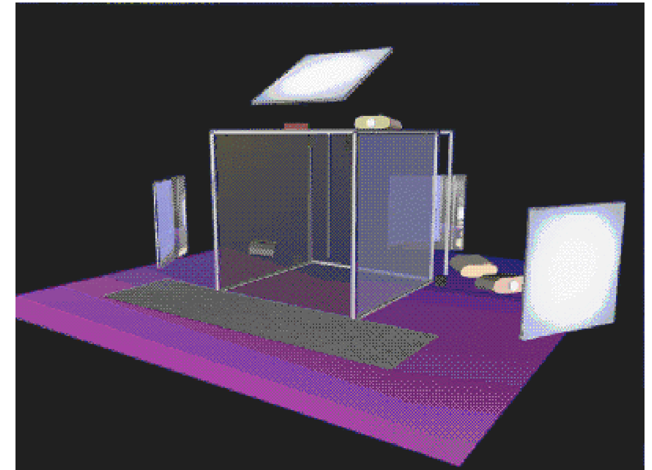
Virtual Reality

What Is Virtual Reality (VR)

- The term “Virtual Reality” was coined in 1989 by Jaron Lanier, founder of VPL (virtual programming language) research
- Virtual Reality (VR) can be defined as interactive computer graphics that provides viewer-centered perspective, large field of view and stereo
- User becomes fully immersed in an artificial, three-dimensional world that is completely generated by a computer
- Also known as Virtual Environment (VE)
- The Electronic Visualization Laboratory (EVL) of the University of Illinois at Chicago has specialized in projection-based VR systems. EVL's projection-based VR display, the CAVE, premiered at the SIGGRAPH '92 conference
  - CAVE, ImmersaDesk and InfinityWall

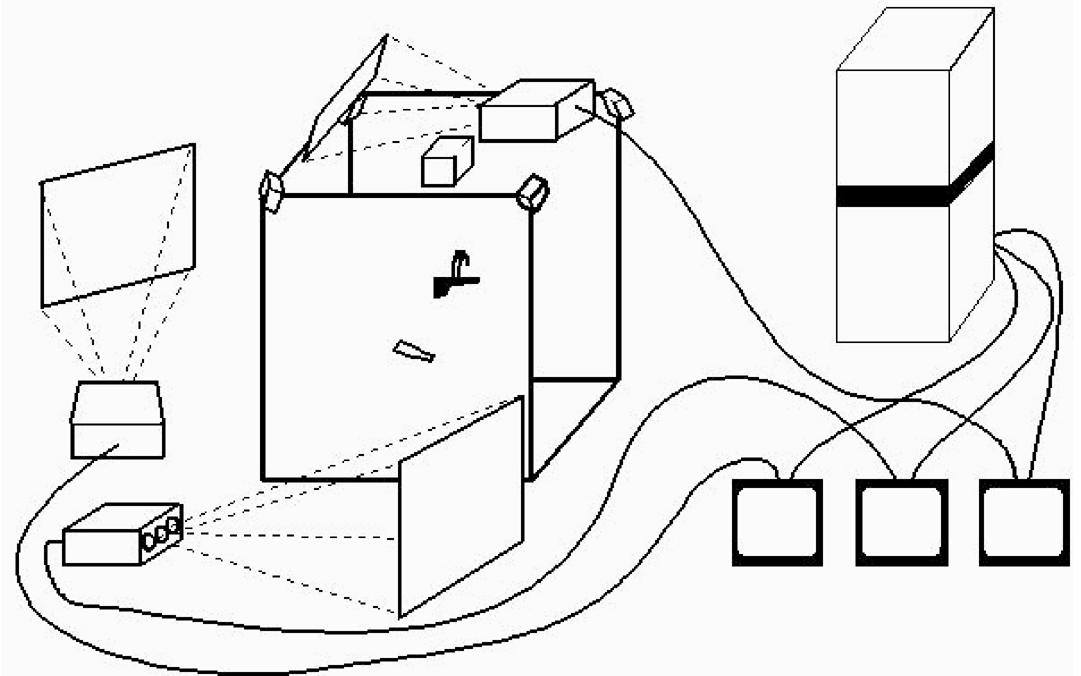
CAVE

CAVE: Cave Automatic Virtual Environment:

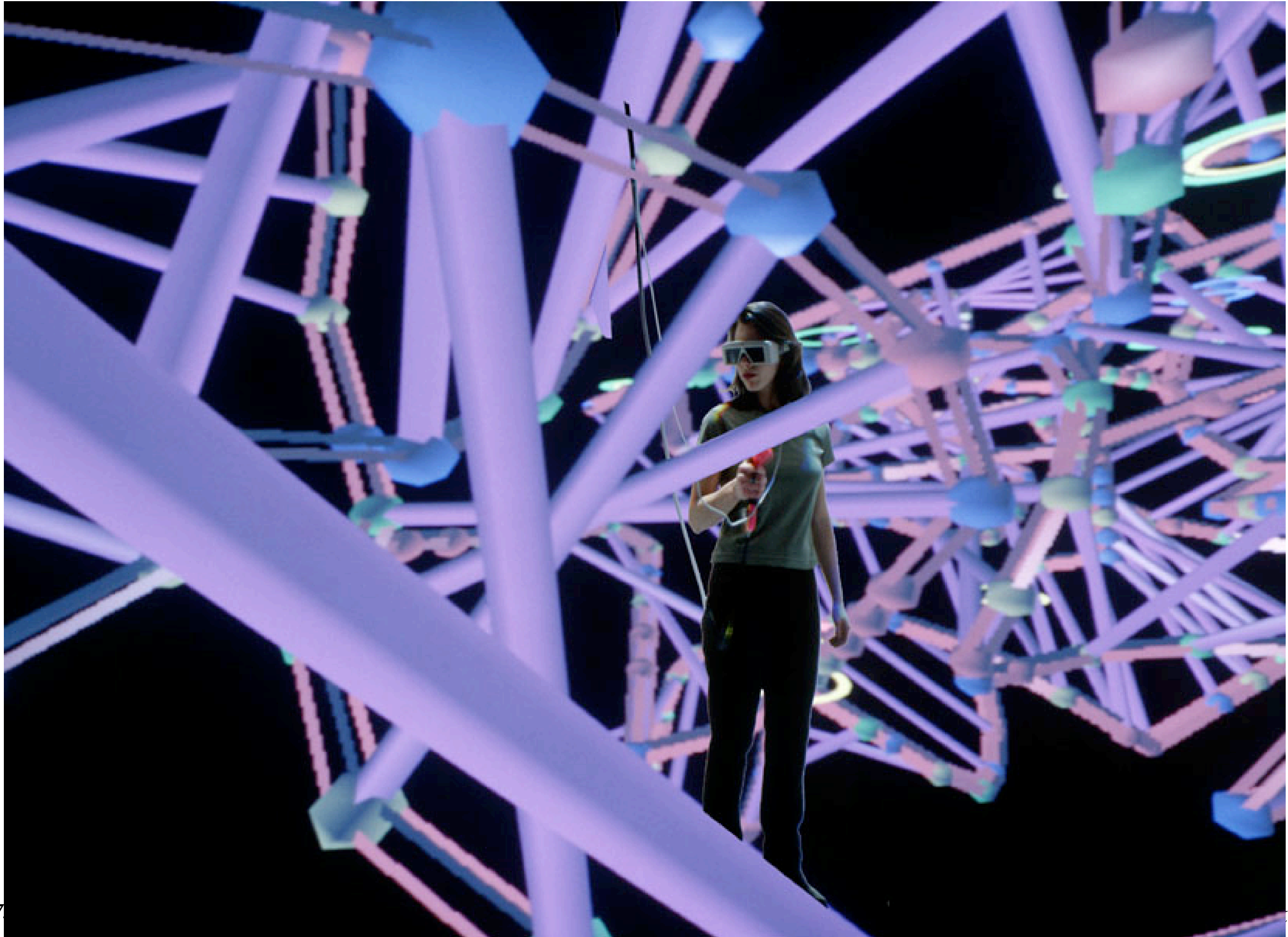


- Immersive and interactive virtual environment
- Projection-based VR system that surrounds the viewer with 4 screens (blocking the outside world)

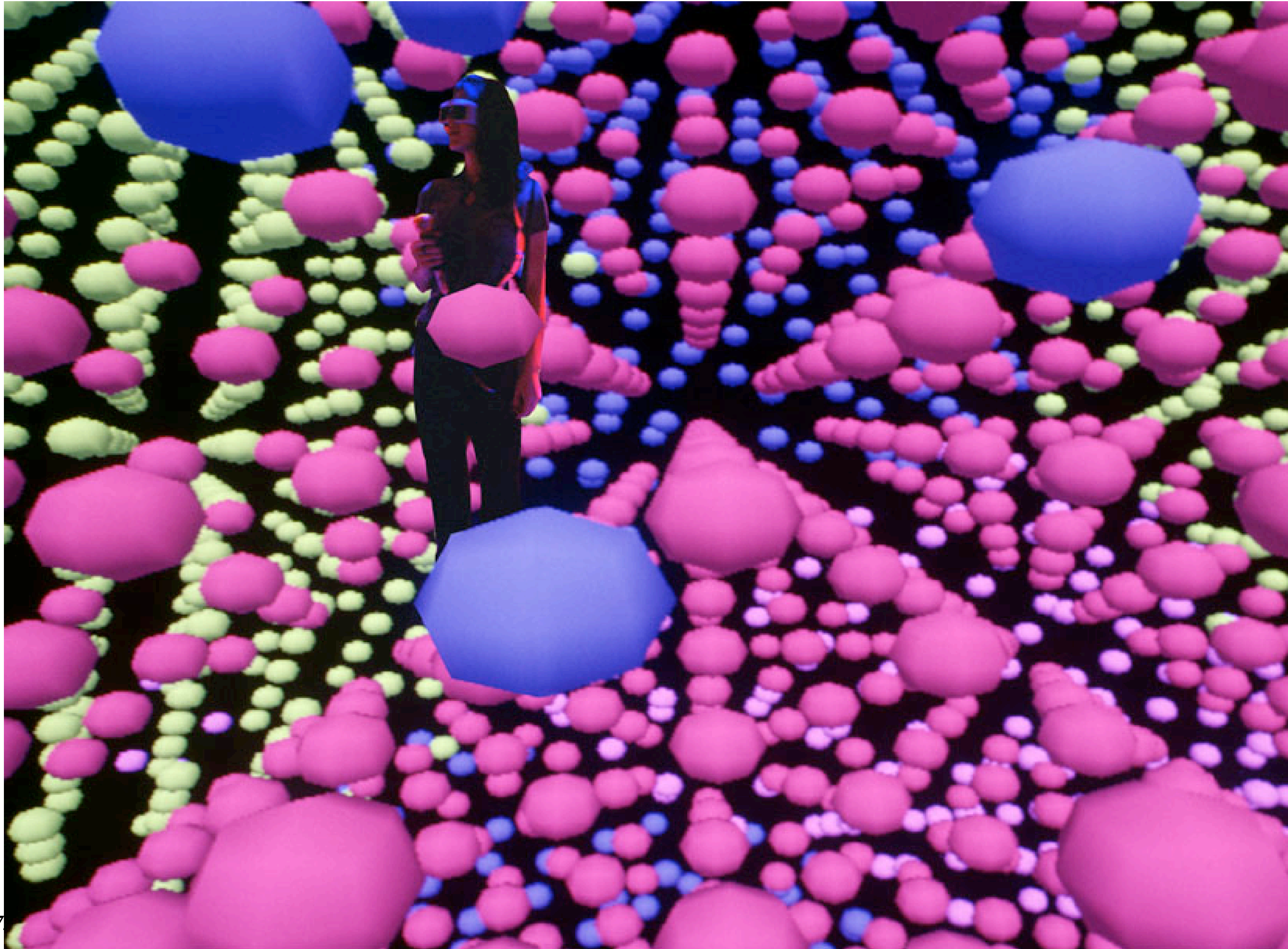
- Screens are arranged in a cube with three rear-projection screens for walls and a down-projection screen for the floor
- Viewer wears stereo shutter glasses and carries a six-degrees-of-freedom head-tracking device
- A second sensor and buttons in a wand to allow interaction with VR
- Correct stereoscopic perspective projections are calculated for each wall
- Any number of walls can be used



Inside DNA Molecule



Inside Nanocrystal



ImmersaDesk

Is a drafting table format virtual prototyping device

ImmersaDesk R2:

A 75.5 to 82.5 inches diagonal high resolution screen with adjustable screen angle (43 to 89 degree)

ImmersaDesk M1:

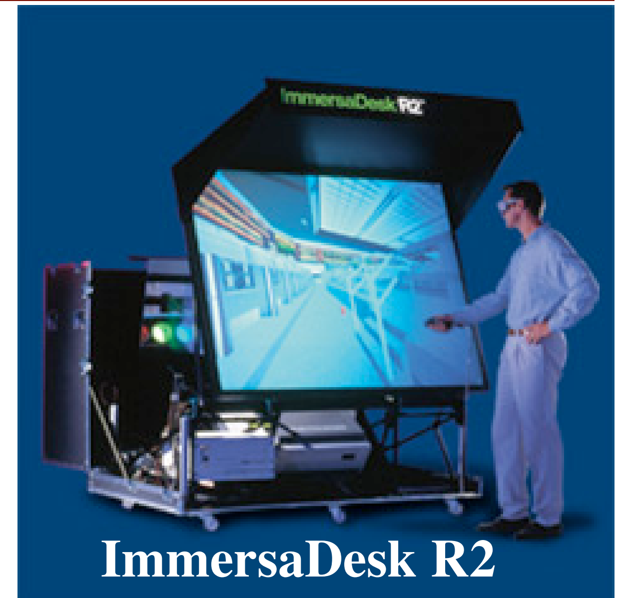
A 44 inches diagonal high resolution screen with adjustable desktop angle (15 to 60 degree)

LSU has an ImmersaDesk M1

Tilted rear-projected screen

Provides a semi-immersive and interactive virtual environment

Except for the display device, the same hardware and software are used as in the CAVE



ImmersaDesk R2

ImmersaDesk M1

Infinity Wall

Larger scale system designed around the same basic resources as the CAVE

Intended for presentations to large groups

Comprises a single 9x12 foot (or larger) screen, four projectors which tile the display, one or two SGI Onyxes to drive the display

First demonstrated at the Supercomputing '95 conference.

Useful Websites

- CAVE was developed by University of Illinois at Chicago

- CAVE programming at Electronic Visualization Laboratory (EVL) at University of Illinois at Chicago

  `www.evl.uic.edu/pape/CAVE/prog`

- Fakespace Systems (the vendor of CAVE and ImmersaDesk)

  `www.fakespacesystems.com`

Hardware

- CAVE hardware should be configured by system and video engineers
- CAVE users are concerned with turning on and off different components
- Cave equipment includes
  - Projectors and Mirrors
  - Stereo Glasses
  - Stereo Emitters
  - Wand
  - Tracking Systems
  - Audio System
  - Workstations

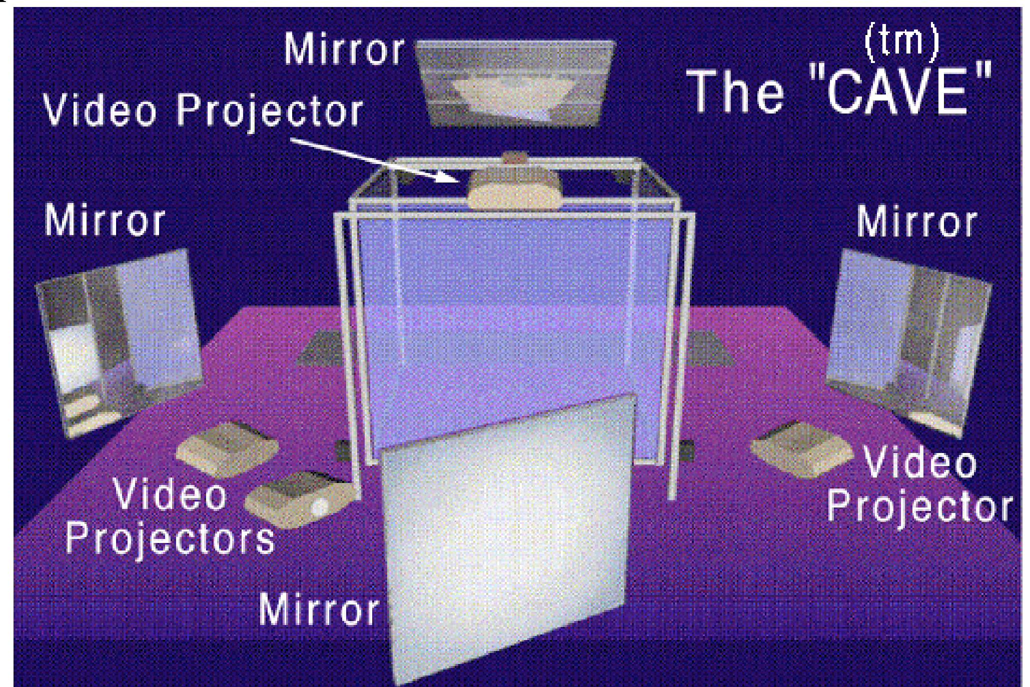
Projectors and Mirrors

- Projectors and mirrors for the side wall are located behind each wall. The projector for the floor is suspended from the ceiling of the cave; and it points to a mirror which reflects the image onto the floor.

- The projectors are very sensitive to almost everything.

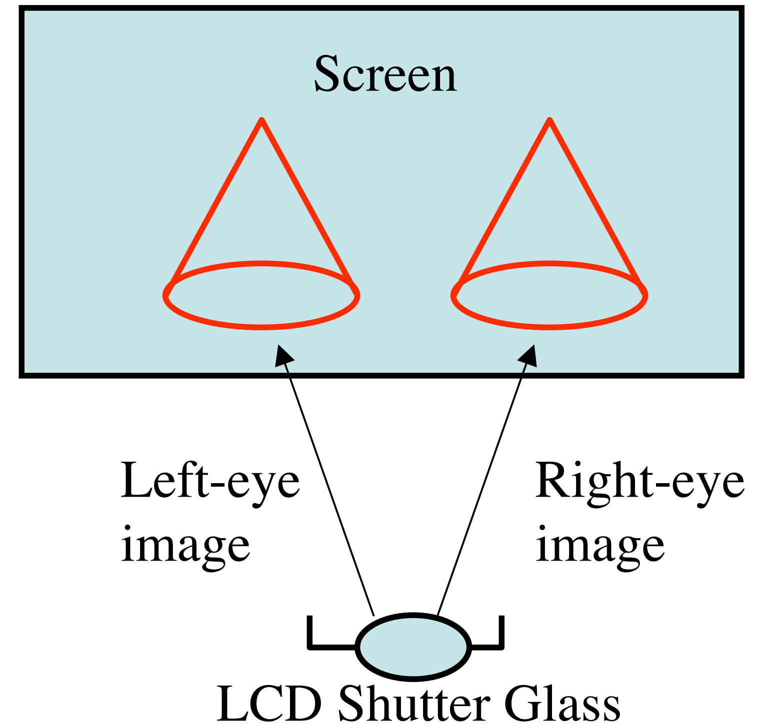
- It takes time to align and calibrate each projector and mirror to match the corners of the CAVE, and achieve the RGB convergence

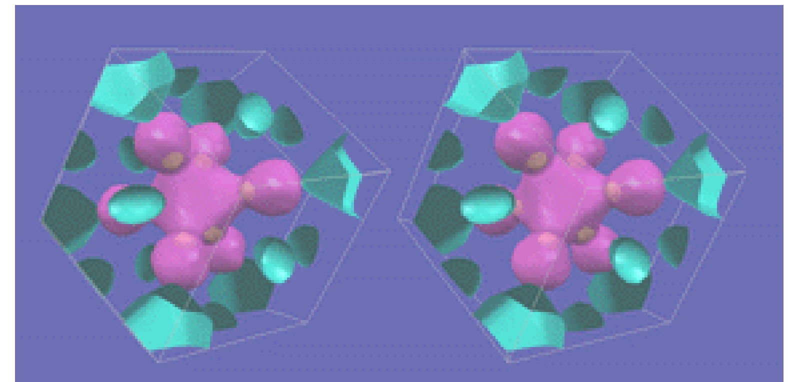
- Never turn off the projectors; this can cause them to go out of alignment (put always them on standby). Standby mode also extends the life span of the projector tubes.



Stereo Glasses

- Stereographics' CrystalEyes liquid crystal shutter glasses:
  - To see the VE in stereoscopic 3D
  - Glasses are very fragile
  - Turn on glasses by pressing a small button located on the right side of frame
  - They will not work if the user is facing away from the emitters.
- Stereographics:
  - The projector interleaves images for left and right eyes at a rate of 120 frames/sec
  - Glass is synchronized via an infrared emitter with the projector so that the left eye sees only one set of images (60 times a second) and the right eye sees the other set (60 times a second)
  - 3D perception is created by showing the two eyes slightly rotated objects





www.phys.ufl.edu/fermisurface/

Stereo Emitters

- Devices that synchronize the stereo glasses to the screen update rate of 120Hz

- Little white boxes placed around the edges of the CAVE or on either side of the projector of ImmersaDesk

- They are always on

- You should not have to do anything with them

Wand

- A 3D mouse with buttons for the interactive input
- The wand has three buttons and a pressure-sensitive joystick. It is connected to the CAVE or ImmersaDesk through a PC which is attached to one of the graphics engine's serial ports
- A server program on the PC reads data from the buttons and joystick and passes them to the graphics engine
- Sensor is located at wand

Tracking System

- A user interacts with a CAVE application using tracker and controller

- Tracker reports the position and orientation of its sensors
  - Up to 8 sensors can be supported
  - The first sensor is always for tracking the user's head, the second for the wand, and others for whatever other objects are being tracked

- Controller (wand) consists of a set of buttons which can be on or off, and joystick

- The CAVE supports several different tracking systems:
  - Electromagnetic system

    Ascension Technologies Flock of Birds

    Ascension Spacepad
  - Acoustic system

    Logitech sonic tracker

Audio System

- The audio system components are
  - Indy workstation

    It functions as sound server
  - Speakers
  - MIDI (Musical instrument digital interface)
  - Synthesizer

Workstation

- The CAVE runs using a Silicon Graphics Onyx with three Reality Engine. Each Reality Engine is attached to a CAVE wall.

- 16 processors Onyx 3400 with InfinityReality3 graphics engine

- A Silicon Graphics Cluster runs ImmersaDesk M1 at LSU
  - A single master node (1 GHz Pentium III processor with 1 GB RAM)
  - Two visual channel nodes
  - Linux 7.1 operating system

CAVE Library

- A library of C functions and macros to control the operation of the CAVE (all files normally located in /usr/local/CAVE)

- It takes care of all the tasks that have to be performed to correctly operate the CAVE

- CAVE functions to
  - Perform any window or projection commands
  - Set in RGB mode, double buffered and z-buffering
  - Keep all the devices synchronized
  - Produce the correct perspective for each wall
  - Keep track of which walls are in use

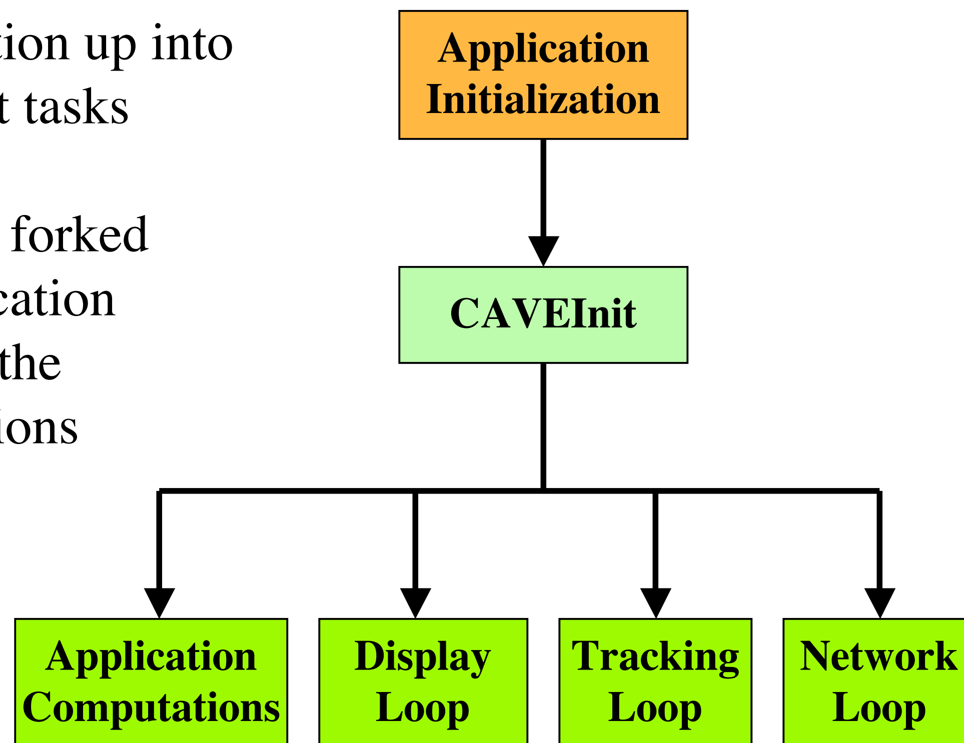
- NOTE: The names of all CAVE functions, macros, and global variables start with the word CAVE (*CAVEDisplay*, for example)

Multiprocessing

- The CAVE library splits an application up into several processes to handle different tasks

- The different child processes are all forked by *CAVEInit*. Only the parent application process will return from *CAVEInit*; the others all start internal library functions

- There is one display process per active wall, one process for tracking and one process for networking (if networking is enabled)



- To maintain acceptable frame rates, display processes perform only rendering while all computations are done in parallel in the main application process

Display Callbacks

- Graphics in a CAVE program are handled using callback functions which are called by the CAVE library's display loop for each view that must be rendered

- Each process will call the application's display function twice per frame in a stereo mode or once in monoscopic
  - By passing a function pointer to *CAVEDisplay*
- A frame function callback is called exactly once per frame in each rendering process, before the display callback
  - By defining with *CAVEFrameFunction*
- Initialization callback is called once at the beginning of the next frame after it is defined (for material properties)
  - By using *CAVEInitApplication*

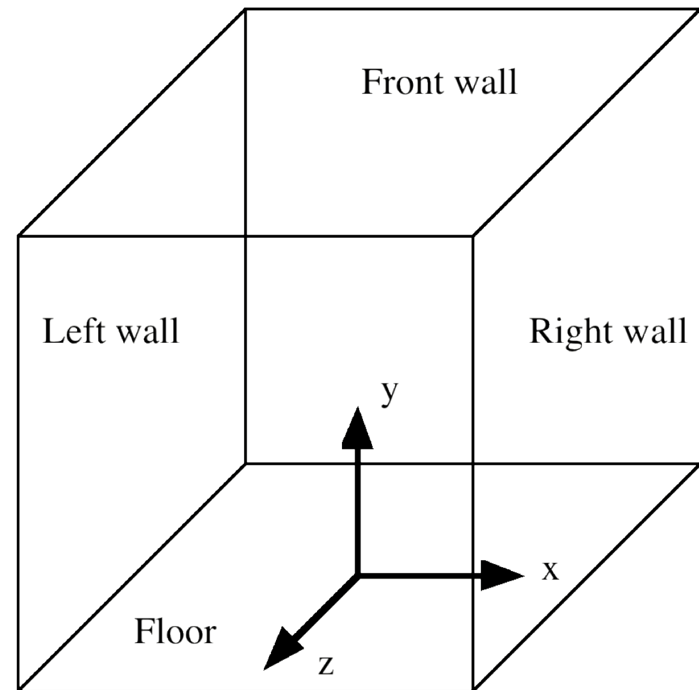
CAVE Coordinate System

The standard CAVE is a 10 foot cube.

The origin of the coordinate system for the CAVE is normally located at the center of the floor, that is, 5 feet away from any wall.

This means that you have from +5 to -5 feet horizontally and 0 to 10 feet vertically to define objects inside the CAVE.

By default, the near and far clipping planes are located at 0.1 and 100.0 feet



Navigation

- The CAVE structure and tracking hardware generally limit a user's movement to a 10 foot square area or smaller
- Needs a navigation coordinate transformation that can move the CAVE's physical coordinate system around in the virtual space
- The CAVE library maintains a navigation transformation matrix and provides conversions between the tracker (physical) and world (navigated) coordinate systems
- The basic functions for navigation are *CAVENavTranslate*, *CAVENavRot*, and *CAVENavScale*
- These are equivalent to the corresponding OpenGL functions

Basic CAVE Functions

- *CAVEConfigure(int *argc,char *argv,char*appdefaults);*
  - Initializes the CAVE configuration
  - Internal shared memory arena is created
  - Various global variables are initialized
  - Configuration files are read
  - Any user-specified configuration options given in *appdefaults* or *argc/argv* are set

- *CAVEInit(void);*
  - Initializes the CAVE environment.
  - Starts rendering processes, and initializes the trackers and graphics
  - After this function is called, the rendering processes are separated from the main computation process

Basic CAVE Functions

- *CAVEInitApplication(CAVECALLBACK function,int num_args, ….);*
  - Passes the CAVE library a pointer to your graphics initialization routine function
  - This routine does any GL initialization that is required for display
  - The rendering processes will call this only once, at the beginning of the next frame
  - *CAVEInitApplication* blocks until the next swap-buffers call by the rendering processes
  - *CAVEInitApplication* should be called after CAVEInit, and before *CAVEDisplay*

  First argument is a pointer to the GL initialization routine

  Second argument is the number of arguments that your routine receives

  Remainders are the arguments to be passed to your routine. The information is passed to the rendering processes.

Basic CAVE Functions

- *CAVEDisplay(CAVECALLBACK function, int num_args,…);*
  - Passes the CAVE library a pointer to your drawing routine
  - Rendering processes will call your routine twice per frame for stereoscopic mode or once for monoscopic mode
  - All rendering should be done from this routine
  - *CAVEDisplay* blocks until the next swap-buffers call by the rendering processes
  - *CAVEDisplay* can only be called after *CAVEInit*

- *CAVEFrameFunction(CAVECALLBACK function, int num_args,…);*
  - Gives the library a pointer to a routine which should be called once per frame
  - This routine will be called exactly once per frame (for both mono and stereo modes)
  - It is called before initialization and display routines but after *CAVEInit*

- *CAVEExit(void);*
  - Exits the CAVE and restores the machine to its normal state

CAVE Macros and Variables

- Macros
  - Sensor micros

    CAVESENSOR(i) A pointer to ***i*** th sensor

    CAVEGetSensorPosition, CAVEGetSensorOrientation
  - Controller micros

    CAVEBUTTONn=[0|1] 1 means the button is pressed

    CAVE_JOYSTICK_X Give the x and y coordinate values

    CAVE_JOYSTICK_Y of joystick

- Global Variables

  Int CAVENear, CAVEFar Near and far clipping plane distances

  Int CAVEEye Eye view currently drawn (Left or right)

  Int CAVEWall Wall currently being drawn (Front, Left,..)

Form of a Basic CAVE Program

```
#include <cave.h>

void app_shared_init(), app_compute_init(),
    app_init_gl(), app_draw(), app_compute();

main(int argc,char **argv)
{
CAVEConfigure(&argc,argv,NULL);
app_shared_init(argc,argv);
CAVEInit();
CAVEInitApplication(app_init_gl,0);
CAVEDisplay(app_draw,0);
app_compute_init(argc,argv);
while (!getbutton(ESCKEY))
              app_compute();
CAVEExit();
}
```

Meaning of Different Terms

- *CAVEConfigure():* This routine reads the CAVE configuration file, and parses argc/argv for any user-specified configuration options.
- *app_shared_init():* This initializes anything that will be shared by the computation and rendering processes (allocating shared memory, etc.).
- *CAVEInit():* This routine initializes the CAVE. It forks several processes. One process (the "computation process") returns from CAVEInit and runs the rest of main(). The other processes handle the tracking and rendering.
- *CAVEInitApplication(), app_init_gl():* A pointer to the application's graphics initialization function is passed to the rendering process. Since the rendering is not done by the computation process, but by a separate rendering process, any GL initialization needed for the rendering cannot be done directly by the computation process. Instead, this function sets a pointer in shared memory to tell the rendering process what function to call.
- *CAVEDisplay(), app_draw():* A pointer to the application's drawing function is passed to the rendering process. As in the CAVEInitApplication routine, this sets a pointer in shared memory to the function. The rendering process then sees this pointer and calls the function itself.
- *app_compute_init():* This initializes any non-shared data that will be used by the computation process.
- *app_compute():* This performs the application's computations. Any results that are used by the drawing function should be stored in shared memory.
- *CAVEExit():* This causes all CAVE processes to exit and restores the machine to its normal state.

CAVE Sample Program

```
/* ball.c :: It demonstrates the most basic CAVE library functions.
This program just draws a red ball in the front of the CAVE. No
interaction (outside of moving around), and nothing changes.*/

#include <cave_ogl.h>
#include <GL/glu.h>

void int_gl(void), draw_ball(void);

static GLUquadricObj *sphereobj

main(int argc,char **argv)
{
 CAVEConfigure(&argc,argv,NULL); /* Initialize the CAVE */
 CAVEInit();
 CAVEInitApplication(init_gl,0); /* a pointer to GL initialization  */
                                 /*   function */
 CAVEDisplay(draw_ball,0);       /* a pointer to drawing function  */
 while (!CAVEgetbutton(CAVE_ESCKEY))/* Main loop continues until */
    sginap(10);                    /* until the escape key is hit */
 CAVEExit();                        /* Clean up & exit */
}
```

init_gl()

```
/* init_gl - GL initialization function. This is called exactly once
by each of the drawing processes, at the beginning of the next frame
after the pointer to it is passed to CAVEInitApplication.
It defines the lighting and material data for the rendering and
creates a quadric object to use when drawing the sphere */

void init_gl(void)
{
  float redMaterial[] = {1,0,0,1];

/* Enable one light source */
  glEnable(GL_LIGHT0);

/* Set material to color both faces to a diffuse red */
  glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,redMaterial);

/* Create a glu quadratic object */
  sphereObj = gluNewQuadric();
}
```

draw_ball()

```
/* draw_ball - display function. This is called by the CAVE library
in the rendering processes' display loop. It draws a ball 1 foot in
radius, 4 feet off the floor, and 1 foot in front of the front wall
(assuming a 10' CAVE). */

void draw_ball(void)
{
/* Set clear color to black and clear both screen and z-
buffer */
  glClearColor(0,0,0,0);
  glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);
/* Turn lighting on */
  glEnable(GL_LIGHTING);
/* Draw a sphere of radius=1 foot, slices=8, and stacks=8 */
  glPushMatrix();
    glTranslatef(0.0,2.0,-3.0);
    gluSphere(sphereObj, 1.0, 8,8);
  glPopMatrix();

  glDisable(GL_LIGHTING);
}
```

Compiling a CAVE Program

- A CAVE program should include the appropriate CAVE header file - cave_ogl.h for OpenGL programs.
- OpenGL programs will need to be linked with the OpenGL CAVE library, the OpenGL library, the math library, and the X libraries (*-lcave_ogl -lGL -lX11 -lXi -lm*).
- The following `Makefile` is used to compile a CAVE application, `sample.c`:

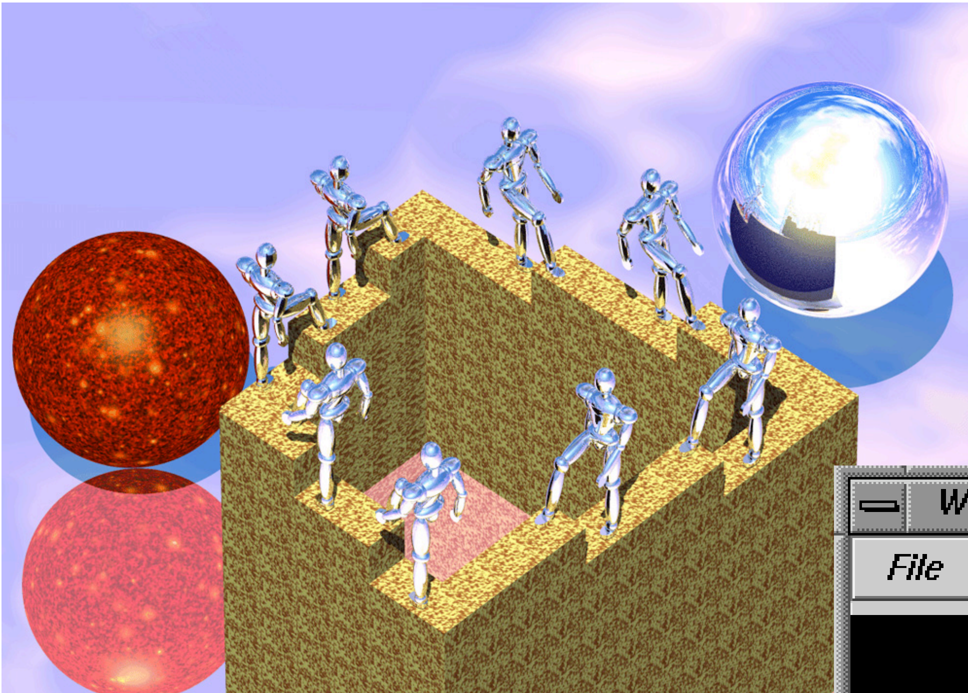
```
LIBS = -L/usr/local/CAVE/lib32 -
   lcave_ogl -lGLU -lGL -lXi -lX11 -lm
sample: sample.o
cc -O -o sample sample.o $(LIBS)
```

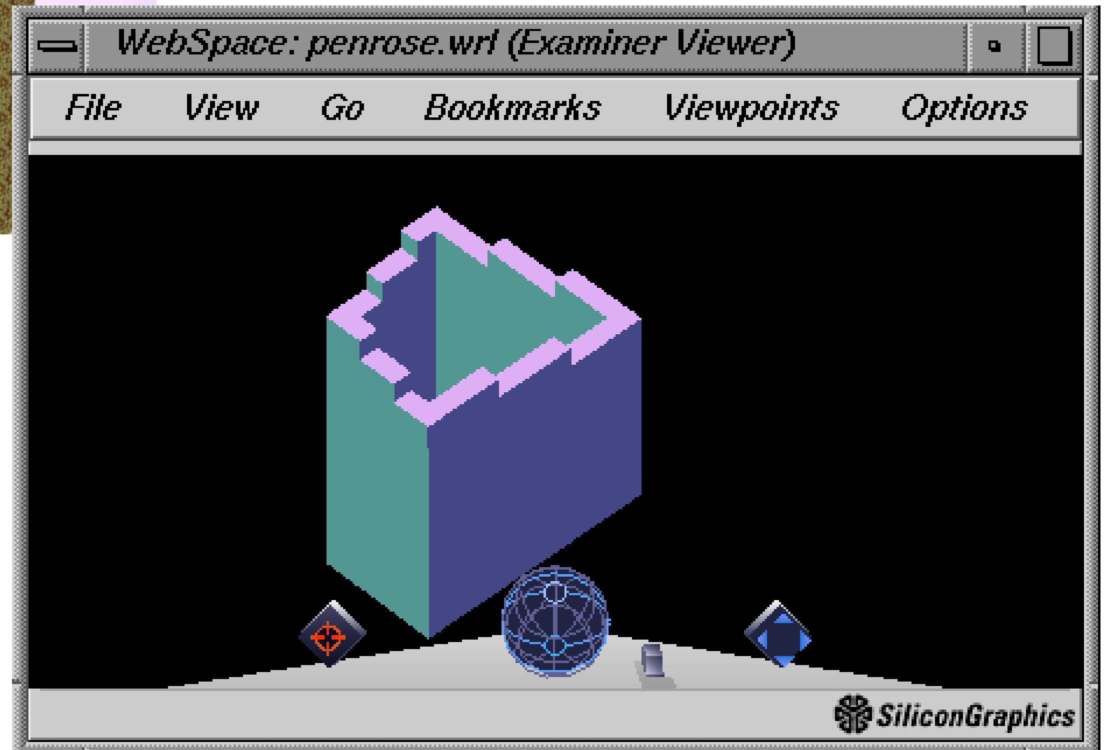
Virtual Reality Modeling Language: VRML

What is VRML

- VRML is a format to describe 3D objects
  - Virtual reality modeling langauge
  - Generating a VRML file is simple than GL programming, since a mapping from 3D model to the 2D screen is not necessary
- With VRML browser such as SGI Cosmo Player, a VRML becomes immersive allowing a user to walk through the 3D scene
- VRML provides 3D worlds with integrated hyperlinks on web
  - Users can afford viewing 3D scenes without having to access expensive graphic workstations
- The VRML has become international standard under name VRML97
- www.web3d.org/vrml/vrml.htm

Rendering Penrose Staircase





CosmoPlayer plug-in for Netscape or Explorer Web browsers is used
Download from http://ca.com/cosmo

www-vrl.umich.edu/intro

VRML Basics

- **Header:** Every VRML file should start with a header

```
#VRML V1.0 ascii
```

- **Pen position:** A state variable specifying the current position of a virtual pen. A geometric node (such as a sphere and cylinder) lets the virtual pen down and draws the corresponding object centered at the current virtual pen position
- **Translation node:** Translate the virtual pen position, keeping the pen up. The translation field consists of three numbers specifying the x, y and z components of the translation vector

```
Translation {translation 5.0 0.0 5.0}
```

- **Sphere node:** Sphere node draws a sphere with a specified radius around the current pen position

```
Sphere {radius 0.2}
```

- **Material node:** Defines material properties

```
Material {diffuseColor 1 0 0}
```

VRML Visualization of MD Data

- Visualizing MD atomic configurations using a ball model
- Save atomic positions from MD to a file, **`conf.d`**
- Write a program to translate the file into a VRML file, **`bmd.wrl`**, which can be visualized by a standard VRML viewer such SGI Cosmo Player
- A VRML file is a description of a 3D model. It contains a sequence of primitive geometrical objects and state-variable specifications

Input Atomic Configuration File

`conf.d`

108	Number of atoms
5.0 0.0 5.0	x, y and z coordinates of 1st atom
5.0 5.0 0.0	
……	

Output VRML File

bmd.wrl

```
#VRML V1.0 ascii

Material {diffuseColor 1 0 0}
Translation {translation 5.0 0.0 5.0}
Sphere {radius 0.2}
Material {diffuseColor 0 1 0}
Translation {translation 5.0 5.0 0.0}
Sphere {radius 0.2}
......
```

Creating VRML File

ball_vrml.c

```
ofstream fvr(“bmd.wrl”);
fvr << “#VRML V1.0 ascii\n”;          // VRML header
xp=yp=zp=0.0;              // Initialize the pen position
for (i=0; i<n; i++){             // Loop over atoms
    setcolor(i % 4);       // Choose a color over atom ID
    fvr << “Translation {“               // Move the pen
        << “translation” << “ ”
        << x[i][0]-xp << “” <<x[i][1]-yp<<“”<<x[i][2]-zp;
    fvr <<“}\n”;
    fvr << “Sphere{radius “<< radius <<“}\n”; // Draw a
                                              // sphere
    xp=x[i][0]; yp=x[i][1]; z[=[i][2];     // Prepare for
}                                        // the next pen shift
```

Creating VRML File (Contd.)

```
void setcolor(int ic){
    float r, g, b;
    Switch (ic) {
       case 0:
              r=1.0, g=0.0; b=0.0; break;
       case 1:
              r=0.0; g=1.0; b=0.0; break;
       case 2:
              r=0.0; g=0.0; b=1.0; break;
       default;
       r=1.0; g=1.0; b=0.0;
    }
    fvr << “material{diffuseColor”
        << r << “ ” << g << “ “ << b << “}\n”;
}
```