Introduction to Virtual Reality Visualization
by the CAVE system

Nobuaki Ohno and Akira Kageyama

Earth Simulator Center,
Japan Agency for Marine-Earth Science and Technology,
3173-25 Showa-machi, Yokohama 236-0001, Japan

The virtual reality (VR) technology provides us a three-dimensional, immersive, and fully interactive visualization environment. Among various VR systems, the CAVE is the most attractive option for simulation researchers. In this chapter, we introduce basics of CAVE programming. The readers will be able to make a visualization software for the CAVE by themselves.

1 Introduction

Simulation science is supported by two key technologies; computation and visualization. The technology of computation, or computer, is still keeping an exponential growth. In this sense, we are living in a golden age of simulation science. However, a high-rise building should be supported by balanced pillars. Unfortunately, the development of visualization technology does not catch up with its counterpart. It would be absurd if we had to spend a month to understand a simulation data obtained by one-hour-job on a high-performance computer.

In the Stone Age of computer simulation when 1-dimensional (1-D) phenomena are mostly simulated, simple graph plot on a piece of paper would have been sufficient to understand or to visualize the data. When 2-D simulation were common, iso-line plots and contour color plots on papers or computer monitor screens would have been enough.

In the beginning of 1990’s, 3-D simulations with the grid size of $O(100^3)$ became to be not rare. We familiarized ourselves with newly introduced visualization technology at that time; it was a combination of visualization software and graphic workstations (GWS). We could instantly see, for example, an isosurface of 100$^3$ grid points through the GWS’s screen. We could zoom in, rotate, color the objects, and change the isosurface levels, via graphical user interface with the mouse. By the interactive manipulation through the GWS’s screen, we could grasp 3-D structure of the numerical data. This visualization technology based on GWS was certainly powerful enough at that time.

Now, the complexity of target phenomena on the present high-end super computer, such as the Earth Simulator\(^1\), are extremely high; the typical grid size is $O(1000^3)$.

\(^1\)http://www.es.jamstec.go.jp/esc/eng/index.html
Thanks to the advanced graphic hardware these days, we can certainly render, rotate and control an isosurface of a scalar data on $O(1000^3)$ grid points. However, the complicated shape itself of the isosurface prevents us from the intuitive and immediate understanding of the 3-D structure of the scalar field.

Another limitation of the GWS-based visualization technology is the ability (or inability) of visualization of vector fields. In our simulation research, 3-D vector field is common, for example, the fluid flow, electric field, and magnetic field. We need to grasp spatial structure of the 3-D vector fields defined on $O(1000^3)$ grid points in order to understand what was simulated on the computer. This is really a challenging task and obviously beyond the GWS-based visualization technology. Simulation scientists today really need innovation in the visualization technology that suits the modern high-performance computer. And we believe that the virtual reality (VR) is the answer.

Everyone who experiences a modern VR system for the first time would be surprised by its high quality of computed reality. They feel like they are deeply immersed, or really standing, in the mimicked world. In order to produce such a deep
absorption into the VR world, there are three important visual factors\(^2\); (1) stereo
view, (2) immersive view, and (3) interactive view. Among various kinds of available
VR hardware, we believe that the CAVE [Cruz-Neira:1993] system is the best suited
to our purpose of the visualization of large scale 3-D scalar and vector fields.

The CAVE is developed at Electric Visualization Laboratory (EVL), University of
Illinois\(^3\). We installed our CAVE system at Earth Simulator Center, Japan Agency for
Marine-Earth Science and Technology (JAMSTEC) in 2002, and named it BRAVE.

2 Hardware of VR Visualization: BRAVE

BRAVE has a cubic room of the volume \((3m)^3\). It has four stereo screens (three
walls and a floor) as shown in Fig. 1. The viewer stands in the BRAVE room on the
floor screen, wearing a pair of stereo glasses [Fig. 2(a)], having a portable controller
called wand [Fig. 2(b)]. Stereo images are projected on the three walls and the floor
by four DLP projectors [Fig. 3(a)]. A GWS (SGI Onyx3800) is used to generate the
stereo images [Fig. 3(b)]\(^4\). The Onyx3800 in our system has 12 CPUs, 24GB mem-
ory, and 2 graphics pipes. Since the viewer inside the BRAVE room is surrounded
by stereo images on the three walls and the floor, he or she can look around in the
BRAVE room, keeping the wide range of stereo view angle. This strongly gener-
ates the immersive sense of the viewer. The refresh rate of the image is 96 Hz: The
right-eye image and the left-eye image are projected alternately with 48 Hz each. The
image refresh is synchronized with the alternate shutter of the stereo glasses by the
infrared emitter/sensor system. The images on the boundaries between the walls and
the floor are smoothly connected. The viewer can easily forget the existence of the
boundaries. Actually, we saw some people had banged on the BRAVE’s walls!

BRAVE has a magnetic tracking system to detect the position and direction of
the viewer’s eyes in the room. A transmitter of the magnetic tracking system is in-
stalled on the ceiling of the BRAVE room and a small magnetic sensor is attached
on the stereo glasses [see Fig. 2(a)]. Another magnetic sensor is installed in the
wand [Fig. 2(b)] to detect the position and direction of the wand (or the viewer’s
hand). All the projected images are automatically adjusted following the viewer’s
head motion in real time. Therefore, everything looks natural from the viewer who
can incline, walk, sit, or even jump in the BRAVE room to observe 3-D objects in the
virtual world.

The wand, which has 3 buttons and 1 joystick\(^5\), is used as an interface with the
virtual world (or simulation data). For example, when the viewer presses a wand
button, a virtual menu appears in front of him or her, and he or she can choose a visu-
alization method by shooting a menu panel by a virtual laser beam emitted from the
wand. If one presses another button after choosing “Tracer Particle” menu, a virtual
tracer particle appears at the tip of the wand, and when the button is released, the

\(^2\)Sonifications [Tamura:2001] and haptics are other important VR technology that could be applied in
scientific visualization.

\(^3\)http://www.evl.uic.edu/index2.php

\(^4\)Recently, the CAVE system is sometimes constructed with PC cluster instead of expensive GWS.

\(^5\)A “trackball” may suit to its shape, but it is historically called joystick.
particle “flies” under the eyes following the velocity field showing the flow structure by its trajectory.

To model the virtual world in the CAVE, one can use (i) OpenGL, or (ii) OpenGL Performer. The OpenGL is a standard API for developing 3D computer graphics (CG) software [Shreiner:2003, 2004], it is used to define the visualization objects and its illumination. A CAVE application program is essentially a kind of common CG software without the projection part which is automatically processed by a basic VR library for the CAVE called CAVELib. So, if you are familiar with CG programming using OpenGL, it would be easy to make your own CAVE applications.

When you want to make an elaborated visual simulation software such as a drive simulator, you can use pCAVELib, which enables you to make a CAVE application written on OpenGL Performer. Fig. 4 shows an example of CAVE application programs using OpenGL Performer with pCaveLib. Here whole premises of Na-
tional Institute for Fusion Science (NIFS) in Japan are shown in the CAVE. One of the authors (N. Ohno), who developed this software, is walking through this “Virtual NIFS”. The object data of Virtual NIFS, which is written in the Open Inventor format, was created by himself using a freely available 3-D modeling software and a digital camera for textures. OpenGL Performer can display not only the Open Inventor format but also various formats of 3-D data. Although we will not mention it any further, interested users may refer to the web site of OpenGL Performer6.

3 CAVE Programming Introduction

In this section, we introduce how to make a program of the CAVE application. The readers are supposed to be familiar with C/C++ and OpenGL but need not to be expert. You will find that making a CAVE program is a lot easier than you expect.

3.1 Sample Program 1

First, see the following very simple CAVE application, which displays a white triangle in the CAVE [Fig. 5].

```c
/*
 *--------------------------------------------------------
 * Sample code : triangle.c
 * by Nobuaki Ohno, 2005
 *--------------------------------------------------------
 */

#include <cave_ogl.h>
#include <unistd.h>

void init_gl (void)
{
    glClearColor (0., 0., 0., 0.);
}

void draw (void)
{
    glClearColor (0., 0., 0., 0.);

    glColor3f (1.0, 1.0, 1.0);
    glTranslatef (0.0, 5.0, -5.0);
    glBegin (GL_TRIANGLES);
    glVertex3f (-2.0, 0.0, 0.0);
```

6http://www.sgi.com/products/software/performer/
This is very short but a still complete CAVE program. The function `init_gl()` [line 14] does the initialization of OpenGL, and `draw` [line 22] displays a white triangle in the CAVE. The readers who are familiar with OpenGL may easily understand the two functions. Notice that there is no projection- or stereo-related function in `draw` or any other part. As we mentioned, the projection part of a CAVE program is automatically taken care by CAVELib. In the main function, some important CAVELib functions are called. `void CAVEInit` [line 41] does the initialization of the CAVE. By `void CAVEInitApplication(CAVECALLBACK, int,...)` [line 43] and our `init_gl`, OpenGL is initialized. `void CAVEDisplay(CAVECALLBACK, int,...)` [line 45] specifies rendering function, which is `draw` in this case. This program displays a white triangle until the ESC key is pressed [line 47].

When you use GLUT library, which is a popular toolkit for writing OpenGL programs, the main function would be like this,

```c
main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(100, 100);
    glutInitWindowPosition(500, 500);
    glutCreateWindow("Analogy");
```
Fig. 5. A white triangle is displayed in the CAVE by sample program “triangle.c”.

Fig. 6. A developer uses the CAVE simulator.

Here, `draw` includes rendering part using GLUT.

```c
9
10    glutDisplayFunc(draw);
11    glutReshapeFunc(resize);
12    glutKeyboardFunc(keyboard);
13    glutIdleFunc(compute);
14    init_gl();
15    glutMainLoop();
16
19 }
```
3.2 CAVE Simulator

It is a common practice that the program development, including the CAVE program development, involves repeated cycle of to code, compile, test and debug. It would be very annoying if one had to get into the CAVE room, with the stereo glasses every time for the test step in the above development cycle. Fortunately, CAVE simulator is available which simulates the projected images and operations on the X Window. By using the CAVE simulator, the developer can test the application without going into the CAVE room. Fig. 6 shows a snapshot of the CAVE simulator for the test program “triangle.c”.

The CAVE simulator can display the CAVE’s room on GWS or PC screen from the viewer’s point of view as well as from outside the room. The wand and the viewer’s head are displayed by symbols. The viewer’s movement and the operations of the wand in the CAVE can be simulated by using the keyboard and mouse. Frequently used operations of the CAVE simulator are summarized in Table 1 to Table 4.

| Table 1. Head motions |  |
|-----------------------|  |
| **Keyboard and Mouse** | **Head Motion** |
| ← / → / ↑ / ↓ key | move left/right/forward/backward |
| Shift key + ↑ key | move upward |
| Shift key + ↓ key | move downward |

| Table 2. Wand Operations |  |
|--------------------------|  |
| **Keyboard and Mouse** | **Wand** |
| Ctrl key + Mouse movement | move the wand left/right/forward/backward |
| Shift key + Mouse movement | move the wand left/right/upward/downward |
| Alt key + Mouse movement | rotate the wand left/right/upward/downward |

| Table 3. Wand buttons and trackball operations |  |
|-----------------------------------------------|  |
| **Keyboard and Mouse** | **Wand** |
| Mouse left/middle/right button | wand left/middle/right button |
| Space key + Mouse movement | move trackball(Joystick) |

Fig. 7 shows a screen snapshot of the CAVE simulator which simulates “triangle.c” result. The red gadget in the center is the symbol for the wand.

Hereafter, the sample snapshots in this section are those of the CAVE simulator.
Table 4. Other Controls

<table>
<thead>
<tr>
<th>Keyboard</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 key</td>
<td>view from viewer’s point of view</td>
</tr>
<tr>
<td>2 key</td>
<td>view from outside</td>
</tr>
<tr>
<td>w key</td>
<td>make wand visible/invisible</td>
</tr>
<tr>
<td>u key</td>
<td>make viewer’s head visible/invisible</td>
</tr>
<tr>
<td>p key</td>
<td>reset head and wand to the initial</td>
</tr>
<tr>
<td>Insert key</td>
<td>make frame of the CAVE’s room visible/invisible</td>
</tr>
</tbody>
</table>

Fig. 7. This is a snapshot of CAVE simulator of “triangle.c”. The red gadget indicates the wand.

3.3 Processes of CAVE Application

A CAVE program is a multi-process program. There are three kinds of processes.

- **Tracking Process**
- **Display Process**
- **Application Computation Process**

The **Tracking Process** detects, in real time, the positions and directions of the stereo glasses and the wand while a CAVE application is running. The CAVE programmer can retrieve the position/direction data from this Tracking Process.

The **Display Process** draws the stereo images on the screens. The CAVELib automatically calculates the stereo projection based on the viewer’s eyes. Generally, there are multiple Display Processes while a CAVE application is running.
The Application Computation Process does calculations of, for example, orbits of the particle tracers. The functions which you want the Application Computation Process to do should be called in the "while loop" (for example, at line 47 triangle.c) in the main function.

The Display Processes and the Application Computation Process communicate via the shared memory allocated by "void *CAVEMalloc(size_t)" [Fig. 8]. If you want to move a ball in the CAVE, for example, the variables for the ball’s coordinates have to be stored on the shared memory.

3.4 Animation

The following is a sample program to show how to use the shared memory and carry out the animation in the CAVE. This program makes white star-shaped objects, or snowflakes, rotate and fall in the CAVE [Fig. 9].

Fig. 8. Processes communicate each other via the shared memory.

Fig. 9. Snowflakes are falling in the CAVE. This is a snapshot of CAVE simulator.
#include <cave_ogl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>

#define NFLAKES 500
#define XMAX 20.0
#define YMAX 20.0
#define ZMAX 10.0
#define XMIN (-20.0)
#define YMIN (-5.0)
#define ZMIN (-20.0)

/* The data that will be shared between processes */
struct _snowdata
{
  float xpos;
  float ypos;
  float zpos;
  float xaxis;
  float yaxis;
  float zaxis;
  float spin;
};

void init_gl (void), draw (struct _snowdata *);
struct _snowdata *init_shmem (void);
void compute (struct _snowdata *);

static GLuint flake_indx;

main (int argc, char **argv)
{
  struct _snowdata *snows;
CAVEConfigure (&argc, argv, NULL);
snows = init_shmem();
CAVEInit();
CAVEInitApplication (init_gl, 0);
CAVEDisplay ((CAVECALLBACK) draw, 1, snows);
while (!CAVEgetbutton (CAVE_ESCKEY))
{
    compute (snows);
    sginap (1);
}
CAVEExit();

float
randmf (void)
{
    float r, rndmax;
    rndmax = (float)(RAND_MAX + 1);
    r = rand () / rndmax;
    return r;
}

struct _snowdata *
init_shmem (void)
{
    int i;
    struct _snowdata *snows;
    snows = (_snowdata *) CAVEMalloc (
        NFLAKES * sizeof (struct _snowdata));
    bzero (snows, NFLAKES * sizeof (struct _snowdata));
    for (i = 0; i < NFLAKES; i++)
    {
        snows[i].xpos = XMIN + (XMAX - XMIN) * randmf ();
        snows[i].ypos = YMIN + (YMAX - YMIN) * randmf ();
        snows[i].zpos = ZMIN + (ZMAX - ZMIN) * randmf ();
        snows[i].xaxis = randmf ();
        snows[i].yaxis = randmf ();
        snows[i].zaxis = randmf ();
    }
    return snows;
}
void compute (struct _snowdata *snows)
{
    int i;
    float y, angle, velocity = 0.5;
    float t = CAVEGetTime ();
    static float prev_t = 0.0;
    for (i = 0; i < NFLAKES; i++)
    {
        y = snows[i].ypos - velocity * (t - prev_t);
        if (y < YMIN)
            y = YMAX;
        snows[i].ypos = y;
        angle = t * 50;
        angle = (int)angle % 360;
        snows[i].spin = angle;
    }
    prev_t = t;
}

void init_gl (void)
{
    float flake_size = 0.5;
    GLuint tmp_indx;
    glDisable (GL_LIGHTING);
    glColor3f (1.0, 1.0, 1.0);
    glClearColor (0., 0., 0., 0.);
    tmp_indx = glGenLists (1);
    glNewList (tmp_indx, GL_COMPILE);
    glBegin (GL_POLYGON);
    glVertex3f (flake_size / 2, 0.0, 0.0);
    glVertex3f (0.0, flake_size * sqrt (3.0) / 2.0, 0.0);
    glVertex3f (-flake_size / 2, 0.0, 0.0);
    glEnd ();
    glEndList ();
    flake_indx = glGenLists (1);
    glNewList (flake_indx, GL_COMPILE);
    glPushMatrix ();
    glCallList (tmp_indx);
    glPushMatrix ();
The function `compute(struct _snowdata *)` [line 95] is called in the Application Computation Process. This function calculates the coordinates and the rotation-angles of the snowflakes. Because the results are stored on the shared memory, the Display Processes can access them and draw the snowflakes rotating and falling in the CAVE.

As suggested in `compute` [line 95] and `draw` [line 149], in which only the y-component of the coordinates is changed to simulate the falling snowflakes. The coordinate system of the CAVE is set as shown in Fig. 10. (Make sure of the coordinate system of your CAVE system especially when your CAVE has less than four screens.)

The function to allocate the shared memory, `void *CAVEAlloc(size_t)`, must be called between CAVEConfigure and CAVEInit, so `init_shmem` is called at line 48. The function `float CAVEGetTime()` returns the seconds passed since the initialization of the CAVE. The default limit of the shared memory is 8MB. If you want to increase it, use `int CAVESetOption(CAVEID pid, int size)` and `CAVE_SHMEM_SIZE` for `pid`, with the `size` in bytes.
Here is one more sample program for the animation. This program takes you the outer space to see the solar system [Fig. 11].

```c
/*
 * Sample code : solar_system.c
 * by Nobuaki Ohno, 2005
 */

#include <cave_ogl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <math.h>

#define PI 3.14159
#define NUM_PLANET 10
#define YPOS 5.0
#define ORBIT_MAG 1.5
#define PLANET_MAG 0.15

/* The data that will be shared between processes */
struct _planetdata {
    float xpos;
    float ypos;
};
```
float zpos;

void init_gl(void), draw(struct _planetdata *);
struct _planetdata *init_shmem(void);
void compute(struct _planetdata *);

GLUquadricObj *sphereOBJ;
GLUquadricObj *diskOBJ;

GLuint orbits;
GLuint planet_obj[NUM_PLANET];
/
0: Mercury, 1: Venus, 2: Earth, 3: Mars
4: Jupiter, 5: Saturn, 6: Uranus, 7: Neptune, 8: Pluto
/*

GLfloat planet_color[NUM_PLANET][3] = {
{1.0, 0.89, 0.76}, {1.0, 0.49, 0.31}, {0.0, 0.0, 1.0},
{0.82, 0.41, 0.12}, {0.96, 0.96, 0.86}, {1.0, 0.92, 0.76},
{0.69, 0.77, 0.05}, {0.0, 0.0, 1.0}, {0.5, 0.5, 0.05}
};

GLfloat planet_rad[NUM_PLANET] =
{ 0.38, 0.95, 1.0, 0.53, 11.2, 9.5, 4.0, 3.9, 0.18 };
float rev_rad[NUM_PLANET] =
{ 0.39, 0.72, 1.0, 1.5, 5.2, 9.6, 19.1, 30.2, 39.5 };
float period[NUM_PLANET] =
{ 0.24, 0.63, 1.0, 1.9, 11.9, 29.5, 84.0, 165.0, 247.0 };

main(int argc, char **argv)
{
struct _planetdata *planet;
CAVEConfigure(&argc, argv, NULL);
planet = init_shmem();
CAVEInit();
CAVEInitApplication(init_gl, 0);
CAVEdisplay((CAVECALLBACK) draw, 1, planet);
while (!CAVEgetbutton(CAVE_ESCKEY)) {
    compute(planet);
    sginap(1);
}
CAVEExit();
}

struct _planetdata *init_shmem(void)
int i;
struct _planetdata *planet;

planet = 
(_planetdata *) CAVEMalloc(NUM_PLANET *
sizeof(struct _planetdata));
bzero(planet, NUM_PLANET * sizeof(struct _planetdata));

for (i = 0; i < NUM_PLANET; i++) {
    planet[i].xpos = ORBIT_MAG * rev_rad[i];
    planet[i].ypos = YPOS;
    planet[i].zpos = 0.0;
}

return planet;

void compute(struct _planetdata *planet)
{
int i;
float angle;
float t = CAVEGetTime();

for (i = 0; i < NUM_PLANET; i++) {
    angle = (int) (t * 50.0 / period[i]) % 360;
    planet[i].xpos = ORBIT_MAG * rev_rad[i] * cos(2.0 * PI * angle / 360.0);
    planet[i].zpos = ORBIT_MAG * rev_rad[i] * sin(2.0 * PI * angle / 360.0);
}

}

void init_gl(void)
{
int i, j;
GLfloat diffuse[4];
GLfloat ambient[4];
GLfloat specular[4] = { 1.0, 1.0, 1.0, 1.0);
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat light_ambient[] = { 0.3, 0.3, 0.3, 0.0 };  
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 0.0 };  
GLfloat light_position[] = { 0.0, 1.0, 1.0, 0.0 };  

glEnable(GL_LIGHT0);  

gllightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);  
gllightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);  
gllightfv(GL_LIGHT0, GL_SPECULAR, light_specular);  
gllightfv(GL_LIGHT0, GL_POSITION, light_position);  

gColor3f(1.0, 1.0, 1.0);  
glClearColor(0., 0., 0., 0.);  

gluNewQuadric();  
dglNewQuadric();  
ambient[3] = 1.0;  
diffuse[3] = 1.0;  

for (i = 0; i < NUM_PLANET; i++) {  
    ambient[0] = planet_color[i][0];  
    ambient[1] = planet_color[i][1];  
    ambient[2] = planet_color[i][2];  
    diffuse[0] = 0.75 * planet_color[i][0];  
    diffuse[1] = 0.75 * planet_color[i][1];  
    diffuse[2] = 0.75 * planet_color[i][2];  
    
    planet_obj[i] = glGenLists(1);  
    glNewList(planet_obj[i], GL_COMPILE);  
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);  
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);  
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);  
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100.0);  
    
    gluSphere(sphereOBJ,  
              planet_rad[i] * PLANET_MAG, 32, 32);  
    
    if (i == 5) { /* For Saturn’s ring */  
        glPushMatrix();  
        glRotatef(75.0, 1.0, 0.0, 0.0);  
        gluDisk(diskOBJ, 1.2 * PLANET_MAG * planet_rad[i],  
                2.25 * PLANET_MAG * planet_rad[i], 32, 32);  
        glPopMatrix();  
    }  
}
null
The orbits of the planets are supposed to be pure circles in this program. Incorporating more accurate orbits and periods of the revolutions, the sun, and comets as well as texture mapping to the planets for more realistic appearance are left to the readers.

The change of button’s state can be retrieved by

- \texttt{int CAVEButtonChange(int button) /* button = 1, 2, 3 */}.  

3.5 Stereo Glasses and Wand

As we mentioned, the \textit{Tracking Process} detects the positions and directions of the stereo glasses and the wand all the time and you can retrieve the data. The \textit{Tracking Process} also detects the states of the buttons and joystick of the wand, for example, whether the left button are pressed or not. You can create a highly interactive application by using following functions,

- \texttt{void CAVEGetPosition(CAVEID pid, float vec[3])},

- \texttt{void CAVEGetVector(CAVEID vid, float vec[3])}.

Various positions can be obtained by the former function. For example, Setting “\texttt{pid}” \texttt{CAVE\_HEAD}, and you can obtain the viewer’s position via \texttt{vec[3]}. And the directions can also be obtained by the latter function. The obtained vector is automatically normalized. The \texttt{pids} and \texttt{vids} are summarized in Table 5 and Table 6.
If the state does not change compared to the one you checked last time, this function returns 0. It returns 1, if the button has been pressed, and -1 if released.

There are some useful macros for the wand. The macros CAVEBUTTON1, CAVEBUTTON2 and CAVEBUTTON3 give us whether each wand button is being pressed or not. If being pressed, it returns value 1, and if not, value 0.

Some functions and macros introduced here are used in the following program. This program makes your wand a “beam sword” like the one appearing in science fiction movies (but you cannot cut anything with it). When pressing the left button of the wand, the wand is emitting a pale green blade [Fig. 12].

```c
#include <cave_ogl.h>
#include <unistd.h>

#define LENGTH 3.0

/* The data that will be shared between processes */
```
struct _sworddata
{
  float wand[3];
  float tip[3];
  int on;
};

void init_gl (void), draw (struct _sworddata *);
struct _sworddata *init_shmem (void);
void compute (struct _sworddata *);

main (int argc, char **argv)
{
  struct _sworddata *sword;
  CAVEConfigure (&argc, argv, NULL);
  sword = init_shmem ();
  CAVEInit ();
  CAVEInitApplication (init_gl, 0);
  CAVEDisplay ((CAVECALLBACK) draw, 1, sword);
  while (!CAVEgetbutton (CAVE_ESCKEY))
  {
    compute (sword);
    sginap (1);
  }
  CAVEExit ();
}

struct _sworddata *
init_shmem (void)
{
  int i;
  struct _sworddata *sword;
  sword = (_sworddata *) CAVEMalloc {
    sizeof (struct _sworddata));
  for (i = 0; i < 3; i++)
  {
    sword->wand[i] = 0.0;
    sword->tip[i] = 0.0;
  }
  sword->on = 0;
  return sword;
void compute (struct _sworddata *sword)
{
    int i;
    float w[3], d[3], t[3];
    if (CAVEBUTTON1)
    {
        CAVEGetPosition (CAVE_WAND, w);
        CAVEGetVector (CAVE_WAND_FRONT, d);
        for (i = 0; i < 3; i++)
        {
            t[i] = LENGTH * d[i] + w[i];
            sword->wand[i] = w[i];
            sword->tip[i] = t[i];
        }
        sword->on = 1;
    }
    else
    {
        sword->on = 0;
    }
}

void init_gl (void)
{
    glEnable (GL_LIGHTING);
    glClearColor (0., 0., 0., 0.);
}

void draw (struct _sworddata *sword)
{
    glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    if (sword->on)
    {
        glLineWidth (12.0);
        glBegin (GL_LINES);
        glColor3f (1.0, 1.0, 1.0);
        glVertex3fv (sword->wand);
If you do not want to keep pressing the left button of the wand to make it emitting the blade, use `CAVEButtonChange(1)` instead of `CAVEBUTTON1` in `compute`. If you replace `compute` by the following, you can switch on and off the blade by pushing the left button.

```c
void compute (struct _sworddata *sword)
{
  int i;
  float w[3], d[3], t[3];
  if(CAVEButtonChange(1) == -1) {
    if(sword->on == 0) sword->on = 1;
    else sword->on = 0;
  }
  if (sword->on == 1)
    CAVEGetPosition (CAVE_WAND, w);
    CAVEGetVector (CAVE_WAND_FRONT, d);
    for (i = 0; i < 3; i++)
      { t[i] = LENGTH * d[i] + w[i];
      sword->wand[i] = w[i];
      sword->tip[i] = t[i];
      }
  }
if (sword->on == 1)
  { CAVEGetPosition (CAVE_WAND, w);
    CAVEGetVector (CAVE_WAND_FRONT, d);
    for (i = 0; i < 3; i++)
      { t[i] = LENGTH * d[i] + w[i];
      sword->wand[i] = w[i];
      sword->tip[i] = t[i];
      }
  }
```

Note that if your wand has four buttons, `CAVEButtonChange(4)` and `CAVEBUTTON4` works for the fourth button.
The macros `CAVE_JOYSTICK_X` and `CAVE_JOYSTICK_Y` give us the coordinates of joystick, namely how much the trackball (joystick) is moved from the neutral position, in range from -1.0 to 1.0. Here, $X$ means left-right direction and $Y$ means up-down direction. They will be used in the program in the next subsection.

### 3.6 Navigation

You can walk in the CAVE’s 3m $\times$ 3m $\times$ 3m room, and the magnetic head tracking system automatically adjusts the screen images as you move. When you want to look the backside of an object floating in front of you in the center of the CAVE room, just walk through the object and turn around. But what if the object is floating in the space over the other side of a CAVE’s wall screen? For those cases CAVELib provides a convenient mechanism called navigation which enables you to translate or rotate the coordinates (or CAVE room) in the VR world by calling the following functions,

- `void CAVENavTranslate(float, float, float)`.
- `void CAVENavRot(float, char)`,
  
  /* first argument is the angle and the second is the axis */
- `void CAVENavTransform()`.

The following program is a modified version of “Snowfall.c” with navigation. The function `navigate` is added [line 184] and called in the “while loop” in the main function, and `CAVENavTransform` is added [line 157] in the `draw` in the Snowfall.c. Moving the joystick up or down, the CAVE room (and you) moves into the direction where
the wand points in the VR world, and moving it left or right, that of the room ro-
tates [Fig. 13]. As a result, you can move the objects to where you want in the VR
world by this mechanism.

Fig. 13. Concept of Navigation. The object is isosurface of pressure of spherical tokamak
plasma [Mizuguchi:2000].

```c
#include <cave_ogl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>

#define NFLAKES 500
#define XMAX 20.0
#define YMAX 20.0
#define ZMAX 10.0
#define XMIN (-20.0)
#define YMIN (-5.0)
#define ZMIN (-20.0)

#define SPEED 5.0f
/* Max navigation speed in feet per second */
/* The data that will be shared between processes */
struct _snowdata
{
float xpos;
float ypos;
float zpos;
float xaxis;
float yaxis;
float zaxis;
float spin;
float color[3];

void init_gl (void), draw (struct _snowdata *);
struct _snowdata *init_shmem (void);
void compute (struct _snowdata *);
void navigate (void);

main (int argc, char **argv)
{
    struct _snowdata *snows;
    CAVEConfigure (&argc, argv, NULL);
    snows = init_shmem ();
    CAVEInit ();
    CAVEInitApplication (initGl, 0);
    CAVEDisplay ((CAVECALLBACK) draw, 1, snows);
    while (!CAVEgetbutton (CAVE_ESCKEY))
    {
        compute (snows);
        navigate ();
        sginap (1);
    }
    CAVEExit ();
}

float randmf (void)
{
    float r, rndmax;
    rndmax = (float)(RAND_MAX + 1);
    r = rand () / rndmax;
    return r;
}

struct _snowdata *
init_shmem (void)
{  
    int i;
    struct _snowdata *snows;

    snows = (_snowdata *) CAVEMalloc (NFLAKES * sizeof (struct _snowdata));
    bzero (snows, NFLAKES * sizeof (struct _snowdata));
    for (i = 0; i < NFLAKES; i++)
    {
        snows[i].xpos = XMIN + (XMAX - XMIN) * randmf();
        snows[i].ypos = YMIN + (YMAX - YMIN) * randmf();
        snows[i].zpos = ZMIN + (ZMAX - ZMIN) * randmf();
        snows[i].xaxis = randmf();
        snows[i].yaxis = randmf();
        snows[i].zaxis = randmf();
        snows[i].color[0] = randmf();
        snows[i].color[1] = randmf();
        snows[i].color[2] = randmf();
    }
    return snows;
}

void compute (struct _snowdata *snows)
{
    int i;
    float y, angle, velocity = 0.5;
    float t = CAVEGetTime();
    static float prev_t = 0.0;
    for (i = 0; i < NFLAKES; i++)
    {
        y = snows[i].ypos - velocity * (t - prev_t);
        if (y < YMIN)
            y = YMAX;
        snows[i].ypos = y;
        angle = t * 50;
        angle = (int)angle % 360;
        snows[i].spin = angle;
    }
    prev_t = t;
void
init_gl (void)
{
float flake_size = 0.5;
GLuint tmp_indx;

disable (GL_LIGHTING);
ClearColor (0., 0., 0., 0.);

tmp_indx = glGenLists (1);
glNewList (tmp_indx, GL_COMPILE);
 glBegin (GL_POLYGON);
 glVertex3f (flake_size / 2, 0.0, 0.0);
 glVertex3f (0.0, flake_size * sqrt (3.0) / 2.0, 0.0);
 glVertex3f (-flake_size / 2, 0.0, 0.0);
 glEnd ( );
 glEndList ( );

 flake_indx = glGenLists (1);
 glNewList (flake_indx, GL_COMPILE);
 glBegin (GL_POLYGON);
 glVertex3f (flake_size / 2, 0.0, 0.0);
 glVertex3f (0.0, flake_size * sqrt (3.0) / 2.0, 0.0);
 glVertex3f (-flake_size / 2, 0.0, 0.0);
 glEnd ( );
 glEndList ( );

tmp_indx = glGenLists (1);
glNewList (tmp_indx, GL_COMPILE);
 glPushMatrix ( );
 glCallList (tmp_indx);
 glPushMatrix ( );
 glRotatef (180.0, 1.0, 0.0, 0.0);
 glTranslatef (0.0, -flake_size / sqrt (3.0), 0.0);
 glCallList (tmp_indx);
 glPopMatrix ( );
 glEndList ( );

draw (struct _snowdata *snows)
{
 int i;

 glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
 CAVENavTransform ( );
 for (i = 0; i < NFLAKES; i++)
 {
   glColor3fv (snows[i].color);
   glPushMatrix ( );
   ...
glTranslatef (snows[i].xpos, snows[i].ypos, snows[i].zpos);
    glRotatef (snows[i].spin, snows[i].xaxis, snows[i].yaxis, snows[i].zaxis);
    glCallList (flake_indx);
    glPopMatrix ();
}

void navigate (void)
{
    float wfront[3], nav;
    float jx = CAVE_JOYSTICK_X, jy = CAVE_JOYSTICK_Y, dt, t;
    static float prevtime = 0;
    t = CAVEGetTime ();
    dt = t - prevtime;
    if (fabs (jy) > 0.2)
    {
        CAVEGetVector (CAVE_WAND_FRONT, wfront);
        nav = jy * SPEED * dt;
        CAVENavTranslate (wfront[0] * nav, wfront[1] * nav, wfront[2] * nav);
    }
    if (fabs (jx) > 0.2)
        CAVENavRot (-jx * 90.0f * dt, 'y');
    prevtime = t;
}

You can go away from the snow falling area by moving the joystick of the wand [Fig. 14]. Another major change applied to this program is the color of the snowflakes.

Notice that _NAV_ has to be attached to the “pid” and “vid” in the previous subsection to get the positions and the directions of the glasses and wand in the navigated coordinate.
Introduction to Virtual Reality Visualization by the CAVE system

3.7 Summary of this section

So far, we introduced 15 CAVElib functions and 2 kinds of macros which are summarized in Table 7. Those would be minimum set to make basic CAVE applications. If you want to know more about CAVElib, see CAVElib User and Reference Guide\(^7\). Our CAVE programming guide [Kageyama:1998] written in Japanese is available in our web site\(^8\).

4 Virtual LHD

Based on the CAVElib explained in the previous section and OpenGL CG techniques, one of the authors (A. Kageyama) developed Virtual LHD program [see Figs. 15 and 16]\(^9\). The LHD (Large Helical Device) is a fusion experiment device at National Institute for Fusion Science (NIFS), Japan. Mimicking the real LHD, Virtual LHD has toroidal and poloidal coils and vacuum vessel. The viewer in the CAVE can walk through the virtual LHD which is shown in the real scale size. The reality is so high that many people experiencing this program for the first time stretched their arms trying to touch the coils. Virtual LHD program reads MHD equilibrium data calculated by HINT code [Harafuji:1989] which is one of the standard codes for the MHD equilibrium of helical systems. One sees magnetic surfaces, field lines, particle orbits, pressure isosurfaces in the CAVE. One can interactively change the isosurface level by “touching” a virtual level bar in the CAVE. Details of the Virtual LHD program was described in [Kageyama:1998].

Although this software was very useful, it could be used only for analyzing LHD simulation data. We concluded that making a CAVE software for each simulation,

---

\(^{(7)}\)http://www.vrco.com/


Table 7. CAVELib Functions and Macros

<table>
<thead>
<tr>
<th>Functions and Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Functions</strong></td>
</tr>
<tr>
<td>void CAVEConfigure(int *, char **, char **)</td>
</tr>
<tr>
<td>CAVEInit()</td>
</tr>
<tr>
<td>void CAVEInitApplication(CAVECALLBACK, int, ...)</td>
</tr>
<tr>
<td>CAVEDisplay(CAVECALLBACK, int, ...)</td>
</tr>
<tr>
<td>void CAVEExit()</td>
</tr>
<tr>
<td>int CAVESetOption(CAVEID, int)</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
</tr>
<tr>
<td>void *CAVEMalloc(size_t)</td>
</tr>
<tr>
<td><strong>Tracking</strong></td>
</tr>
<tr>
<td>void CAVEGetPosition(CAVEID pid, float v[3])</td>
</tr>
<tr>
<td>void CAVEGetVector(CAVEID vid, float v[3])</td>
</tr>
<tr>
<td><strong>Wand</strong></td>
</tr>
<tr>
<td>int CAVEButtonChange(int)</td>
</tr>
<tr>
<td>CAVE_BUTTONn, n = 1, 2, 3</td>
</tr>
<tr>
<td>CAVE_JOYSTICK_x, x = X, Y</td>
</tr>
<tr>
<td><strong>Navigation</strong></td>
</tr>
<tr>
<td>void CAVENavTranslate(float, float, float)</td>
</tr>
<tr>
<td>void CAVENavRot(float, char)</td>
</tr>
<tr>
<td>void CAVENavTransform()</td>
</tr>
<tr>
<td><strong>Others</strong></td>
</tr>
<tr>
<td>float CAVEGetTime()</td>
</tr>
<tr>
<td>boolean CAVEgetbutton(CAVE_DEVICE_ID)</td>
</tr>
</tbody>
</table>

Fig. 15. Helical coils and a pressure isosurface. By touching a level bar (not seen) by the wand, the isosurface level is interactively controlled. The wand is a portable controller with buttons.

one by one, was not productive. So, we started development of a general-purpose VR visualization software called VFIVE for the scientific VR visualization in the CAVE systems.
Fig. 16. A magnetic field line. The field line tracing starts from the wand tip position when a button is pressed.

5 General-Purpose CAVE Visualization Software VFIVE

The development of VFIVE\textsuperscript{10} started around 1999 [Kageyama:2000] and the latest stable version is v3.7. The present development version with major revise is v3.8. The very final version of VFIVE will be v5.

Fig. 17. Interactive menu of our original VR visualization software VFIVE.

5.1 VFIVE v3.7

The basic design and core functions of VFIVE are almost fixed in v3.7, which are summarized in Table 8. A great emphasis was placed on fully interactive control of the visualization methods, data, and the user interface in the CAVE’s VR environment. For example, one can control the isosurface levels by vertical motion of the hand with the wand; it is also possible to change the slice position of orthoslicer by the wand’s motion. From our experience of using the CAVE as a scientific visualization tool from the end of 1990’s [Kageyama:1999], we see that a common pitfall is to

\textsuperscript{10}http://www.es.jamstec.go.jp/esc/research/Solid/members/kage/vfive/index.html
use it just as a (very expensive) “3-D object viewer”. We should make the best use of 
CAVE’s VR features; that is stereo, immersive, and especially interactive view.

We implemented several kinds of visualization methods for vector fields. All of 
them make use of the VR features of the CAVE. For example, vector arrows shows 
tens of small arrows around your hand, within an invisible sphere of diameter of 
2 feet. As you move your hand, all the arrows follow your hand’s motion. The 
length and direction of each arrow indicates the vector at the sampled point. The 
interpolation is automatically applied in real time.

<table>
<thead>
<tr>
<th>Table 8. VFIVE v3.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
</tr>
<tr>
<td>user interface</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>scalar field vis.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>vector field vis.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Fig. 18. Scalar field visualization by isosurface, and orthoslicer.
Fig. 19. Vector field visualization by tracer particles. When you press a wand button in the CAVE room, a tracer particle appears at the tip of the wand. You can intuitively and directly decide the starting positions of the tracers by controlling the wand's (or your hand's) position.

Fig. 20. Interactive vector field visualization by arrows with the isosurface rendering. The arrows change the length and direction, following your hand's motion in real time with the automatic interpolation of the vector field.

5.2 VFIVE v3.8

Recently, we have improved the VFIVE in two major aspects: First, we included an important scalar visualization method that was missing in the previous version; volume rendering [Drebin:1988]. Another improvement is that we have integrated some basic modules of VTK into VFIVE.
The volume rendering is a visualization method of scalar fields that is useful, for example, to show the distribution of plasma density $\rho(x, y, z)$ of the 3-D global simulation of the magnetosphere. It is known that $\rho$ is highly localized near the Earth. The isosurface is not useful for the visualization of this kind of scalar field; you need to try nearly infinite number of different isosurface levels to grasp the overall 3-D distribution of $\rho$. Similarly, the orthoslicer is not useful because one needs to try a lot of slices to grasp 3-D distribution of $\rho$. The volume rendering fits this kind of visualization of 3-D scalar field. We have developed a volume rendering module of VFIVE and slotted into v3.8.

Generally, the volume rendering is considered as a “heavy” visualization technique for the CAVE-type VR systems since it requires millions of ray cast calculation per second to realize the real time response to viewer’s eyes motion in the CAVE room. We succeeded in realizing a very fast volume rendering by the 3-D texture-map technique [Schroeder:2002]. In this technique, many semi-transparent texture mapped slices, which are orthogonal to the viewer’s line of sight, are pulled out from the 3-D volumetric data. The 3-D volumetric data is made in advance from the scalar field (such as $\rho$) by specifying the colors and opacities of each box element (voxel). The texture slices are blended by the graphics card. Therefore, we can avoid the heavy calculations of software ray casting. This is the reason why we can carry out the fast volume rendering in the CAVE. For example, the refresh rate of the volume rendering shown in Fig. 21 (vorticity distribution of Earth’s outer core) is several frames per second, which is satisfactory for our visualization purpose.

The VTK (Visualization Tool Kit)\footnote{see also \url{http://public.kitware.com/VTK/}} is an general visualization software that includes many visualization methods. It is an attractive idea to combine VFIVE’s interactive visualization environment in the CAVE with VTK’s sophisticated visualization gismos. There are several previous works [Rajlich:1998, Hall:1999, Shamonin:2002] to use VTK in the CAVE. Among them, we follow the idea of vtkActorToPF [Rajlich:1998]. A difference from the original vtkActorToPF is that we do not use OpenGL Performer. We have revised VFIVE that the Application Computation Process computes the polygonal data by VTK and send them to the Display Processes via the hard disk drive, and the Display Processes receive and draw the data. In this way we have succeeded to integrate some basic visualization methods of VTK into
Fig. 21. Real time volume rendering implemented in VFIVE v3.8 with arrows. Compare with isosurface rendering in Fig. 20. The high speed volume rendering is realized by the 3-D texture-map technique.

Fig. 22. Contour lines by VTK in VFIVE v3.8. The pressure of MHD simulation of the Earth’s magnetosphere is visualized.

VFIVE v3.8. For instance, we can show VTK’s contour lines [Fig. 22], tubed flow lines [Fig. 23], stream surfaces [Fig. 24] in the CAVE (see Table 9).

6 Summary
Since the output data of 3-D computer simulation today are highly complicated, the visualization should be performed in a 3-D, interactive, and immersive environment. Modern VR technology provides us such an opportunity. Among various VR systems available today, we believe that the CAVE system is superior to others for our purpose. As we explained in this chapter, developing a CAVE application program is
Fig. 23. Stream tube by VTK in VFIVE v3.8. The tube’s color changes according to local amplitude of the vector field. The tube’s diameter can also be changed by the amplitude if one wishes. Here, the flow velocity of the solar wind is visualized. The geomagnetic field is also visualized by the (not tubed) field lines.

Fig. 24. Stream surfaces by VTK in VFIVE v3.8. The color changes according to the local amplitude of the vector field. The flow velocity of the solar wind is visualized. The pressure is also visualized by contour lines.

not difficult owing to the sophisticated CAVElib. Our sample programs will provide a good starting point for your own visualization software for the CAVE.

The main obstacle for popularization of the CAVE has been its cost. However, recent development of the graphics hardware for PCs is drastically changing the situation. Inexpensive PC-based CAVE systems will predominate over more expensive GWS-based CAVEs in near future.

We speculate that every supercomputer center in the world will install the CAVE-
type VR facility for the VR visualization in future, and use the software like VFIVE for practical and productive simulation science.

Acknowledgments. We thank all members of Advanced Perception Research Group of Earth Simulator Center for their help in the development of VFIVE v3.8 and Prof. Tatsuki Ogino at Nagoya Univ. and Dr. Naoki Mizuguchi at NIFS for providing us their MHD simulation data. We also thank Prof. Tetsuya Sato, the director-general of the Earth Simulator Center, for fruitful discussion and encouragement.

Appendix A. VTK in CAVE

When you want to make a visualization software for the CAVE without taking much time to study the visualization algorithms and methods, we suggest to use VTK in CAVE.

VTK is an general visualization software that includes many visualization methods, from basic ones to sophisticated ones, with open source codes. The usage of VTK is very simple. All you have to do is to make “VTK Pipeline”. The following code is a part of VTK pipeline implementing “contour lines”.

```cpp
1 Array = vtkDoubleArray::New();
2
3 x_coord = vtkDoubleArray::New();
4 y_coord = vtkDoubleArray::New();
5 z_coord = vtkDoubleArray::New();
6
7 Rectidata = vtkRectilinearGrid::New();
8
9 Lut = vtkLookupTable::New();
10 CLines = vtkContourFilter::New();
11 Mapper = vtkPolyDataMapper::New();
12 Actor = vtkActor::New();
13
14 Array -> SetArray(elev_data, size, 1);
15
16 x_coord -> SetArray(coorddata->x, n1, 1);
17 y_coord -> SetArray(coorddata->y, n2, 1);
18 z_coord -> SetArray(&z, 1, 1);
19
20 Rectidata -> SetDimensions(n1, n2, 1);
21 Rectidata -> SetXCoordinates(x_coord);
22 Rectidata -> SetYCoordinates(y_coord);
23 Rectidata -> SetZCoordinates(z_coord);
24
25 Rectidata -> GetPointData() -> SetScalars(Array);
26
27 Lut -> SetHueRange(0.7, 0.0);
28 Lut -> Build();
29```
This VTK pipeline visualizes the data, whose size is \( n_1 \times n_2 \) and pointer is "\texttt{elev\_data}" [line 14], by rainbow-colored 30 contour lines. The \( z \)-component of the coordinate of the lines is set to be "\( z \)". Looking at the code, you realize that with VTK, you can make pretty good visualization software without the knowledge of the underlying algorithms of the visualization methods. However, using VTK in a CAVE application is not obvious because of VTK’s design. Some people work out the ways to do it. “\texttt{vtkActorToPF}” is a famous translator and the detail design is described in the paper [Leigh:1998] but this requires your CAVE application to be written on IRIS Performer (or OpenGL Performer). “\texttt{vtk2CAVE}” is a translator which does not require IRIS Performer. This software draws the polygonal data of the \texttt{vtkActor} copied on the shared memory by OpenGL. “\texttt{VtkCave}” is a library which enables you to make CAVE applications with VTK. You can choose the translators and library depending on API or make a translator by yourselves if you want to use VTK in your CAVE applications.

References


http://brighton.ncsa.uiuc.edu/˜prajlich/vtkActorToPF/


http://staff.science.uva.nl/˜dshamoni/myprojects/VtkCave.html

