if the tabletop object is rotated, its legs will rotate as well.

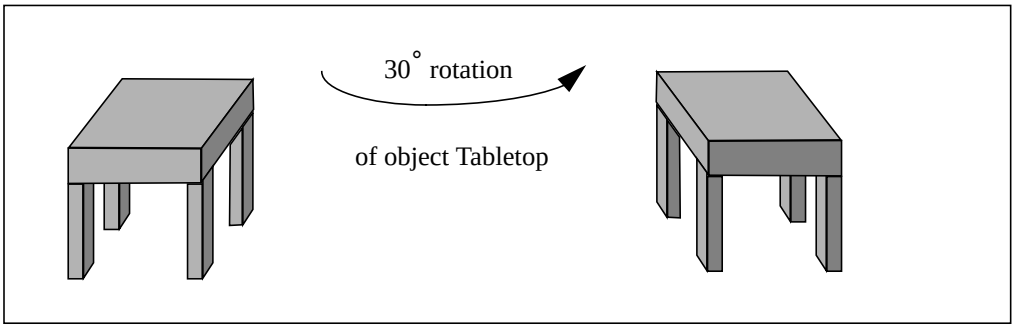


Figure 10. The Propagation of transformations.

The library contains several routines to interact with the world and object structures, including: *SVE_findWorldObject(), SVE_getWorldMatrix(), SVE_printObjectList(),SVE_createEmptyObject()* and *SVE_addChildToObject()*. They are listed in appendix C, section 2.2, page 55 (world/object utilities).

3.3.4 Rendering

Each primitive has its own rendering method, defined in the module obj-render.c. This is optimized using the gl object drawing mechanism, see chapter 16 of the gl manual. This means that the database is defined once within gl and from then on only ‘called’. If the geometry of an object is changed the object needs to be updated in the gl-database. The internal rendering routine will automatically take care of this, provided that the programmer changes the object field *needsUpdating* to TRUE whenever the geometry is changed. Object matrices are automatically updated each frame. If the object has a color assigned, that color will override primitives’ colors, and its children’s colors.

We now discuss some of the peculiarities of these rendering routines. The face normals for flat-shaded polygons are calculated using the right-handed coordinate system, so the direction of the normal depends on the order of the vertices within the polygon:

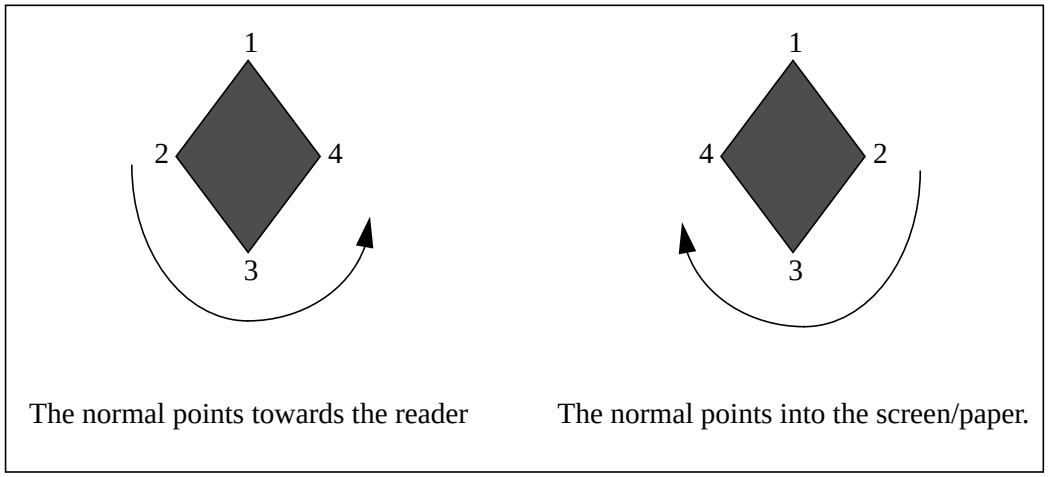


Figure 11. The order of the vertices and normals of the polygon.

As standard gl doesn’t provide true three-dimensional texts, text rendering is done by the module *3Dtext.c*. This module draws the characters as white polylines, and incorporates a simple level of detail management; because text rendering eats up performance, full text is only rendered when the viewpoint is close. From farther away only dashes are presented and eventually even these disappear.

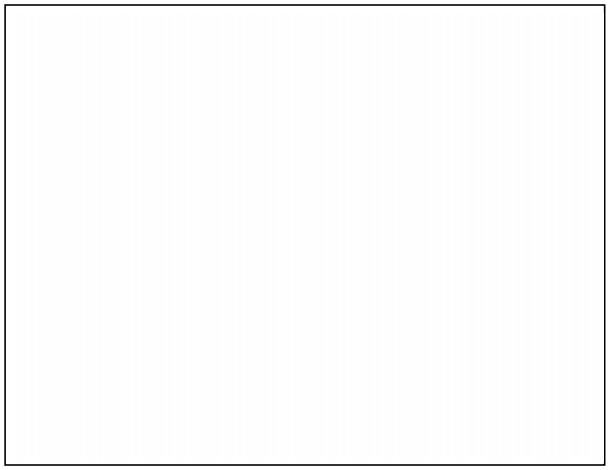


Figure 12. Texts with several levels of detail

Only an ambient light is used by default. Lights can be easily added to the environment using the standard gl commands *lmdef()* and *lmbind()* (see chapter 9 of the gl manual). This will decrease rendering speed, especially for big models (more that 10,000 polygons).

Source vii: example4

```
/* example4 (sve module)

   Reads an object file and its primitives,
   Adds a rotating light source to the scene.
   The standard SVE key commands are recognized.

*/

#include <gl.h>
#include <gl/device.h>
#include “sve.h”

#define MY_LIGHT 1
#define MY_MATERIAL 2
#define MY_MODEL 3

/* defines a spotlight (page 9-16 of gl manual) */
float spotLight[] = {
   LCOLOR,1.0 ,0.6 ,0.3,
   POSITION,20.0,50.0,-60.0,0.0,
   SPOTDIRECTION, -1.0, -0.4, 0.0,
   LMNULL
};

float lmodel[] = {
   ATTENUATION, 1, 0.4,
   ATTENUATION2, 1.0,
   AMBIENT, 0.4, 0.4, 0.4,
   LOCALVIEWER, 0.0,
   TWOSIDE, 0.0,
   LMNULL
};

/* initializes the light and the lighting model */
void setLights(void)
{
   /* define and bind the light, lighting model and material */
   lmdef(DEFLIGHT, MY_LIGHT, 0, spotLight);
   lmdef(DEFMATERIAL,MY_MATERIAL,0,NULL); /* default */
   lmdef(DEFLMODEL,MY_MODEL,0,lmodel);

   lmbind(LIGHT0, MY_LIGHT);
   lmbind(MATERIAL,MY_MATERIAL);
   lmbind(LMODEL,MY_MODEL);
}

/* rotates the light source each frame update */
void rotateLight(SVE_state state)
{
 static Matrix light={1,0,0,0,
                      0,1,0,0,
                      0,0,1,0,
                      0,0,0,1 };
   pushmatrix();
   loadmatrix(light);
   rot(6, ‘y’);
   rot(4, ‘x’);
```

```
    lmbind(LIGHT0, MY_LIGHT);
    getmatrix(light);
    popmatrix();
}

main()
{
    SVE_config config = SVE_NORMAL;

    printf("Starting application\n");
    SVE_init("Example4 (sve)-- Light!!", config);

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

    SVE_setFrameCallback(rotateLight);

    SVE_setBackgroundColor(10, 10, 30);

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

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

As the Elan can render gouraud shaded polygons, our next step will be to incorporate these into the system. Other possible extensions include: Phong-shaded polyhedrons, meshes, texture mapped polygons, multiple textfonts, nurbs, spheres etc. All these types should have their own rendering algorithms in *obj-render.c* and their own read/write functions in *obj-files.c*

3.4. Using Input Devices

3.4.1 Cursor and HMD objects

Both bird receivers have their shadow representation in the world tree: "*SVE cursor*" (receiver 2) and "*SVE HMD*" (receiver 1). These objects are empty, their position matrix however is constantly updated when the tracker system is active. They can be retrieved with the functions *SVE_getCursorObject()* and *SVE_getHMDObject()*, the matrices can be copied by calling *SVE_getCursorPosition(matrix)* and *SVE_getHMDPosition(matrix).*

Objects can be moved along with the trackers by adding them as a child. For example, a cube is attached to the first tracker (the HMD view) by:

```
SVE_object cube;
cube = SVE_findWorldObject("cube");

/* delete the old cube entry from the world tree */
SVE_removeObject("cube");

/* now add the object to the HMDObject */
SVE_addChildToObject(SVE_getHMDObject(), cube);
```

A permanent offset will occur, as the cube object is a child with a (predefined) transformation matrix. When the glove is active, a complex group of objects that constitute the graphical appearance of a hand is added to the "*SVE cursor*"; the next section describes how to work with the glove.

A more dynamic illustration of the use of the empty objects is showed in example 5:

Source viii: example of grabbing and releasing objects.

```
/* example5 (sve module)

   This example demonstrates 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 <device.h>
#include <math.h>
#include "sve.h"

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

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

SVE_object testObject(SVE_object obj, Matrix m)
{
   Matrix pos;
   float distance;

   if(obj->boundaries) {
      if (obj->boundaries->hasSphere) {
          SVE_getWorldMatrix(obj, pos);
          pos[3][0] += obj->boundaries->sphereOrigin[0];
          pos[3][1] += obj->boundaries->sphereOrigin[1];
          pos[3][2] += obj->boundaries->sphereOrigin[2];

          distance = SVE_getMatrixDist(pos, m);

          if (distance < obj->boundaries->sphereRadius)
             return(obj);
      }
   }

   return(NULL);
}


SVE_object testObjectList(SVE_objectList *list, Matrix m)
{
   SVE_objectList *objNode;
   SVE_object object;
```

```
    SVE_object result;

    for (objNode = list; (objNode); objNode = objNode->next)
    {
        if ((strcmp(objNode->o->name, "SVE HMD") != 0)
            && (strcmp(objNode->o->name, "SVE cursor") != 0))
        {
            if (objNode->o->children != NULL)
                if ((result = testObjectList(objNode->o->children, m)))
                    return(result);
            if ((object = testObject(objNode->o, m)))
                return(object);
        }
    }

    return(NULL);
}


void grab(SVE_state state)
{
    Matrix pos, pos1;
    SVE_object object;
    SVE_object pointer;
    SVE_object cursor;
    Matrix pointerPos;

    if(state->eventVal) {
        cursor = SVE_getCursorObject();
        pointer = SVE_findObject("pointer", cursor->children);
        SVE_getWorldMatrix(pointer, pointerPos);
        object = testObjectList(state->objectTree, pointerPos);

        if(object) {
            SVE_changeText(pointer, "Grabbed");
            current_object = object;
            current_parent = object->parent;

            SVE_getWorldMatrix(object, pos);
            SVE_getRelativeMatrix(pointerPos, pos, object->position);

            if(current_parent) {
                SVE_removeObject(object->name, &(current_parent->children));
            }
            else {
                SVE_removeObject(object->name, &(state->objectTree));
            }
            SVE_addChildToObject(object, pointer);
            SVE_registerCallback(LEFTMOUSE, release);
        } /* if */
    } /* if */
}


void release(SVE_state state)
{
    Matrix pos1, pos2;
    SVE_objectList objNode;
    SVE_object pointer;
```

```
    SVE_object cursor;

    if(state->eventVal == 0)
    {
        cursor = SVE_getCursorObject();
        pointer = SVE_findObject("pointer", cursor->children);
        SVE_changeText(pointer, "<=>");

        if(!current_parent) {
            /* object must be linked in state->objectTree */
            SVE_getWorldMatrix(current_object, current_object->position);
            SVE_removeObject(current_object->name, &(pointer->children));
            SVE_addToObjectList(current_object, &(state->objectTree));
            current_object->parent = NULL;
        } else {
            /* the object had a parent, put it back on its place */
            SVE_getWorldMatrix(current_parent, pos1);
            SVE_getWorldMatrix(current_object, pos2);
            SVE_getRelativeMatrix(pos1, pos2, current_object->position);
            SVE_removeObject(current_object->name, &(pointer->children));
            SVE_addToObjectList(current_object, &(current_parent->children));
            current_object->parent = current_parent;
        } /* if */
        current_object = NULL; /* reset globals */
        current_parent = NULL;
        SVE_registerCallback(LEFTMOUSE, grab);
    } /* if */
}


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

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

    SVE_init("example 5 (sve)", config);

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

    pointer = SVE_loadObject("pointer.obj", "pointer");
    SVE_addChildToObject(pointer, SVE_getCursorObject());

    if(config & SVE_HMD)
    {
        SVE_registerCallback(LEFTMOUSE, grab);
        SVE_disableDefaultCallback(LEFTMOUSE);
        SVE_disableDefaultCallback(LEFTMOUSEDRAG);
    }

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

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

3.4.2 Audio support

Auditory feedback is certainly an important and sometimes even necessary part of the user interface. Apart from augmenting engagement (as in computer games), short sounds can be used as auditory cues for events. This kind of feedback is especially useful while performing tasks with the Bird trackers, for example grabbing objects from a pile. Though another important characteristic of perceiving sounds, we currently do not have any support for spatialization.

Playing sounds is done by calling the application *playaiff* from the command line, for example by calling the following function in your application:

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

    sprintf(command, "playaiff %s &", filename)
    system(command);
}
```

Soundfiles are recorded with the *recordaiff* application. This application can sample sounds with several sample rates, durations, numbers of channels etc. The following function was used in the 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);
}
```

3.4.3 Spatial audio support

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).*

3.5. Glove handling

At this point, the SVE library supports a Cyberglove input device connected to serial port 1 or 2 (if port 1 is not available). The glove is initialized by OR-ing the SVE_GLOVE value with the configuration value used in *SVE_init*. Once the glove is initialized, it is polled once each time through the SVE system loop. The values of the sensors in the glove can be obtained by the functions provided by the Cyberglove interface (see "*include/cg_glove.h*"), or through the glove structure in the state structure (*state->glove->angle[finger][joint]*) (see Appendix C: Glove section). Also, through the functions *SVE_saveCurrentGesture*() and *SVE_recognizeGestures(TRUE),* gestures can be defined by example. Each time *SVE_saveCurrentGesture()* is called, the hand position at the time of the last polling of glove sensor

values is saved. After a *SVE_recognizedGestures(TRUE)* call is made, whenever that hand position is made (within an error given as range at the time the gesture is saved), a GESTURE event will be entered onto the GL event queue, with the identifying value given with the *SVE_saveCurrentGesture* call (priority). 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). 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 saved gesture.

The glove button is treated as a mouse button. Whenever the button changes value (pressed to not pressed, and vice versa), a GLOVEBUTTON event is placed on the GL event queue. The value of the event is the status of the button.

The following example demonstrates the use of the glove. Different gestures can be saved by pressing the ‘o’ key. A calibration routine is demonstrated with the *openHandCal* and *closedHandCal* function calls, which make Cyberglove calls to set the calibration values implicitly used by the Cyberglove interface routines (see Cyberglove documentation).

Source ix: glove_example

```
/* Glove example

 starts up with the bird trackers if 1st parameter is “t”,
 reads a world and registers an event-callback routine for recognizing
 glove gestures and the glove button. Gestures are saved every time the
 ‘o’ key is pressed.

 The glove can be calibrated to a hand by pressing ‘0’ when the hand is
 stretched out, and pressing ‘9’ when the hand is a closed fist.

 If the tracker isn’t used, the glove can be placed at the viewing origin
 by pressing the ‘g’ key. (You will want to back up by pressing ‘z’
 so that you can see the glove.)

*/

#include “sve.h”

int priority = 1;
#define GESTURE_RANGE 0.33

void gestureCallback(SVE_state state)
{
   if (state->eventVal != NULL_GESTURE)
      printf(“Gesture %d event recognized\n”, state->eventVal);
}

void gloveButtonCallback(SVE_state state)
{
   printf(“Glove button is now %d\n”, state->eventVal);
}

void openHandCal(SVE_state state)
{
   int i, j;

   for (i=INDX; i<FNGRS; i++)
      for (j=MPHL; j<ABDT; j++)
         CG_set_offset(i, j, CG_get_value(i, j));
}
```

```
void closedHandCal(SVE_state state)
{
   int i, j;

   for (i=INDX; i<FNGRS; i++)
      for (j=MPHL; j<ABDT; j++)
         CG_set_gain(i, j, (3.14/2)/(CG_get_value(i,j) - CG_get_offset(i, j)));
}

void setGloveStuff(SVE_state state)
{
   int allright;
   Matrix hmdPos;

   switch(state->eventVal)
   {
      case 'g':
         SVE_getHMDPosition(hmdPos);
         SVE_copyMatrix(hmdPos, state->cursorObject->position);
         break;
      case 'o':
         SVE_recognizeGestures(TRUE);
         if (priority<MAX_GESTURE) {
            SVE_saveCurrentGesture(state, priority, GESTURE_RANGE);
            priority++;
         }
         break;
      case '0':
         openHandCal(state);
         break;
      case '9':
         closedHandCal(state);
         break;
   } /* switch */
}

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

   printf("Starting application\n");

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

   SVE_init("Glove example", config);

   if(!SVE_loadWorld("axis.des"))
   {
      printf("error occurred during SVE_loadWorld, exiting \n");
      SVE_done();
} /* if */

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

   SVE_registerCallback(GESTURE, gestureCallback);
   SVE_registerCallback(GLOVEBUTTON, gloveButtonCallback);
```

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

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

3.6. Using the Reality Engine -- Networking

As the trackers are physically connected to buckhead, problems occur when the Reality Engine (or other SGI workstations) are being used. The SVE library currently has a limited facility to tackle this problem, using buckhead as a tracker server. By initializing the application with *SVE_NETWORKSLAVE*, it will wait on events that are send to a certain internet-address. These events are put on the ordinary gl event-queue and thus are processed as if they originated from the local machine. When the application is initialized with *SVE_HMD* as well, it will retrieve bird samples from over the network.

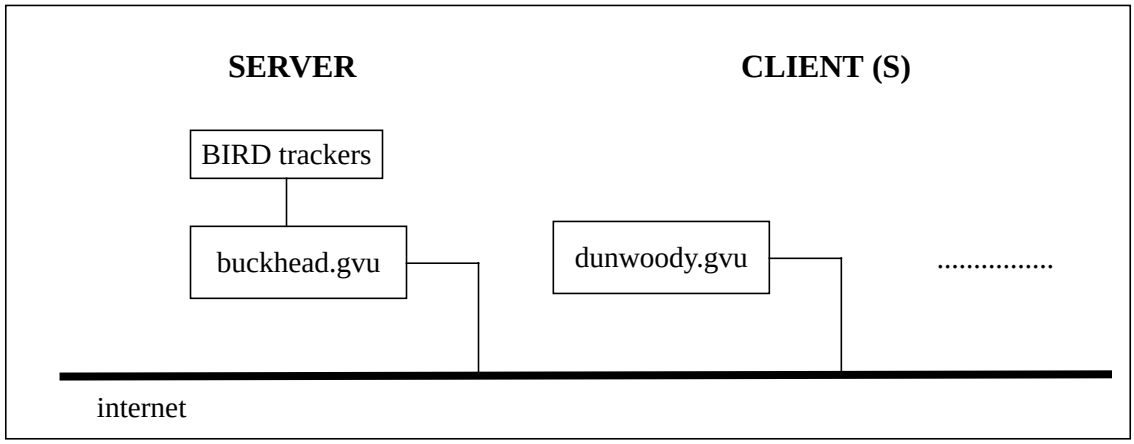


Figure 13. Using the tracker-server.

As we use the multi-point broadcast functions, there is no limit to the number of clients that make use of the information. This is the first step towards support for shared virtual environments.

EXPERT INFORMATION

A big disadvantage of the current implementation is that the information is passed using UDP sockets, a low level communications facility that does not guarantee the arrival of data. Thus events can get "lost" on the internet, which can be very harmful if the application completely changes its state when certain events happen. Future versions should be based on reliable TCP connections instead.

3.7. SVE Modules

3.7.1 Overview

Entirely meant for the curious, this chapter gives a global overview of the SVE library and its four modules. More detailed information is found in the sources themselves.

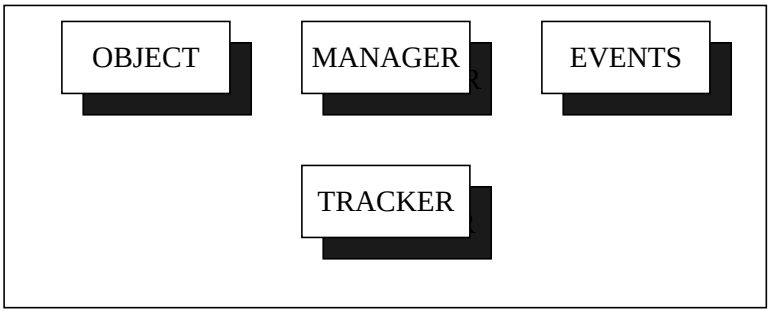


Figure 14. The modules.

MANAGER: The central module, supervises the I/O devices, keeps track of the objects to be drawn, calls the eventmanager etc. The source code to this module can be found in *manager.c*

OBJECTS: An extendable data structure with several methods (load/save/render). This module is subdivided into the files *obj-files.c, obj-render.c* and *object.c*.

EVENTS: A set of functions to poll the input devices, create events and a callback mechanism to process the events. The source code for these functions is in the files *event.c* and *defaultcallbacks.c*.

TRACKER: Provides basic functions to initialize, read and close the Bird three-dimensional trackers. This module is contained in the file *tracker.c*. (If another tracker system is used, this module should be rewritten/modified).

3.7.2 Control Flow

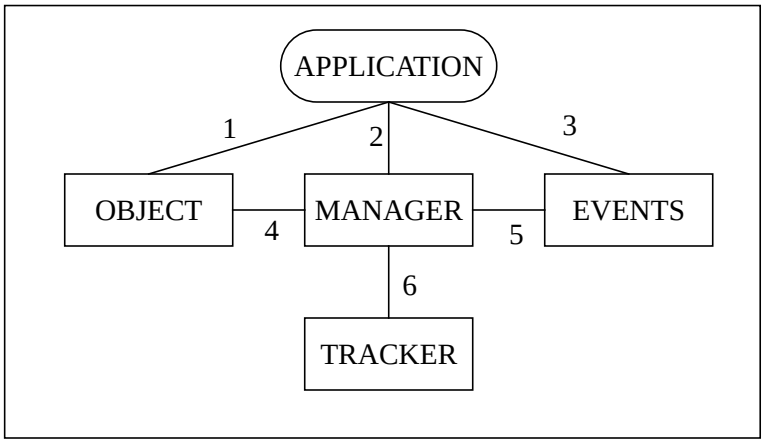


Figure 15. The control flow.

1) Functions to transform object and to get their attributes.

2) Initialize SVE, load/save a world, goto the event loop, stop the event loop, call the frame callback routine.

3) Register/enable/disable user defined or default callback routines. Call the callback routines.

4) Load/save/render objects.

5) initialize the event-callback mechanism, poll the input devices and process the events.

6) initialize the trackers (when available) and retrieve tracker data.

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.
- Stereoscopic displays.
- Spatial sound cues.
- Networking, 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 lots of people working on dynamics, simulation and animation; e.g. Augusto Op Den Bosch has already constructed a simulation system which works with the helmet mounted display).

APPENDICES
APPENDIX A: Starter Kit
The Simple Virtual Environment-Library Starter Kit, Version 1.2

Drew Kessler, Jouke Verlinden, Larry Hodges

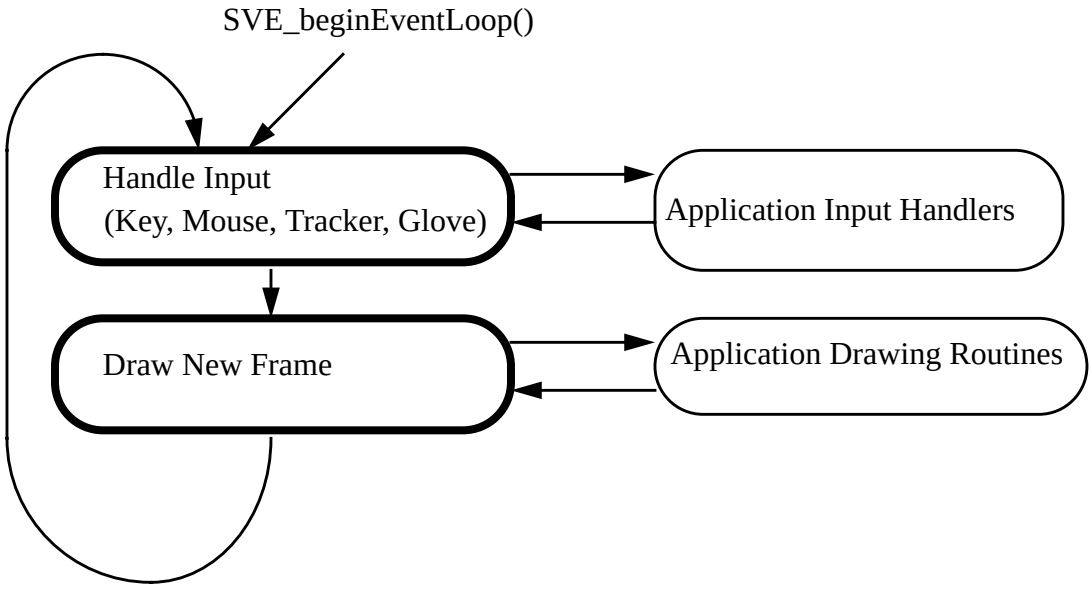
drew@cc.gatech.edu, jouke@cc.gatech.edu, hodges@cc.gatech.edu

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 which allows for easily creating simple Virtual Environment applications, and allows for extensive and straight-forward addition of functionality. What this means is that, using this library, you can create a simple Virtual World which has minimal interaction (a pure Architectural walkthrough, for example), and allows for adding features (such as allowing a user to move the kitchen sink) in a straight-forward and consistent manner.

General Overview

The SVE system works using the "don't call us...we'll call you" 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:



An Example

What follows is a very simple example application. An explanation follows it. The file is called "`example1.c`" and resides in the directory "`~vrgroup/sve/v1.2/examples`". If you wish to copy it and compile it, be sure to copy the `Makefile` in the `examples` 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 the directory "`~vrgroup/sve/v1.2/objects`". Thus, you need not copy those files unless you wish to modify them. A following section describes the file formats.

```
/* example1 (sve module)

 reads an object file and its primitives...

*/

#include “sve.h”

main()
{
   SVE_config config = SVE_NORMAL;

   printf(“Starting application\n”);
   SVE_init(“Example1 (sve)”, config);

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

   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 a Virtual World described in the file “`hello_world.world`” with the call `SVE_loadWorld`. The SVE system loop (described in the Overview above) is begun with the call `SVE_beginEventLoop`. If the users presses ‘q’ in the application window, or the program aborts for any reason, control returns to the application, and it makes the call `SVE_done` to shut down the SVE system.

You will notice that if 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 Y), pressing ‘z’ or ‘Z’ (shift-’z’) to change Z, and by pressing and dragging the mouse to rotate the scene. If you press ‘q’, the application shuts down.

Second Example

In this example, which is “`example3.c`” in the `examples` directory, we introduce the user specified input handler and frame renderer. These routines are called in addition to the SVE input handler and frame renderer. In this example the 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 cub will rotate to the left, and if the space bar is pressed, the cube will rotate to the left. If you wish to compile this program, you need to copy the file “`example3.c`” from the `examples` directory, and you need to modify the `Makefile` used in the above section so that the line that reads

```
PROGRAM = example1
```

now reads

```
PROGRAM = example3
```

The code follows.

```
/* example3 (sve module)

 starts up with the bird trackers if 1st parameter is “t”,
 reads a world and registers an event-callback and a
 frame-callback routine.

 */
#include <gl/device.h>
```

```
#include “sve.h”

/* global variable object */
SVE_object globalObject;

/* two frame-callback routines, are invoked by pressing keys */
void rotate_right(SVE_state state)
{
   pushmatrix();
   loadmatrix(globalObject->position); /* put object matrix on the stack */
   rot(3, ‘y’); /* apply rotation */
   getmatrix(globalObject->position); /* update the object matrix */
   popmatrix();
}

void rotate_left(SVE_state state)
{
   pushmatrix();
   loadmatrix(globalObject->position); /* put object matrix on the stack */
   rot(-3, ‘y’); /* apply rotation */
   getmatrix(globalObject->position); /* update the object matrix */
   popmatrix();
}

/* callback routine for KEYBD-events */
void handleKey(SVE_state state)
{
   switch(state->eventVal) {
      case ‘r’: SVE_setFrameCallback(rotate_right);
         break;
      case ‘l’: SVE_setFrameCallback(rotate_left);
         break;
      case ‘ ‘: SVE_setFrameCallback(NULL);
   } /* switch */
} /* printKey */

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

   printf(“Starting application\n”);

   if ((argc > 1) && (strcmp(argv[1],”t”) == 0))
      config = SVE_HMD;

   SVE_init(“Example3 (sve)”, config);

   if(!SVE_loadWorld(“hello_world.world”)) {
      printf(“error occurred during SVE_loadWorld, exiting \n”);
      SVE_done();
   } /* if */

   globalObject = SVE_findWorldObject(“cube_text”);
   printf(“Registering an input callback\n”);
   SVE_registerCallback(KEYBD, handleKey);

   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 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 it is still present in the object tree, and will be rendered in each frame.

We have also added 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 KEYBD as the first parameter.

When a ‘r’ or ‘l’ key is pressed, we have set the frame callback to a function which will rotate the cube for each frame. This is accomplished with a call to SVE_setFrameCallback. When the space bar is pressed, the frame callback function is set to NULL to indicate that there is no user frame callback. It is in the frame callback that the application can draw it own graphics which would be in the world coordinates. The user’s viewpoint is taken care of by the SVE system. You will need to use SGI GL function calls if you want to draw something in the scene this is not an object.

All callback functions with SVE need to be declared in the form:

```
void functionName(SVE_state state)
```

The SVE_state data structure contains the SVE state, including the definition of the world, the users viewpoint and orientation, and the most recent events (such as eventVal, as used above). See the User’s Guide for more details.

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. Since the HMD has a limited lifetime, if it is not being used it should be turned off (the switch is in the back where the cord exits the helmet). If the HMD is being used, the first tracker (with the black strip) should be attached to the front of the HMD.

Where to Find Additional Help

Manual pages for the man utility can be found in the directory “~vrgroup/sve/v1.2/man”. If that directory is included in your MANPATH environment variable, you will be able to obtain man pages on SVE and its functions. If you are using ksh, you will want to add the following line to your .profile start-up file:

```
export MANPATH=$MANPATH:~vrgroup/sve/v1.2/man
```

Additional documentation can be found in the directory “~vrgroup/sve/v1.2/doc”. In particular, the User’s Guide contains a complete description of the system with a function and date structure reference.

APPENDIX B: File Formats

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 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 geometrical information of the objects, which can be made up of many primitives (polyhedrons, text, lines etc.).

The advantage of defining the geometrical information of the objects in separate files is that they can be used among several worlds, and more than once in the same world.

Here is the world description file used for the examples ("`hello_world.world`").

```
Simple Virtual Object File Format version 1.1
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 and textured_polyhedron.

```
Simple Virtual Primitive File Format version 1.0
number of components: 4

component 1 type: polyhedron
Data of component 1:
# 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
# of faces:
6
faces: R G B #_of_vertices v1 v2 v3 ...
30 0 0 4 0 1 2 3
```

```
10 0 0 4 0 1 5 4
20 0 0 4 1 2 6 5
22 0 0 4 2 3 7 6
35 0 0 4 3 0 4 7
10 0 0 4 4 5 6 7

component 2 type: polyline
Data of component 2:
# 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
# of polylines:
6
polylines: R G B #_of_vertices v1 v2 v3 ...
100 100 100 4 0 1 2 3
100 100 100 4 0 1 5 4
100 100 100 4 1 2 6 5
100 100 100 4 2 3 7 6
100 100 100 4 3 0 4 7
100 100 100 4 4 5 6 7

component 3 type: text
Data of component 3:
transformation matrix:
0.4  0    0    0
0    0.4  0    0
0    0    0.3  0
0.1  1.5  0.36 1
# of lines:
1
***this is text***

component 4 type: textured_polyhedron
Data of component 4:
image file: testimage.rgb
repeat texture: FALSE
blending: TRUE
# of vertices:
4
vertices: x y z
-0.3 -1.2 -0.0 0 0
-0.3 -0.2 -0.0 0 1
 0.3 -0.2 -0.0 1 1
 0.3 -1.2 -0.0 1 0
# of faces:
1
faces: R G B #_of_vertices v1 v2 v3 ...
100 100 100 4 0 1 2 3
```

File conversion from Wavefront

a simple convertor exists to convert objects made with the Wavefront modeler (.obj files) to object files:

```
wave2sve <source> <dest>
```

The converter currently converts the.objects to (flat-shaded) polygons. Colors are only converted when the material properties were explicitly included in the model before saving (by choosing “load materials” in the properties menu of wavefront).

We will expand the capabilities of this convertor, as wavefront provides useful tools to define texture coordinates to polygons (a very tedious task to do by hand).

Formal Definition of File Format -- World Description File

This section describes the world file format in grammar notation. The following notation is used:

{} specifies a list of alternatives (1 has to be chosen),

[] denotes an optional section,

[]* is 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

<string> = string, any character in the range [32..] (no control characters).

<stringID> = string, any character in the range {[‘0’..’9’], [‘a’..’z’], [‘A’..’Z’], ‘_’}

Boldfaced and capitalized entries refer to other grammatical expression.

Indentation is for clarity only, it is not necessary.

WORLDFILE:

Formal Description of file format version 1.1↵

number of objects: <n>↵

ø

[**OBJECTENTRY**]+

The worldfile specifies all the objects that are 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 rootobjects etc.)

OBJECTENTRY:

object name: <stringID>↵

primitives file: <stringID>↵

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>↵
[child data:↵
[**OBJECTENTRY**]+
]

The object is specified by a primitives file name (look at the object description for the file format of these) 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:

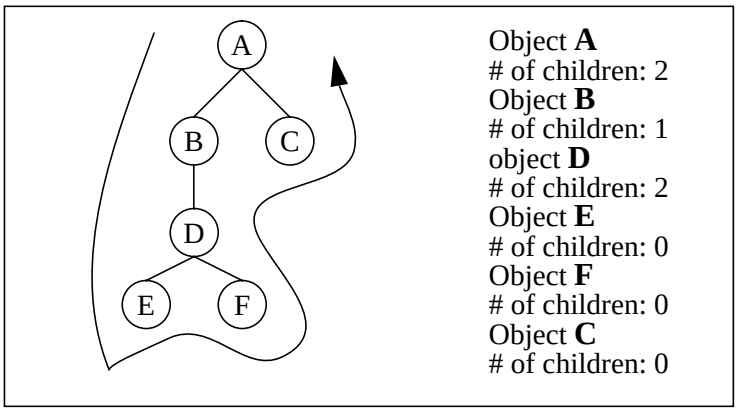


Figure B1: Depth first notation of a tree hierarchy.

ATTRIBUTE-ENTRY:

{color **COLORENTRY**↵,
visible {TRUE, FALSE}↵,
visible_sphere <f>↵,
selectable {TRUE, FALSE}↵,
title <f> <f> <f> <f> <string>↵
}

Extra attributes can be assigned to each object, currently, the following:

- 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 isSelectable in the object structure.
- title x y z s string: a text string will be rendered on the x,y,z (object) coordinates with size s.

COLORENTRY:

<f> <f> <f>

Red, green and blue values in the range [0,1] (entries larger than 1.0 will be divided by 256!)

Formal Definition of File Format -- Object Description File

This section describes the object file format in grammar notation. The following notation is used:

{} specifies a list of alternatives (1 has to be chosen),

[] denotes an optional section,

[]* is 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

<string> = string, any character in the range [32..] (no control characters).

<stringID> = string, any character in the range {['0'..'9'], ['a'..'z'], ['A'..'Z'], '_'}

Boldfaced and capitalized entries refer to other grammatical expression.

Indentation is for clarity only, it is not necessary.

OBJECT FILE:

Simple Virtual Primitive File Format version 1.1↵

[**BOUNDARY-ENTRY**]

number of components: <n>↵

[**PRIMITIVE-ENTRY**]+

BOUNDARY-ENTRY:

{boundary box <f> <f> <f> <f> <f> <f>↵,

boundary 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:

- boundary 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.

The object has no boundaries by default, in that case the *boundaries*-entry of the object is assigned to *NULL*.

PRIMTIVEENTRY:

{**POLYHEDRONENTRY**,

1. these algorithms are currently still under development, the descriptions are only loaded into the boundaries-entry of the object.

POLYLINENENTRY,
TEXTENTRY,
TEXTURED_POLYHEDRONENTRY}

POLYHEDRONENTRY:

component <n> type: polyhedron↵
Data of component <n>:↵
# of vertices:↵
<n>↵
vertices: x y z↵
[<f> <f> <f>↵]+
# of faces:↵
<n>↵
faces: R G B #_of_vertices v1 v2 v3 ...↵
[**COLORENTRY** <n> <n> <n> [<n>]+ ↵]

The polygons are defined by referring to the vertices lists. These indices begin at 0 (zero).

POLYLINEENTRY:

component <n> type: polyline↵
Data of component <n>:↵
# of vertices:↵
<n>↵
vertices: x y z↵
[<f> <f> <f>]+↵
# of polylines:↵
<n>↵
polylines: R G B #_of_vertices v1 v2 v3 ...↵
[**COLORENTRY** <n> <n> <n> [<n>]+ ↵]

TEXTENTRY:

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>↵
# 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. Texts are always rendered in white, unless its object overrides the color (by having the extra attribute *color* in the world file format).

TEXTURED_POLYHEDRONENTRY:

component <n> type: textured_polyhedron↵

Data of component <n>:↵

image file: <stringID>↵

repeat texture: {TRUE, FALSE}↵

blending: {TRUE, FALSE, GREYSCALE}↵

# of vertices:↵

<n>↵

vertices: x y z↵

[<f> <f> <f> ↵]+

# of texture vertices:↵

<n>↵

texture vertices: u v↵

[<f> <f> ↵]+

# of faces:

<n>↵

faces: R G B #_of_vertices v1 v2 v3 ... [t1 t2 t3 ...]↵

[**COLORENTRY** <n> [<v>]+ [<t>]+↵]

The textured polygon entry is similar to the regular polyhedron, its major difference is the addition of three more lines at the begin of the definition, a list of two dimensional points which map a texture on a face, and an index to that list for each vertex index in the face definition.

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 texture vertices list of (0,0), (0,2), (2,2), (2,0) will result in four images on the face being defined. 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

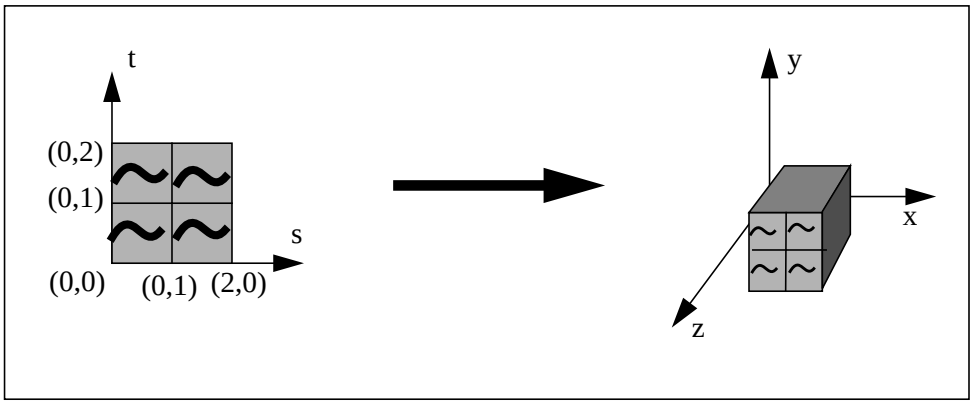


Figure A2: mapping s,t coordinates to polygons.

face is therefore defined as its color values, the number of vertices, a list of 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.

As for the first additional entries *image file*, *blending* and *repeat*:

- The image file must have the SGI rgb format (a variety of applications can be used to convert .gif and other file formats to these[1]).
- The repeat flag determines wether the texture should be repeated when the s,t coordinates exceed the range [0,1] (see gl-manual 18-7).
- The blending flag determines 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.

COLORENTRY:

<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!)

1. Golden oldies are the applications *fromtiff* <*sourcefile*> <*destfile*> and *fromgif* <*sourcefile*> <*destfile*>, these can be found in the directory *~ccoprrm/bin*.

APPENDIX C: Reference Manual

1. 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.

1.1. State information

```
typedef struct SVE_stateStruct {
   char *programName;
   SVE_objectList *objectTree;
   Matrix viewingMatrix;
   Matrix hmd;
   SVE_object hmdObject;
   SVE_object cursorObject;
   SVE_gloveData *glove;
   SVE_point origin;
   SVE_eventType event;
   short eventVal;
   boolean ntscOn;
   int config;
   float flightSpeed;
   boolean checkForGesture;
   SVE_gestureList *gestures;
} SVE_stateStruct;

typedef SVE_stateStruct *SVE_state;
```

This is the data structure given to each callback routine defined by the application. This structure is the life blood of the SVE system. Every piece of information about the current state is contained in 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, event, eventVal, config), although some fields are intended to be altered for desired effects (these are shown in italics).

Here is a short description of each field:

```
char *
```
- **programName;**

Given name of the application. This is the name on the application's window.

```
SVE_objectList *
```
- ***objectTree*;**

This references another data structure that defines the objects in the SVE world (the objects SVE will render for each frame). (See SVE_objectList)

Matrix
- **viewingMatrix;**

This is the matrix defining the position and orientation of the user (the first tracker). The position of the user is relative to the “origin” defined below. This the viewing matrix used by the render routine of the SVE system. It is updated once per frame if a tracker is being used. If a tracker is not being used, only the position information is changed (it is set to be at the “origin”).

Matrix
- **hmd;**

This is the position/orientation matrix of the first tracker (which has not been altered to obtain a viewing matrix).

SVE_object
- **hmdObject;**

This references an empty object whose position matrix mirrors the `position` matrix of the first tracker. Any objects attached as children to this object will follow the first tracker’s position and orientation.

SVE_object
- **cursorObject;**

This references the root object of the cursor. The root object itself is an invisible, empty object. If the glove is active, a glove object is defined (“hand” is the glove’s name), and is one of the children of the cursor. The cursor object’s `position` matrix is given the position and orientation of the second tracker (in relation to the `origin` defined in the `SVE_state` structure), and is therefore drawn each frame at that location.

SVE_gloveData *
- **glove;**

This is a reference to the position and orientation of the second tracker, and the definition of the object attached to it. If the glove input device is active, the values given by the device are also stored here. (see SVE_gloveData)

SVE_point
- ***origin*;**

The origin of the SVE world in relation to the user. This structure is typically altered during flying etc.

SVE_eventType
- **event;**

This is the gl event type. Possible values are gl events plus SVE events:

LEFTMOUSEDRAG
MIDDLEMOUSEDRAG

```
RIGHTMOUSEDRAG
ALLBUTTONS
GESTURE
```

---

short
- **eventVal;**

This is the gl event value. (See Glove utilities section in the function reference appendix for `GESTURE` event values.)

---

boolean
- **ntscOn;**

This flag determines if the application window is on the computer screen (`FALSE`), or being sent to the video output (`TRUE`).

---

int
- **config;**

This defines the configuration of the SVE system. It is a bit-wise `OR` combination of the following values:

```
SVE_NORMAL   Normal SVE operation.
SVE_HMD      Poll tracker device.
SVE_GOURAUD  Render using gouraud shading.
SVE_TEXTURES Render texture maps (otherwise ignored).
SVE_GLOVE    Poll glove device and load in the model of a hand.
```

---

float
- ***flightSpeed*;**

Determines the speed at which a user moves through the SVE world when “flying”. Its is in meters per frame.

---

boolean
- ***checkForGesture*;**

Flag to determine wether the current glove position should be compared to the glove position in the gestures list (below).

SVE_gestureList *
- **gestures;**

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 gl `GESTURE` event is generated. The event value will be `MIN_GESTURE` + (gesture index), where the gesture index is the priority given in *SVE_saveCurrentGesture*.

1.2. SVE_objectList

```
typedef struct SVE_objectList {
 SVE_object o;
```

```
 struct SVE_objectList *next;
} SVE_objectList;
```

This is a linked list of SVE objects (see SVE_object).

1.3. SVE_object

```
typedef struct SVE_objectStruct *SVE_object;

typedef struct SVE_objectStruct {
 SVE_object parent;
 char *name;
 SVE_primitiveList *primitives;
 SVE_primitive *label; /* for hypertexts */
 char *text; /* for hypertexts */
 char *textfile; /* for hypertexts */
 Matrix position;
 float color[4];
 boolean visible, hascolor, selectable;
 boolean hasBoundaries, hasSphere;
 float radius; /* for boundary spheres */
 SVE_boundaries *boundaries;
 boolean hasVisibleSphere ;
 float visibleSphere;
 boolean primitivesChanged;
 char *primitivesFilename;
 Object glID;
 boolean needsUpdating;
 SVE_objectList *children;
} SVE_objectStruct;
```

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 the object's parent. The root object in an object tree has a NULL parent.

---

char *
- **name;**

The string identifier of an object.

SVE_primitiveList *
- **primitives;**

This references a list of the parts that make up the object (see SVE_primitiveList).

```
   SVE_primitive *
•  label;
```

Hypertext that will be displayed at the object's position and orientation (given in the `position` matrix).

```
   char *
•  text;
```

```
   char *
•  textfile;
```

Used to save the hypertext if it has been changed.

```
   Matrix
•  position;
```

This 4X4 matrix defines the position and orientation of the object in relation to its parent.

```
   float
•  color[4];
```

This is the (red, green, blue, alpha) color value of the object as a whole. This value is only used if the `hascolor` flag (below) is `TRUE`.

```
   boolean
•  visible;
```

This flag determines if the object is rendered by SVE during the frame rendering. 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.

```
   boolean
•  selectable;
```

Determines if an object is considered for selection during the intersection algorithm the SVE system uses to determine which object has been selected.

```
boolean
hasBoundaries, hasSphere;

float
radius;
```

These variables are for backword compatibility and should not be used.

```
SVE_boundaries *
boundaries;
```

This references a structure that determines the bounding volume of the object, used for intersection algorithms.

```
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 if within this distance from the user (first tracker) in the SVE world, and not visible if the user is outside of this range.

```
boolean
primitivesChanged;
```

This flag, initially `FALSE`, is set to `TRUE` if an attribute of the object changes by the SVE system. It is used by the SVE save functions to decide if a file needs to be updated.

```
char *
primitivesFilename;
```

This string gives the filename of the file from which the object's attributes where drawn. Used to update the file during the SVE save functions.

```
Object
glID;
```

This is the handle for the gl object defined for this SVE object. It is used to increase the speed of rendering the object.

---

boolean
- **needsUpdating;**

Flag used to decide if a new gl object needs to be redefined for the object, as its geometry has changed in some manor.

SVE_objectList *
- **children;**

This is a linked list of children objects to this object. It is unordered and unbounded.

1.4. SVE_primitiveList

This is a linked list of primitives that make up an object. (see ***SVE_primitive***)

```
typedef struct SVE_primitiveList {
 SVE_primitive primitive;
 struct SVE_primitiveList *next;
} SVE_primitiveList;
```

1.5. SVE_primitive

```
typedef struct SVE_primitive {
 SVE_primitiveType type;
 int numVertices;
 SVE_pointList *vertices;
 SVE_faceList *faces;
 float color[4];
 char *text;
 int lineWidth;
 Matrix matrix; /* only used for text primitives.. */
} SVE_primitive;
```

A short description of each field of the SVE_primitive follows.

SVE_primitiveType
- **type;**

This defines the type if primitive. Possible values are:

```
POLYHEDRONPrimitive consists of a list of faces.
LINEPrimitive consists of one line with many points.
TEXTPrimitive consists of a string of characters.
```

---

int
- **numVertices;**

Number of vertices in the vertex list.

SVE_pointList *
- **vertices;**

Linked list of vertices (and normal for gouraud shading).

```
typedef struct SVE_pointList {
```

```
        SVE_point vertex;
        SVE_point normal; /* for gouraud shading */
        struct SVE_pointList *next;
} SVE_pointList;
```

**SVE_faceList *
• faces;**

Linked list of faces (for a `POLYHEDRON`). Each face has a list of vertices (actually an index into the primitive's vertex list), a color (red, green, blue, alpha), and a normal (used for flat shading).

```
typedef struct SVE_faceList {
    SVE_indexList *indices;
    float color[4];
    SVE_point  normal;
    struct SVE_faceList *next;
   } SVE_faceList;

   typedef struct SVE_indexList {
    SVE_pointList *index;
    struct SVE_indexList *next;
   } SVE_indexList;
```

**float
• color[4];**

Color value (red, green, blue, alpha) of the primitive.

**char *
• text;**

Text string used for a `TEXT` primitive.

**int
• lineWidth;**

line width used for a `LINE` primitive.

**Matrix
• matrix;**

Transformation matrix used for the `TEXT` primitive.

1.6. 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;
```

1.7. SVE_gloveData

This data structure defines the cursor used for the second tracking device and (if active) the glove input device.

```
typedef struct SVE_gloveData {
    boolean gloveActive;
    CyberGlove *gloveData;
    float angle[FNGRS][JNTS];
} SVE_gloveData;
```

What follows is a short description of each field of this structure:

---

boolean
- **gloveActive;**

Determines if the glove input device is active.

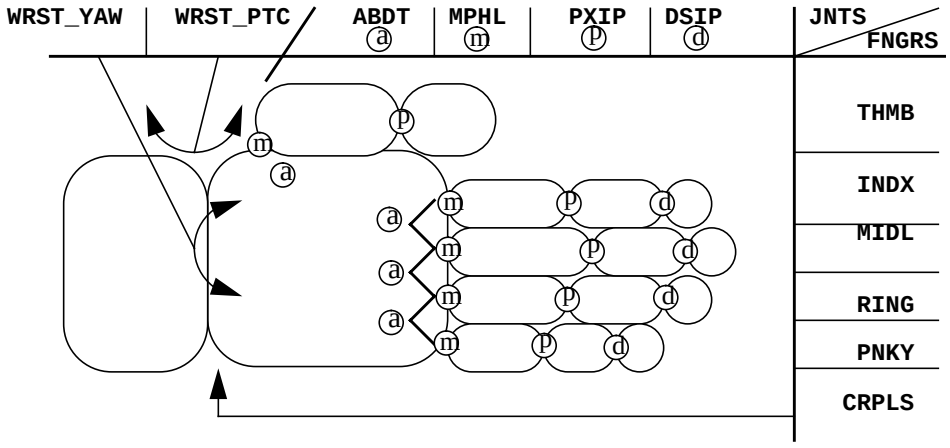
CyberGlove *
- **gloveData;**

If the glove input device is active, this references the data structure used by the glove routines.

---

float
- **angle[FNGRS][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

1.8. SVE_gestureList

This is an ordered linked list of gestures.

```
typedef struct SVE_gestureList {
 SVE_gesture *gesture;
 struct SVE_gestureList *next;
} SVE_gestureList;
```

1.9. SVE_gesture

This defines a gesture used to match against the current configuration of the glove device. A gesture is defined as the set of angle values for each joint, within a certain range. Each gesture has a unique priority. 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 struct SVE_gesture {
 int priority;
 float range;
 float angle[FNGRS][JNTS];
} SVE_gesture;
```

2. SVE Function Reference

2.1. Main SVE loop

---

boolean
- **SVE_init(char *programName, SVE_config config);**

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. (see SVE_state, config field)

This function must be called before any other SVE function call. It sets up the SVE system state.

---

void
- **SVE_beginEventLoop(void);**

This function begins the SVE system loop. Execution will not return to this point until the application is quit, or *SVE_stopEventLoop* is called. The application will only receive execution 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.

---

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 video output, the computer screen is returned to normal operation.

2.2. World/object utilities

2.2.1 Load/Save

---

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

char *filename	Path and filename of the SVE world file.

This function loads in an SVE world from the file given by the filename, and sets that world to be the SVE world rendered during the SVE render phase. If the given file does not exist at the path given, the Default Object Directory is used. The world read becomes the SVE world which the SVE system will render at each frame. If it is not successful, the function returns FALSE.

SVE_objectList *
- **SVE_loadObjects(char *filename, SVE_object parent);**

char *filename	Path and filename of an SVE world file.
SVE_object parent	Parent object of all objects to be read from the given file.

Reads in the list of objects contained in the given file. The parent of each object is set to the given `parent` value. Returns an unordered linked list of `SVE_object`'s.

SVE_object
- **SVE_loadObject(char *filename, char *name);**

char *filename	SVE object file path and file name.
char *name	Name to be associated with the object.

This function allocates memory for a 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`.

SVE_object
- **SVE_addObjects(char *file);**

char *file	Object file name.

Loads a group of objects from the world file named `file` (using the Default Directory if it is not found at the given path) and stores it in the global object tree of the SVE environment.

---

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

  `char *filename` `SVE world file path and file name.`

The current SVE world object hierarchy is saved in a SVE world file format in the file given. Objects which have changed (see the `SVE_object` data structure description) will be saved to their SVE object file.

---

boolean
- **SVE_saveObjects(SVE_objectList *o, char *filename);**

  `SVE_objectList *o` `Linked list of SVE objects to save.`

  `char *filename` `Path and name of the file to save to.`

Saves the list of objects in an SVE world file format to the file given. Objects which have changed (see the `SVE_object` data structure description) will be saved to their SVE object file.

2.2.2 Information

SVE_object
- **SVE_findObject(char *name, SVE_objectList *objectTree);**

  `char *name` `Name of desired object.`

  `SVE_objectList *objectTree` `Tree objects 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.

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.

---

void
- **SVE_getWorldMatrix(SVE_object o, Matrix m);**

  `SVE_object o` `The SVE object in question.`

  `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).

---

void
- **SVE_printObjectList(SVE_objectList *list, boolean printChildren);**

SVE_objectList *list	SVE object tree.
boolean printChildren	Flag used to shorten/lengthen output.

Prints to `stderr` the names of the objects in the `list` SVE objectlist. Their children (and their children's children, etc.) are printed also if the `printChildren` flag is `TRUE`.

2.2.3 Manipulation

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

char *name	Name to be given to the object.

Allocates memory for an object with no primitives 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
primitives = NULL
visible = TRUE
selectable = FALSE
hasVisibleSphere = FALSE
visibleSphere = 0
hascolor = FALSE
color = 0, 0, 0, 0
hasBoundaries = FALSE
hasSphere = FALSE
radius = 0.0
children = NULL
label = NULL
primitivesFilename = NULL
primitivesChanged = FALSE
boundaries = NULL
needsUpdating = TRUE
glID = NULL
text = NULL
textfile = NULL
position = identity matrix
```

SVE_object
- **SVE_removeObject(char *name, SVE_objectList **objectTree);**

char *name	Name of object to be removed.
SVE_objectList **objectTree	Address of reference to an object tree.

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.

---

void
- **SVE_removeObjectEntry(SVE_object object, SVE_objectList **objectTree);**

SVE_object object	Object to be removed.
SVE_objectList **objectTree	Address of reference to an object tree.

Removes the object (including its children) from the given object tree.

---

void
- **SVE_addToObjectList(SVE_object o, SVE_objectList **objectTree);**

SVE_object o	An object to add to an object tree.
SVE_objectList **objectTree	Address of reference to an object tree.

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`.)

---

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`.

---

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.

2.3. Callback utilities

2.3.1 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 documentation on events and callbacks.)

---

void
- **SVE_registerCallback(int event, SVE_functionPtr function);**

int event	gl 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 gl event queue. If the default callback is enabled, it will be called after the user defined callback has been executed (see *SVE_enableDefaultCallback* below).

---

void
- **SVE_removeCallback(int event);**

int event	gl or user defined event.

Removes the user defined callback function (set by *SVE_registerCallback*). When the event `event` reaches the front of the event queue, the user defined callback function will not be called. The default callback function will be called if it is enabled (see *SVE_enableDefaultCallback* below).

---

void
- **SVE_enableDefaultCallback(int event);**

int event	gl or user defined event.

Enables the default callback for the event `event`. The default callback will be called when the event reaches the front of the event queue. It is called after the user defined callback for the given event, if it has been set (see *SVE_registerCallback*).

---

void
- **SVE_disableDefaultCallback(int event);**

int event	gl or user defined event.

Disables the default callback for the event `event`. The default callback will not be called when the event reaches the front of the event queue. If a user defined callback function has been defined for this event (see *SVE_registerCallback*), it will be called.

SVE_functionPtr
- **SVE_getEventCallback(int event);**

`int event`	`gl or user defined event.`

Returns a pointer to the current callback for the event `event`. This function is useful for storing old event-callback functions before replacing them.

2.3.2 Frame callback

---

void
- **SVE_setFrameCallback(SVE_functionPtr function);**

`SVE_functionPtr function`	`User defined function.`

Sets the function that will be called by the SVE system just after it clears the back buffer and loads the current viewing matrix (see the `SVE_state` documentation), and just before the SVE system renders all of the objects in its object tree (given in the `SVE_state` structure). See the `SVE_functionPtr` documentation for its format.

SVE_functionPtr
- **SVE_getFrameCallback(void);**

Gets the frame callback function, see the `SVE_functionPtr` documentation for its format. This function is useful when the application needs a couple of frames to do an animation and than has to restore to its old behavior.

2.4. General utilities

2.4.1 State functions

---

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.

2.4.2 Matrix functions

---

void
- **SVE_invertMatrix(Matrix mat, Matrix inv);**

`Matrix mat`	`Operand.`
`Matrix inv`	`Result.`

Inverts the given matrix `mat` and returns the result in the matrix `inv`.

---

void
- **SVE_getRelativeMatrix(Matrix a, Matrix b, Matrix result);**

Matrix a	Source position matrix.
Matrix b	Destination position matrix.
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 [a][result] = [b].

---

double
- **SVE_getMatrixDist(Matrix a, Matrix b);**

Matrix a	Operand.
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(Matrix s, Matrix d);**

Matrix s	Source matrix.
Matrix d	Destination matrix.

Copies the 4X4 matrix `s` to the matrix `d`.

2.5. Cursor utilities

2.5.1 Cursor Information

---

void
- **SVE_getCursorPosition(Matrix pos);**

Matrix pos	Result.

Stores the position and orientation matrix of the second tracker when it was last polled in the matrix `pos`. This position will be equal the state->origin vector when the trackers are not active.

SVE_object
- **SVE_getCursorObject(void);**

Returns the `cursor` object, which has the position and orientation of the second tracker. The 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 second tracker.

2.5.2 HMD Information

---

void
- **SVE_getHMDPosition(Matrix pos);**

`Matrix pos` Result.

Stores the position and orientation matrix of the first bird tracker when it was last polled in the matrix `pos`. This position will be equal the state->origin vector when the trackers are not active.

SVE_object
- **SVE_getHMDObject(void);**

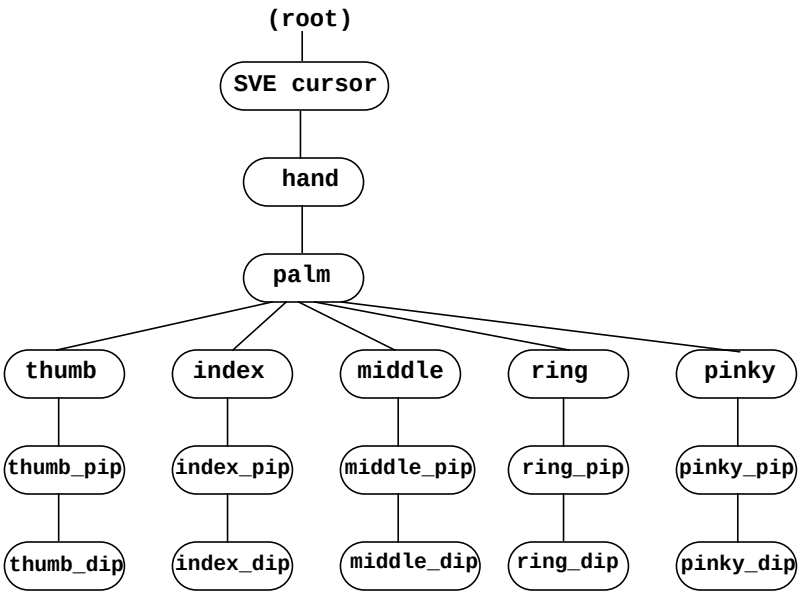
Returns the `HMD` object, which has the position and orientation of the first tracker. The HMD object begins as an empty object, but can be given primitives and/or children objects which will be rendered at the location and with the orientation of the first tracker.

2.5.3 Glove utilities

---

boolean
- **SVE_initGlove();**

Initializes the glove input device and sets up an object structure for the hand's representation in the SVE environment. The glove is defined to be attached to serial port 1 or port 2 (if port 1 is unavailable), and the hand object structure is retrieved from the "glove" sub-directory of the SVE "objects" directory. The hand object structure is attached to the "SVE cursor" object, which always has the position and orientation of the second tracker.



The glove structure

---

void
- **SVE_saveCurrentGesture(SVE_state worldState, int priority, float range);**

SVE_state worldState	SVE state structure.
int priority	Unique priority value. Ranges from 1 to 49.
float range	Range (in radians) of error for each joint angle.

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 keyed by the given priority. There can only be one gesture per priority. 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 `GESTURE` event. The `GESTURE` event, with the priority of the matching gesture begin the event value, will be placed on the gl event queue.

---

void
- **SVE_recognizeGestures(boolean flag);**

boolean flag	Enable flag.

If `flag` is `TRUE`, the current gesture made by the hand in the glove input device will be checked with the list of gestures which have been saved for a match (with-in the error given when the gesture was saved). If a match is found, a `GESTURE` event will be placed on the gl event queue (see *SVE_saveCurrentGesture*) with an event value corresponding to the gesture index. Otherwise a `GESTURE` event will occur with the value `NULL_GESTURE`.

SVE_object
- **SVE_getPalmObject(SVE_state worldState);**

SVE_state SVE_worldState	SVE system state structure.

Returns the object that is the palm object of the hand structure (see *SVE_initGlove*).

---

void
- **SVE_setGloveLight(boolean flag);**

boolean flag	switch flag.

Switches the red LED on the cyberGlove on when the flag is TRUE, or off when FALSE.

2.6. 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. Can only be used when the system was initialized with the constant *SVE_SPATIALSOUND*.

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. Can only be used when the system was initialized with the constant *SVE_SPATIALSOUND*.

SVE Function Index

SVE Data Types & Fields Index