Using AVRIL -- A Tutorial

Version 2.0
March 28, 1995

Bernie Roehl

Note: This is the AVRIL tutorial. The detailed technical reference is a separate document.

What is AVRIL?

AVRIL is A Virtual Reality Interface Library, a software package that allows you to create and interact with virtual worlds. It consists of a fast polygon-based rendering engine and a set of support routines that make virtual world creation a reasonably straightforward process.

AVRIL is designed to be fast, portable and easy to use. It's written entirely in ANSI C; the PC version also has a few short assembly-language routines to handle some of the fixed point math. The API (Applications Programming Interface) is simple and well-documented, which should make applications easy to develop.

Most important, AVRIL is free for non-commercial use. The problem with most current VR libraries is that they're very, very expensive; many of them cost more than the computers they run on! AVRIL is intended to give everyone with an interest in VR an opportunity to develop some simple applications without having to invest huge sums of money.

What does "free for non-commercial use" mean?

It means that if you're only writing programs for your own use, or to give away free to others, you can use AVRIL without paying anything.

Who developed AVRIL?

AVRIL was developed by Bernie Roehl, between November of 1993 and April of 1995. It's designed to be somewhat backward-compatible with an earlier rendering engine called REND386 that was developed by Bernie Roehl and Dave Stampe.

So what makes AVRIL different from REND386?

From the beginning, we knew that REND386 would never run on anything but computers in the 386 family; that's why we called it "REND386" in the first place. REND386 was fast, but it achieved its speed at the price of portability; large parts of the code were hand-translated to 386 assembly language. This obviously reduced the portability of the software, as well as making it more difficult to maintain.

AVRIL, by contrast, is written entirely in C. It's fast because the algorithms are well-chosen and carefully written. While it's not as fast overall as REND386, there are actually some situations where it's faster; once it's been optimized a bit, the speed should be comparable. Since it's written in C, AVRIL is also much easier to maintain than REND386 was.

Using AVRIL

AVRIL is very easy to use. Rather than spend a lot of time discussing the details of how it works, let's start by creating a simple AVRIL program:

```
/* EXAMPLE1 -- a cube */

/* Written by Bernie Roehl, April 1994 */

#include "avril.h"

void main()
        {
        vrl_Object *cube;
        vrl_Light *light;
        vrl_Camera *camera;

        vrl_SystemStartup();

        cube = vrl_ObjectCreate(vrl_PrimitiveBox(100, 100, 100, NULL));
        vrl_ObjectRotY(cube, float2angle(45));

        light = vrl_LightCreate();
        vrl_LightRotY(light, float2angle(45));
        vrl_LightRotX(light, float2angle(45));

        camera = vrl_CameraCreate();
        vrl_CameraRotX(camera, float2angle(45));
        vrl_CameraMove(camera, 0, 500, -500);

        vrl_SystemRun();
        }
```

Notice that the only file we had to #include was "avril.h"; that file contains prototypes for all the AVRIL functions, along with a number of useful macros. The avril.h file #includes <stdio.h> (since it references the FILE * type) so there's no need for you to do so yourself. Since some of the macros in avril.h use the memcpy() function, the avril.h file will automatically #include whatever header file is needed to define memcpy(); this is <string.h> on most platforms.

The program shown above simply creates a cube, a light source and a virtual camera. All the AVRIL routines and data types have names beginning with "vrl_"; this ensures that they won't conflict with any routines you write. The vrl_SystemStartup() routine does all the system initialization; the source code for all the vrl_System functions is found in system.c, in case you're curious as to how they work. We'll be looking at them in detail later.

Once the initialization is done, the program creates the cube by calling a routine that generates a primitive box shape; the sides are all 100 units in length. After it's been created, the cube is rotated 45 degrees around the vertical (Y) axis. The float2angle() routine converts a floating-point number into an internal format used for storing angles.

A directional light source is then created, and rotated 45 degrees in each of X and Y. Next, a virtual camera is created, rotated and moved into position. Finally, vrl_SystemRun() is called; vrl_SystemRun() sits in a loop, checking for keyboard or mouse activity and doing the rendering as needed.

To compile and link the program using Borland C++, you would give the following command:

```
bcc -ml example1.c input.c avril.lib
```

This compiles example1.c and input.c and links them with the AVRIL library. The routines in input.c are discussed in a later section.

Sharing Shapes

Our first example was pretty straightforward; let's try something more complex.

```
/* EXAMPLE2 -- several asteroids, sharing the same geometry */

/* Written by Bernie Roehl, April 1994 */

#include "avril.h"
#include <stdlib.h>  /* needed for rand() */

void main()
        {
        FILE *infile;
        vrl_Light *light;
        vrl_Camera *camera;
        vrl_Shape *asteroidshape = NULL;
        int i;

        vrl_SystemStartup();

        vrl_WorldSetHorizon(0);    /* turn off horizon */
        vrl_WorldSetSkyColor(0);   /* black sky */

        infile = fopen("asteroid.plg", "r");
        if (infile)
                {
                asteroidshape = vrl_ReadPLG(infile);
                fclose(infile);
                }

        light = vrl_LightCreate();
        vrl_LightRotY(light, float2angle(45));
        vrl_LightRotX(light, float2angle(45));
        vrl_LightSetIntensity(light, float2factor(0.9));
```

```
camera = vrl_CameraCreate();
vrl_CameraMove(camera, 0, 100, -50);

for (i = 0; i < 5; ++i)
        {
        vrl_Object *obj = vrl_ObjectCreate(asteroidshape);
        vrl_ObjectMove(obj, rand() % 1000, rand() % 1000, rand() % 1000);
        }

vrl_SystemRun();
}
```

When you run this program, look around using the arrow keys to spot the (stationary) asteroids. This program illustrates a useful memory-saving feature of AVRIL. The shape of an object (i.e., its geometric description) is separate from the information about its location and orientation. Any number of objects can share the same geometric description, saving substantial amounts of memory. A geometric description is called a vrl_Shape, and consists of a set of vertices, facets and other information.

The program shown above begins by turning off the horizon (it’s on by default) and setting the sky color to 0 (black). The sky color is used as the screen clear color if there’s no horizon. Next, the file "asteroid.plg" is loaded; AVRIL supports the PLG file format, described in Appendix C. The vrl_ReadPLG() function returns a pointer to a vrl_Shape (the same data type that was returned by the vrl_PrimitiveBox() function in our first example).

A light source and camera are again set up, and five virtual objects are created using the shape that was loaded by vrl_ReadPLG(). Notice that the file only had to be read once, and that the vertices and facets making up an asteroid are only stored once in memory. Each of the asteroids is moved to a random location in an imaginary box 1000 units on a side.

As you move around, you’ll notice that the appearance of an asteroid changes depending on how far away you are from it; if you get close enough, it’s a rough, craggy surface. The "asteroid.plg" file stores multiple representations of the object, and AVRIL automatically selects one of those representations based on distance. This can speed up the rendering process by allowing fewer vertices and facets to be used when an object is far away.

Making Maps

AVRIL not only separates geometry from location/orientation information, it also stores surface descriptions separately. Each object has a "surface map" associated with it, which stores pointers to actual vrl_Surface specifiers. Each surface has a type, a hue and a brightness; in our examples, the surface type is always SURFACE_FLAT (meaning that flat shading is used). The hue is what most people think of as the "color", and the brightness is how much light the surface reflects back to the eye. The higher the brightness value and the more directly that light is striking the surface, the more intense the color.

You can assign surface maps to objects, and change them whenever you like. Our third example program uses two different surface maps, called *map1* and *map2*:

```
/* EXAMPLE3 -- surface maps */

/* Written by Bernie Roehl, April 1994 */

#include "avril.h"
#include <stdlib.h>  /* needed for rand() */

void main()
       {
       FILE *infile;
       vrl_Light *light;
       vrl_Camera *camera;
       vrl_Shape *colorthing = NULL;
       vrl_Surfacemap *map1, *map2;
       int i;

       vrl_SystemStartup();

       map1 = vrl_SurfacemapCreate(6);
       map2 = vrl_SurfacemapCreate(6);
       for (i = 0; i < 6; ++i)
              {
              vrl_SurfacemapSetSurface(map1, i, vrl_SurfaceCreate(i + 1));
              vrl_SurfacemapSetSurface(map2, i, vrl_SurfaceCreate(7 + i));
              }

       infile = fopen("colorful.plg", "r");
       if (infile)
              {
              colorthing = vrl_ReadPLG(infile);
              fclose(infile);
              }

       light = vrl_LightCreate();
       vrl_LightRotY(light, float2angle(45));
       vrl_LightRotX(light, float2angle(45));

       camera = vrl_CameraCreate();
       vrl_CameraMove(camera, 0, 100, -50);

       for (i = 0; i < 10; ++i)
              {
              vrl_Object *obj = vrl_ObjectCreate(colorthing);
              if (i & 1)
                     vrl_ObjectSetSurfacemap(obj, map1);
              else
                     vrl_ObjectSetSurfacemap(obj, map2);
              vrl_ObjectMove(obj, rand() % 1000, rand() % 1000, rand() % 1000);
              }

       vrl_SystemRun();
       }
```

The program creates the two maps using the vrl_SurfacemapCreate() function; the parameter is the number of entries the map should have. Six entries are then created in each map by calling vrl_SurfaceCreate(); the parameter to that function is the hue. The first map will use hues 1 through 6 inclusive, the second will use hues 7 through 12. A shape is then loaded from the file "colorful.plg"; that file uses indexed surface descriptors (0x8000, 0x8001 etc) that refer to entries in the surface map. Refer to Appendix C for more details about surface descriptors.

The light source and camera are again set up, and ten objects are created. Half of them (the odd-numbered ones) are assigned *map1* and the others are assigned *map2*. The objects are again positioned randomly.

Notice how half the cubes are a different color from the other half. Each set of surface descriptions is only stored once, and each surface map is shared by five of the ten cubes. All the cubes share the same vrl_Shape information, which is only stored once.

A Real Taskmaster

AVRIL has a pseudo-tasking facility, which allows you to add routines to a list that gets processed continuously while the system runs. Each task has a function and possibly some data, as well as an indication of how often it should be run.

Our fourth example is more complex that the first three; it creates several primitive shapes, sets up surface maps, and creates tasks to make the objects move by themselves. We'll have spinning cubes, bouncing spheres and pulsating cylinders.

```
/* EXAMPLE4 -- simple object behaviours */

/* Written by Bernie Roehl, April 1994 */

#include "avril.h"
#include <stdlib.h>  /* needed for rand() */

static vrl_Angle spinrate;
static vrl_Time bounce_period;
static vrl_Scalar maxheight;
static vrl_Time pulse_period;

static void spin(void)
        {
        vrl_ObjectRotY(vrl_TaskGetData(), vrl_TaskGetElapsed() * spinrate);
        vrl_SystemRequestRefresh();
        }

static void bounce(void)
        {
        vrl_Object *obj = vrl_TaskGetData();
        unsigned long off;
        vrl_Scalar height;
        off = (360 * (vrl_TaskGetTimeNow() % bounce_period)) / bounce_period;
        height = vrl_FactorMultiply(vrl_Sine(float2angle(off)), maxheight);
        vrl_ObjectMove(obj, vrl_ObjectGetWorldX(obj), height, vrl_ObjectGetWorldZ(obj));
        vrl_SystemRequestRefresh();
        }

static void pulsate(void)
        {
        vrl_Surface *surf = vrl_SurfacemapGetSurface((vrl_Surfacemap *) vrl_TaskGetData(), 0);
        unsigned long off;
        int brightness;
        off = (360 * (vrl_TaskGetTimeNow() % pulse_period)) / pulse_period;
        brightness = abs(vrl_FactorMultiply(vrl_Sine(float2angle(off)), 255));
        vrl_SurfaceSetBrightness(surf, brightness);
        vrl_SystemRequestRefresh();
        }

void main()
        {
        vrl_Light *light;
        vrl_Camera *camera;
        vrl_Shape *cube, *sphere, *cylinder;
        vrl_Surfacemap *cubemap, *pulsemap;
        int i;

        vrl_SystemStartup();

        cube = vrl_PrimitiveBox(100, 100, 100, NULL);
        sphere = vrl_PrimitiveSphere(100, 6, 6, NULL);
        cylinder = vrl_PrimitiveCylinder(100, 50, 100, 8, NULL);
```

```
cubemap = vrl_SurfacemapCreate(1);
vrl_SurfacemapSetSurface(cubemap, 0, vrl_SurfaceCreate(5));
pulsemap = vrl_SurfacemapCreate(1);
vrl_SurfacemapSetSurface(pulsemap, 0, vrl_SurfaceCreate(14));

spinrate = float2angle(72.0 / vrl_TimerGetTickRate());  /* deg per tick */
bounce_period = 4 * vrl_TimerGetTickRate();  /* four-second period */
maxheight = float2scalar(400);    /* maximum height in units */
pulse_period =  2 * vrl_TimerGetTickRate();  /* two-second period */

light = vrl_LightCreate();
vrl_LightRotY(light, float2angle(45));
vrl_LightRotX(light, float2angle(45));

camera = vrl_CameraCreate();
vrl_CameraRotY(camera, float2angle(5));
vrl_CameraMove(camera, 0, 200, -4400);

for (i = 0; i < 10; ++i)
        {
        vrl_Object *obj = vrl_ObjectCreate(NULL);
        vrl_ObjectMove(obj, rand() % 1000, rand() % 1000, rand() % 1000);
        switch (i & 3)
                {
                case 0:
                        vrl_ObjectSetShape(obj, cube);
                        break;
                case 1:
                        vrl_ObjectSetShape(obj, cube);
                        vrl_ObjectSetSurfacemap(obj, cubemap);
                        vrl_TaskCreate(spin, obj, 10);
                        break;
                case 2:
                        vrl_ObjectSetShape(obj, sphere);
                        vrl_TaskCreate(bounce, obj, 10);
                        break;
                case 3:
                        vrl_ObjectSetShape(obj, cylinder);
                        vrl_ObjectSetSurfacemap(obj, pulsemap);
                        break;
                }
        vrl_TaskCreate(pulsate, pulsemap, 10);
        }

vrl_SystemRun();
}
```

Let's start by looking at main(). Three primitive shapes are created -- a box (100 units on a side), a sphere (100 units in radius, with 6 facets around its "latitude" and 6 slices around its "longitude") and a tapered cylinder (base radius 100, top radius 50, height 100 units with 8 sides). Two surface maps are created, each with a single surface; one called *cubemap* using hue 5 and one called *pulsemap* using hue 14.

Some global variables are then set; *spinrate* is the rate that the cubes should spin, in degrees per "tick". A tick is a small unit of time; the timer runs at 1000 ticks per second, so each tick is one millisecond. In case this changes, you should use the routine vrl_TimerGetTickRate() to found out how many ticks per second the timer is running at.

We do the float2angle() conversion here rather than in the spin() task itself; by storing the vrl_Angle value, we avoid having to do the conversion each time through the simulation loop. Also notice that we divide by the rate at which the system timer runs, in ticks per second; the rotation rate is 72 degrees per second, so we divide by ticks per second to get the rotation rate in degrees per tick.

The *bounce_period* is 4 seconds, converted to ticks; this is the time it takes a bouncing ball to go through one complete up-down cycle. The maximum height a ball will rise to is *maxheight*, arbitrarily set to be 400 units. Note the conversion from floating-point to the internal "vrl_Scalar" format. The *pulse_period* is set to two seconds.

Again, a light and camera are set up so we can view the scene, and ten objects are created and randomly positioned. Some of them are simple cubes (using the default color assigned by vrl_PrimitiveBox()). Some of them are spinning cubes, with a single-entry surfacemap.

A task is created to make each cube spin. Each task has a function, some data, and a "period" which indicates how often the task should be run. In this case, the function is spin(), the data is a pointer to the object to be spun, and the period is 10 ticks. The period doesn't affect the speed at which the cube will spin; it only determines how often the spin() function should be called. The smaller the number, the more often the routine will run and the "smoother" the motion will be; of course, running the tasks more often takes CPU cycles away from rendering.

The bouncing balls are handled the same way as the spinning cubes. The cylinders don't have a task associated with them; instead a separate task is set up that will cause the pulsing to happen. The data for that task is not an object pointer, but rather a pointer to a surface map.

The tasks themselves are quite straightforward. The simplest is the spin() task, which is only two lines long:

```
static void spin(void)
        {
        vrl_ObjectRotY(vrl_TaskGetData(), vrl_TaskGetElapsed() * spinrate);
        vrl_SystemRequestRefresh();
        }
```

This task gets a pointer to its data using vrl_TaskGetData(); this is a pointer to the object associated with this task. The task also gets the elapsed time (in ticks) since it last ran, multiplies that value by *spinrate*, and rotates the object by that amount around the vertical (Y) axis. The spin() function then calls vrl_SystemRequestRefresh(), which tells the system that the screen should be refreshed (since an object has moved).

The bounce() task is only slightly more complex; it uses the sine function to determine the height at which the object should be positioned:

```
static void bounce(void)
        {
        vrl_Object *obj = vrl_TaskGetData();
        unsigned long off;
        vrl_Scalar height;
        off = (360 * (vrl_TaskGetTimeNow() % bounce_period)) / bounce_period;
        height = vrl_FactorMultiply(vrl_Sine(float2angle(off)), maxheight);
        vrl_ObjectMove(obj, vrl_ObjectGetWorldX(obj), height, vrl_ObjectGetWorldZ(obj));
        vrl_SystemRequestRefresh();
```

```
	}
```

The current time is obtained from vrl_TaskGetTimeNow(), and the % operator is used to find the modulus (remainder) of the current time relative to the bounce period. That value, divided by the bounce period, is the fraction of the bounce period that has elapsed. We multiply that by 360 (the number of degrees in a circle) to get an offset value; we take the sine of that value (using the fast vrl_Sine() routine) and multiply by the maximum height value. The vrl_FactorMultiply() routine takes a fractional number (of the type returned by vrl_Sine()) and multiplies it by a vrl_Scalar value to get a (smaller) vrl_Scalar value.

We use vrl_ObjectMove() to actually position the object. Notice the use of vrl_ObjectGetWorldX() and vrl_ObjectGetWorldZ() to find the current X and Z values of the object's location; we don't want to alter those values, only the height. A call to the function vrl_SystemRequestRefresh() ensures that the screen will be redrawn with the object at its new height.

The pulsate() task is similar to the bounce() task, but instead of computing a height it computes a brightness and sets it as the new brightness value of the surface. Brightness values are in the range of 0 to 255.

Left to Our Own Devices

AVRIL supports the use of a variety of input devices for manipulating your viewpoint and the objects in your virtual world. Our next example shows one way to use them.

```
/* EXAMPLE5 -- manipulating a cube with the Logitech Cyberman */

/* Written by Bernie Roehl, August 1994 */

#include "avril.h"
#include "avrildrv.h"

vrl_Object *cube = NULL;

static void cube_mover(void)
	{
	vrl_Device *dev = vrl_TaskGetData();
	vrl_Object *viewer = vrl_CameraGetObject(vrl_WorldGetCamera());
	vrl_Vector v;
	vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, YROT), Y, VRL_COORD_OBJREL, viewer);
	vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, XROT), X, VRL_COORD_OBJREL, viewer);
	vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, ZROT), Z, VRL_COORD_OBJREL, viewer);
	vrl_VectorCreate(v, vrl_DeviceGetValue(dev, X), vrl_DeviceGetValue(dev, Y),
		vrl_DeviceGetValue(dev, Z));
	vrl_ObjectTranslate(cube, v, VRL_COORD_OBJREL, viewer);
	vrl_SystemRequestRefresh();
	}

void main()
	{
	vrl_Light *light;
	vrl_Camera *camera;
	vrl_Device *dev;

	vrl_SystemStartup();

	cube = vrl_ObjectCreate(vrl_PrimitiveBox(100, 100, 100, NULL));
	vrl_ObjectRotY(cube, float2angle(45));
```

```
light = vrl_LightCreate();
vrl_LightRotY(light, float2angle(45));
vrl_LightRotX(light, float2angle(45));

camera = vrl_CameraCreate();
vrl_CameraRotX(camera, float2angle(45));
vrl_CameraMove(camera, 0, 500, -500);

dev = vrl_DeviceOpen(vrl_CybermanDevice, vrl_SerialOpen(0x2F8, 3, 2000));
if (dev)
        {
        vrl_DeviceSetScale(dev, X, float2scalar(50));
        vrl_DeviceSetScale(dev, Y, float2scalar(50));
        vrl_DeviceSetScale(dev, Z, float2scalar(50));
        vrl_TaskCreate(cube_mover, dev, 0);
        }

vrl_SystemRun();
}
```

As you can see, there's not much to it. Most of the code is exactly the same as our first example; The only difference is that just before we start running the main loop, we open up a device. The first parameter to the vrl_DeviceOpen() routine is the address of a function that is responsible for operating the device; in this case, it's called vrl_CybermanDevice, and it reads the Logitech Cyberman. Notice that we #included the avrildrv.h file; it has declarations for all the device functions. When you create a new device driver (as described in Appendices F of the technical reference manual) you should put an entry into the avrildrv.h file for it.

The second parameter to vrl_DeviceOpen() is a pointer to a serial port; we could have opened the serial port, assigned it to a variable, and passed that variable to the vrl_DeviceOpen() function, but there was no need to in this case.

The values 0x2F8 and 3 are the hardware address and IRQ number of the COM2 port on a PC-compatible; this example is very platform-specific, but we'll see shortly how to get around that. The value 2000 is the size of the input buffer the serial port should use.

Assuming the device was successfully opened, we scale the X, Y and Z translation values read by the device to be 50 units; that will be the maximum number of world-space units per second that we can move objects using this device. Finally, we create a task whose data parameter is a pointer to our newly-opened device.

The task that does the work of moving the object is called cube_mover(). You'll notice that unlike our first example program, we've declared the cube object as a global variable instead of a local one; this so that cube_mover() can access it.

The cube_mover() task starts by getting the device pointer, and a pointer to the object corresponding to our viewpoint.

```
vrl_Device *dev = vrl_TaskGetData();
vrl_Object *viewer = vrl_CameraGetObject(vrl_WorldGetCamera());
```

Next, cube_mover() rotates the cube. First it does the Y axis, then the X axis, and finally the Z axis. In each case, it rotates the cube relative to the viewer object by an amount that is read from the device.

```
        vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, YROT),
                Y, VRL_COORD_OBJREL, viewer);
        vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, XROT),
                X, VRL_COORD_OBJREL, viewer);
        vrl_ObjectRotate(cube, vrl_DeviceGetValue(dev, ZROT),
                Z, VRL_COORD_OBJREL, viewer);
```

The final step is to read the X, Y and Z translation values from the device, store them in a vector, and translate (move) the object along that vector relative to the viewer.

```
        vrl_VectorCreate(v, vrl_DeviceGetValue(dev, X),
                vrl_DeviceGetValue(dev, Y), vrl_DeviceGetValue(dev, Z));
        vrl_ObjectTranslate(cube, v, VRL_COORD_OBJREL, viewer);
        vrl_SystemRequestRefresh();
```

That's it.

An Independence Movement

The example program above works fine. If you have a Cyberman. And if it's on COM2. And if all you want to do is move a cube. Wouldn't it be nice to have a little more flexibility?

As it turns out, you can. AVRIL supports the use of "configuration files" that store information about a user's preferences and hardware configuration. Our next example uses that configuration information to make our life simpler.

```
/* EXAMPLE6 -- using the configuration file to simplify setup */

/* Written by Bernie Roehl, August 1994 */

#include "avril.h"

static void object_manipulator(void)
        {
        extern vrl_Object *active_object;  /* defined in input.c */
        vrl_Device *dev = vrl_TaskGetData();
        vrl_Object *viewer = vrl_CameraGetObject(vrl_WorldGetCamera());
        vrl_Vector v;
        vrl_ObjectRotate(active_object, vrl_DeviceGetValue(dev, YROT),
                Y, VRL_COORD_OBJREL, viewer);
        vrl_ObjectRotate(active_object, vrl_DeviceGetValue(dev, XROT),
                X, VRL_COORD_OBJREL, viewer);
        vrl_ObjectRotate(active_object, vrl_DeviceGetValue(dev, ZROT),
                Z, VRL_COORD_OBJREL, viewer);
        vrl_VectorCreate(v, vrl_DeviceGetValue(dev, X),
                vrl_DeviceGetValue(dev, Y), vrl_DeviceGetValue(dev, Z));
        vrl_ObjectTranslate(active_object, v, VRL_COORD_OBJREL, viewer);
        vrl_SystemRequestRefresh();
        }

void main(int argc, char *argv[])
        {
        vrl_Device *dev;
        vrl_SystemStartup();
        vrl_ReadCFGfile("example6.cfg");
        vrl_SystemCommandLine(argc, argv);
```

```
        dev = vrl_DeviceFind("manipulator");
        if (dev)
                vrl_TaskCreate(object_manipulator, dev, 0);
        vrl_SystemRun();
        }
```

Our main() is shorter, and simpler. You'll notice that we've added a call to vrl_ReadCFGfile(); it reads the configuration file we specify (in this case it's "example6.cfg"), and configures and initializes all the devices (even opening the serial ports) as specified in the configuration file. The format of the configuration file is described in Appendix B of the technical reference manual.

The vrl_SystemCommandLine() function reads the command line, and loads whatever PLG files, FIG files and WLD files we specify there. The vrl_DeviceFind() function looks for a device that was given the name "manipulator" in the configuration file, and if it finds one it creates a task to move an object using the manipulation device.

The object_manipulator() function is almost the same as cube_mover(), but it uses an external variable called *active_object*. As we'll see later, this variable is found in input.c (where it gets set to the object most recently selected by the mouse).

Using this program, we can explore a virtual world, click on objects to select them, and use the manipulator device we specify in our configuration file to manipulate the selected object. All with just a few lines of code. Note that if you're using the mouse to manipulate objects, you should hit the spacebar to toggle between selecting objects and moving them.

A Smooth Operator

So far all the objects we've been looking at have been flat shaded; that gives them the distinctive "faceted" appearance that you often see in VR systems. However, AVRIL is capable of smooth shading as well, as shown in the following example:

```
/* EXAMPLE7 -- Gouraud shading */

/* Written by Bernie Roehl, April 1995 */

#include "avril.h"

static void load_palette(char *filename)
        {
        FILE *infile = fopen(filename, "rb");
        if (infile)
                {
                vrl_PaletteRead(infile, vrl_WorldGetPalette());
                fclose(infile);
                }
        }

static vrl_Angle tumblerate;

void tumbler(void)
        {
        vrl_Object *obj = vrl_TaskGetData();
        vrl_Angle amount = vrl_TaskGetElapsed() * tumblerate;
        vrl_ObjectRotY(obj, amount);
        vrl_ObjectRotX(obj, amount);
        vrl_SystemRequestRefresh();
```

```
        }

void main()
        {
        vrl_Shape *smooth_shape;
        vrl_Object *thing;
        vrl_Light *light;
        vrl_Camera *camera;
        vrl_Surface *surf;

        vrl_SystemStartup();

        load_palette("shade32.pal");

        smooth_shape = vrl_PrimitiveCylinder(100, 25, 200, 16, NULL);
        vrl_ShapeComputeVertexNormals(smooth_shape);

        surf = vrl_SurfacemapGetSurface(vrl_ShapeGetSurfacemap(smooth_shape), 0);
        vrl_SurfaceSetType(surf, VRL_SURF_GOURAUD);
        vrl_SurfaceSetHue(surf, 4);
        vrl_SurfaceSetBrightness(surf, 243);

        thing = vrl_ObjectCreate(smooth_shape);
        vrl_ObjectRelMove(thing, 0, -100, 0);

        vrl_WorldSetAmbient(0);
        light = vrl_LightCreate();
        vrl_LightRotY(light, float2angle(45));

        camera = vrl_CameraCreate();
        vrl_CameraMove(camera, 0, 0, -1400);

        tumblerate = float2angle(72.0 / vrl_TimerGetTickRate());
        vrl_TaskCreate(tumbler, thing, 0);

        vrl_SystemRun();
        }
```

Almost everything in Example 7 has been used in earlier examples, with three exceptions. The first is the loading of a palette and hue map from a disk file, the second is the setting of the VRL_SURF_GOURAUD shading type on the surface used by the cone, and the third is the call to vrl_ShapeComputeVertexNormals(). In order for smooth (i.e., Gouraud) shading to work, the renderer needs to know the normal vectors at each vertex; the vrl_ShapeComputeVertexNormals() routine computes them by averaging the facet normals.

Technically, you don't need to call vrl_ShapeComputeVertexNormals() on spheres, cones and cylinders created by the vrl_Primitive family of functions; the creation routines do this automatically. Shapes created by the vrl_PrimitiveBox() and vrl_PrimitivePrism() are flat-shaded by default.

The shade32.pal file contains a palette and a hue map, set up to give fewer colors but more shades (in this case, 32 shades of each color instead of the standard 16). This makes the Gouraud shading look a lot better.

You may notice a number of glitches in the shading, especially little white flecks; that's because the Gouraud shading routine was the very last thing I added to this release, and it isn't fully debugged yet! I should have that fixed up in version 2.1, but I didn't want to leave out Gouraud shading altogether for this release. I also didn't want to keep people waiting any longer for this release than I already have.

A Tiny Program

AVRIL provides a number of useful utility routines that reduce the amount of actual programming you have to do in order to create a virtual world. A minimal AVRIL program looks like this:

```
/* A very simple demo of AVRIL */

/* Written by Bernie Roehl, April 1994 */

#include "avril.h"

void main(int argc, char *argv[])
        {
        vrl_SystemStartup();
        vrl_ReadCFGfile(NULL);
        vrl_SystemCommandLine(argc, argv);
        vrl_SystemRun();
        }
```

The NULL parameter to vrl_ReadCFGfile() causes it to use its built-in default of "avril.cfg". This example shows just how little it takes to create a VR program using AVRIL.

Of Mice and Menus

By now, you've probably noticed that something is missing; how have our programs been able to respond to our keystrokes and mouse presses? Well, AVRIL does some of this for you automatically. When you call vrl_SystemRun(), you're essentially turning control of the application over to the system. From time to time, the system will make calls back to your application to give you control if you need it. (If you don't like this approach, you're not stuck with it; the source for the vrl_System functions is provided, so you can do things however you like).

There are currently five places that the system calls your application. Just before starting its main internal loop for the first time, it calls vrl_ApplicationInit(). Just after it clears the screen (or draws the horizon, as the case may be) but before it does the actual rendering of the scene, it calls vrl_ApplicationDrawUnder(). You can use that routine to "underlay" information on the screen that appears behind any objects that are drawn. If you want to use your own background, just turn off screen clearing using vrl_WorldSetScreenClear(0) and do your background drawing in vrl_ApplicationDrawUnder().

After the system has rendered the entire scene, it calls vrl_ApplicationDrawOver(); this allows you to "overlay" information on the screen. The vrl_ApplicationDrawOver() routine is where you would put any "heads-up display" type information, such as frame rate or orientation information.

Whenever a keystroke is detected, it's passed to the vrl_ApplicationKey() routine. Similarly, mouse-up events are passed to the application using vrl_ApplicationMouseUp().

All of these routines have default versions in the AVRIL library, so you don't have to write all of them. The default versions of the functions vrl_ApplicationDrawUnder(), vrl_ApplicationDrawOver() and vrl_ApplicationMouseUp() are empty (i.e., they don't do anything). The default version of vrl_ApplicationKey() just checks to see if the user has pressed the ESC key; if they have, vrl_SystemStopRunning() is called.

In addition to all this, there's a simple menu system built into this version of AVRIL; it will be described later.

A Moving Experience

Objects can have functions and data associated with them. When the system walks through the hierarchy of objects, it calls each object's function; those functions can make use of the data associated with the object.

The default vrl_ApplicationInit() routine sets up an object function to let an input device (the keypad by default) move the user around. You can look at the code in input.c for all the details, but essentially here's what it does:

```
vrl_Object *head = vrl_CameraGetObject(vrl_WorldGetCamera());
vrl_Device *headdev = vrl_DeviceFind("head");
if (headdev == NULL)
        headdev = vrl_DeviceOpen(vrl_KeypadDevice, 0);
vrl_ObjectSetApplicationData(head, headdev);
vrl_ObjectSetFunction(head, head_mover);
```

If no head device was specified in the configuration file, the keypad is used. The head is found (the head being the object to which the camera is attached), and the head object's application-specific data field is set to point to the headdev.

The functions that are set on objects get called whenever the world is updated by the vrl_ObjectUpdate() or vrl_WorldUpdate() routines. When object_move_locally() gets called, it just calls object_mover() on the object, passing the device pointer which is stored in the object's application data.

The object_mover() routine is basically the same as the movement tasks that were described earlier (the ones in example 6) but slightly more general.

Lots of Input

The file input.c contains simple versions of vrl_ApplicationDrawOver(), vrl_ApplicationMouseUp(), vrl_ApplicationKey() and vrl_ApplicationInit() that are shared by all our example programs. The vrl_ApplicationMouseUp() routine looks like this:

```
vrl_Object *active_object = NULL;  /* points to the currently-selected object, if any */

void vrl_ApplicationMouseUp(int x, int y, unsigned int buttons)
        {
        vrl_Object *old_active = active_object;
```

```
        if ((buttons & 1) == 0)
                return;
        vrl_RenderMonitorInit(x, y);
        vrl_SystemRender(NULL);  /* redraw screen */
        if (vrl_RenderMonitorRead(&active_object, NULL, NULL))
                {
                if (active_object == old_active)
                        active_object = NULL;
                else
                        vrl_ObjectSetHighlight(active_object, 1);
                }
        if (old_active)
                vrl_ObjectSetHighlight(old_active, 0);
        vrl_SystemRequestRefresh();
        }
```

This routine uses the "Monitor" facility of AVRIL to allow the user to select objects. The mouse location is passed to vrl_RenderMonitorInit(); this tells the system to keep an eye on that point on the screen. The screen is then re-drawn using vrl_SystemRender(), and the monitor is read using vrl_RenderMonitorRead(). If that function returns a non-zero value, then the mouse cursor was on top of an object; since we passed *&active_object* to the vrl_RenderMonitorRead() function, *active_object* now points to the object that the mouse cursor was on top of. This is the object that got moved around by the manipulation device in example 6. If the user clicks again on the previously-selected object, then the *active_object* is set to NULL; otherwise, the newly-activated object gets its highlighting turned on. In any case, we un-highlight the previously active object, and tell the system the screen needs to be refreshed (since the highlighting of an object has changed).

The vrl_ApplicationKey() routine is very simple; the only complicated part is that it handles auto-repeat of keystrokes:

```
void vrl_ApplicationKey(unsigned int c)
        {
        static int lastkey = 0;
        if (c == INS)
                {
                int i;
                for (i = 0; i < 100; ++i)
                        {
                        process_key(lastkey);
                        vrl_SystemRender(vrl_WorldUpdate());
                        }
                }
        else
                process_key(lastkey = c);
        }
```

If the key is INS (defined in avrilkey.h), the last key is re-processed 100 times; all other keys are processed once, and the *lastkey* variable is updated. Notice the call to vrl_SystemRender(); it looks pretty complicated, but after you read some of the later sections it will make more sense. We need to update the world and re-render the scene after every keystroke, so the user will see the ongoing changes.

The process_key() function is fairly long, and will probably change from version to version of AVRIL. Most of it should be pretty easy to understand, so you may want to take a few minutes to look through the source code in input.c (where you'll also find the source for the vrl_ApplicationMouseUp() and vrl_ApplicationDrawOver() routines).

The vrl_ApplicationDrawOver() routine provides the position, frame rate, compass and "heads-up display" support for the AVRIL demos. It looks like this:

```
void vrl_ApplicationDrawOver(vrl_RenderStatus *stat)
	{
	vrl_Camera *cam = vrl_WorldGetCamera();
	char buff[100];
	if (vrl_ConfigGetPositionDisplay())
		{
		sprintf(buff, "Position: %ld,%ld", vrl_CameraGetWorldX(cam),
			vrl_CameraGetWorldZ(cam));
		vrl_UserInterfaceDropText(10, 10, 15, buff);
		}
	if (vrl_ConfigGetFramerateDisplay())
		{
		sprintf(buff, "Frames/sec: %ld", vrl_SystemGetFrameRate());
		vrl_UserInterfaceDropText(5, 170, 15, buff);
		}
	if (vrl_ConfigGetCompassDisplay())
		vrl_UserInterfaceDrawCompass(cam, 250, 40, 35);
	if (showhud)
		{
		sprintf(buff, "%c%c%c",
			stat->memory ?  'M' : ' ',
			stat->objects ? 'O' : ' ',
			stat->facets ?  'F' : ' ');
		vrl_UserInterfaceDropText(10, 20, 15, buff);
		}
	if (vrl_MouseGetUsage())
		{
		vrl_Device *dev = vrl_MouseGetPointer();
		if (dev)
			{
			int x = vrl_DeviceGetCenter(dev, X);
			int y = vrl_DeviceGetCenter(dev, Y);
			int deadx = vrl_DeviceGetDeadzone(dev, X);
			int deady = vrl_DeviceGetDeadzone(dev, Y);
			/* white inner box */
			vrl_DisplayLine(x - deadx, y - deady, x + deadx, y - deady, 15);
			vrl_DisplayLine(x - deadx, y + deady, x + deadx, y + deady, 15);
			vrl_DisplayLine(x - deadx, y - deady, x - deadx, y + deady, 15);
			vrl_DisplayLine(x + deadx, y - deady, x + deadx, y + deady, 15);
			/* black outer box */
			vrl_DisplayLine(x-deadx-1, y-deady-1, x+deadx+1, y-deady-1, 0);
			vrl_DisplayLine(x-deadx-1, y+deady+1, x+deadx+1, y+deady+1, 0);
			vrl_DisplayLine(x-deadx-1, y-deady-1, x-deadx-1, y+deady+1, 0);
			vrl_DisplayLine(x+deadx+1, y-deady-1, x+deadx+1, y+deady+1, 0);
			}
		}
	}
```

There are several "configuration" settings that get accessed to determine what information should be overlaid on the display; the state of those configuration variables is toggled by code in process_key(). This configuration information will be explained in more detail later, in the section about configuration files.

The call to vrl_WorldGetCamera() returns a pointer to the currently-active virtual camera. The buffer *buff[]* will be used to construct strings that we want to display on the screen.

If the user wants their location displayed, a text string containing the camera's current X and Z values is constructed and displayed at location (10, 10) on the screen. The first value is the horizontal distance in pixels from the left of the screen, and the second value is the vertical distance in pixels from the top of the screen. The color used is 15, which is white. The vrl_UserInterfaceDropText() function automatically produces a "drop shadow" behind the text, ensuring it's visible even if it's overlaid on top of a white background.

If the user wants a compass to be shown, the vrl_UserInterfaceDrawCompass() routine is called. The compass is displayed at location (250, 40) on the screen, and each "arm" of the compass is 35 pixels long.

If the *showhud* variable is set, a variety of debugging information is displayed. When the renderer draws a scene, it may run out of internal memory, or it may find there are too many objects or facets for it to process. If this happens, it sets bits in a special structure; a pointer to this structure is passed to vrl_ApplicationDrawOver(), so that it can alert the user to the problem. In this case, an 'M' is displayed if the renderer ran out of memory, an 'O' is displayed if there were too many objects, and an 'F' is displayed if there were too many facets.

If the mouse is in 6D input device mode, a small square is drawn on the screen; if the mouse cursor is inside this box, there'll be no movement. It's sort of a visual "dead zone", if you will. The idea for this box came from a demo of the Superscape VR system; it was a clever enough idea that I adopted it for this example.

Into the System

We've talked a lot so far about the vrl_System routines; now let's take a closer look at how they work.

```
vrl_Boolean vrl_SystemStartup(void)
        {
        vrl_MathInit();
        vrl_WorldInit(vrl_WorldGetCurrent());
        if (vrl_VideoSetup(0))
                {
                printf("Could not enter graphics mode!\n");
                return -1;
                }
        atexit(vrl_VideoShutdown);
        if (vrl_DisplayInit(NULL))
                return -1;
        atexit(vrl_DisplayQuit);
        vrl_MouseInit();
        atexit(vrl_MouseQuit);
        if (vrl_TimerInit())
                return -2;
        atexit(vrl_TimerQuit);
        if (vrl_RenderInit(800, 800, 500, 5, 65000))
                return -3;
        atexit(vrl_RenderQuit);
        atexit(vrl_DeviceCloseAll);
        atexit(vrl_SerialCloseAll);
        /* make sure that exit() [and therefore the atexit() functions] get
           called if there are any fatal errors */
        signal(SIGABRT, exit);
        signal(SIGFPE, exit);
        signal(SIGILL, exit);
        signal(SIGINT, exit);
        signal(SIGSEGV, exit);
        signal(SIGTERM, exit);
        vrl_SystemStartRunning();
        vrl_SystemRequestRefresh();
```

```
        vrl_SystemRender(NULL);
        return 0;
        }
```

The vrl_SystemStartup() routine does the initialization of all the various AVRIL subsystems. It starts by calling vrl_MathInit(), which sets up the trig tables used internally by AVRIL (for example, a table of sines that's used by the vrl_Sine() function described earlier).

Next, the world is initialized and the video subsystem is started up; from this point on, the system is running in graphics mode. The display subsystem is then initialized, followed by the mouse and the timer.

After that, the rendering engine itself is initialized; the parameters to the vrl_RenderInit() function may change with a future release of the software, but for now just use the values that are shown above. The value 65000 is the amount of memory the renderer should allocate for its internal use; if the renderer needs more than this amount of memory when rendering a scene, it will set the "memory" value in the status struct described earlier (which is passed to vrl_ApplicationDrawOver()). If the renderer is unable to initialize itself (for example, if it couldn't allocate the specified amount of memory) then vrl_RenderInit() returns a non-zero value.

Notice the use of atexit() to ensure that everything is shut down properly when the program exits. The signal() calls ensure that the exit() routine will be called in case of any errors; exit() will in turn call the various atexit() functions, cleanly closing down the system.

Finally, vrl_SystemStartRunning() is called and an initial display refresh is requested. The vrl_SystemStartRunning(), vrl_SystemStopRunning(), and vrl_SystemIsRunning() routines are used to control whether the system is currently "running" or not. They just set and check the value of the variable *system_is_running*; however, using the routines makes your code a bit more readable. It's also possible to redefine those routines to do something in addition to just setting or clearing a flag.

The vrl_SystemRun() routine is the main loop of every AVRIL application. It looks like this:

```
void vrl_SystemRun(void)
        {
        vrl_ApplicationInit();
        if (vrl_WorldGetStereoConfiguration())
                vrl_StereoConfigure(vrl_WorldGetStereoConfiguration());
        while (vrl_SystemIsRunning())
                {
                vrl_Object *list;
                if (vrl_KeyboardCheck())
                        vrl_ApplicationKey(vrl_KeyboardRead());
                check_mouse();
                vrl_TaskRun();
                vrl_DevicePollAll();
                list = vrl_WorldUpdate();
                if (vrl_SystemQueryRefresh())
                        vrl_SystemRender(list);
                }
        }
```

It shouldn't come as any surprise that this looks like an event loop in a GUI application; on some systems, that's exactly how vrl_SystemRun() will be implemented. However, on a DOS platform it's necessary to explicitly check the mouse and keyboard for activity.

If a key has been pressed, the keyboard is read and the value of the key is passed to vrl_ApplicationKey(). The function check_mouse() is used to interrogate the mouse for updates:

```
static void check_mouse(void)
        {
        unsigned int mouse_buttons;
        if (vrl_MouseGetUsage())  /* being used as 6D pointing device */
                return;
        if (!vrl_MouseRead(NULL, NULL, NULL))  /* mouse hasn't changed */
                return;
        vrl_MouseRead(NULL, NULL, &mouse_buttons);
        if (mouse_buttons)  /* button down */
                {
                int mouse_x, mouse_y;
                vrl_ScreenPos win_x, win_y;
                unsigned int down_buttons = mouse_buttons;
                vrl_DisplayGetWindow(&win_x, &win_y, NULL, NULL);
                while (mouse_buttons)  /* wait for button release */
                        vrl_MouseRead(&mouse_x, &mouse_y, &mouse_buttons);
                if (down_buttons & 0x07)
                        vrl_ApplicationMouseUp(mouse_x - win_x, mouse_y - win_y, down_buttons);
                }
        }
```

The vrl_MouseGetUsage() call is necessary because the mouse can be used in either of two completely different ways: as a pointing device for selecting objects on the screen, or as a 6 Degree-Of-Freedom (6D) input device; the 6D mode is described later, in the section on input devices. If vrl_MouseGetUsage() returns a non-zero value, then the mouse is being used as a 6D input device, and input from it shouldn't be processed any further at this point.

If the mouse hasn't changed location or button status, the call to vrl_MouseRead() will return zero, in which case no further processing is done. If the mouse buttons are down, the routine tracks the mouse input until the buttons are released. The button status is saved in the variable *down_buttons*, and then passed to the routine vrl_ApplicationMouseUp() along with the mouse cursor location in the current window.

Back in vrl_SystemRun(), the vrl_TaskRun() function is called to run all the tasks that have been created (like the spin(), bounce() and pulsate() tasks we used in example 4). The input devices are polled, and vrl_WorldUpdate() is called; it walks the hierarchical tree of objects in the world, updating their location and orientation information and threading them onto a linked list which is returned as the value of the vrl_WorldUpdate() function. Walking the tree also causes the function associated with each object to be called.

If the display needs to be redrawn (i.e. the vrl_SystemRequestRefresh() routine that we mentioned earlier has been called at least once since we last re-drew the screen) then the vrl_SystemRender() routine is called, and is given the linked list of objects to render.

The vrl_SystemRender() routine does the actual updating of the screen. Even though source is provided, you should use this routine as-is; it's likely to change in future releases of AVRIL, and several additional features will be added. The code currently looks like this:

```
vrl_RenderStatus *vrl_SystemRender(vrl_Object *list)
        {
        static vrl_Object *lastlist = NULL;
        vrl_Palette *pal;
        vrl_StereoConfiguration *conf;
        vrl_RenderStatus *stat;
        int pagenum;
        int two_eyes = 0;
        vrl_Time render_start = vrl_TimerRead();
        if (list == NULL)
                list = lastlist;
        else
                lastlist = list;
        pal = vrl_WorldGetPalette();
        if (vrl_PaletteHasChanged(pal))
                {
                vrl_VideoSetPalette(0, 255, pal);
                vrl_PaletteSetChanged(pal, 0);
                }
        pagenum = vrl_VideoGetDrawPage();
        if (++pagenum >= vrl_VideoGetNpages())
                pagenum = 0;
        vrl_VideoSetDrawPage(pagenum);
        vrl_RenderSetAmbient(vrl_WorldGetAmbient());
        vrl_DisplayStereoSetDrawEye(VRL_STEREOEYE_BOTH);
        if (vrl_WorldGetScreenClear())
                {
                vrl_DisplayBeginFrame();
                if (vrl_WorldGetHorizon() && !vrl_RenderGetDrawMode())
                        vrl_RenderHorizon();
                else
                        vrl_DisplayClear(vrl_WorldGetSkyColor());
                vrl_DisplayEndFrame();
                }
        vrl_ApplicationDrawUnder();
        conf = vrl_WorldGetStereoConfiguration();
        if (conf)
                two_eyes = vrl_StereoGetNeyes(conf);
        if (vrl_WorldGetStereo() && vrl_WorldGetLeftCamera()
                && vrl_WorldGetRightCamera() && two_eyes)
                {
                /* draw left-eye image */
                vrl_DisplayStereoSetDrawEye(VRL_STEREOEYE_LEFT);
                vrl_RenderSetHorizontalShift(vrl_StereoGetTotalLeftShift(conf));
                vrl_DisplayBeginFrame();
                vrl_RenderBegin(vrl_WorldGetLeftCamera(), vrl_WorldGetLights());
                stat = vrl_RenderObjlist(list);
                vrl_DisplayEndFrame();

                /* draw right-eye image */
                vrl_DisplayStereoSetDrawEye(VRL_STEREOEYE_RIGHT);
                vrl_RenderSetHorizontalShift(vrl_StereoGetTotalRightShift(conf));
                vrl_RenderBegin(vrl_WorldGetRightCamera(), vrl_WorldGetLights());
                vrl_DisplayBeginFrame();
                stat = vrl_RenderObjlist(list);
                vrl_DisplayEndFrame();
                }
        else  /* not two-eye stereo */
                {
                vrl_DisplayStereoSetDrawEye(VRL_STEREOEYE_BOTH);
                vrl_RenderSetHorizontalShift(0);
                vrl_DisplayBeginFrame();
                vrl_RenderBegin(vrl_WorldGetCamera(), vrl_WorldGetLights());
                stat = vrl_RenderObjlist(list);
                vrl_DisplayEndFrame();
                }
        vrl_DisplayStereoSetDrawEye(VRL_STEREOEYE_BOTH);
        vrl_RenderSetHorizontalShift(0);
        vrl_ApplicationDrawOver(stat);
        vrl_VideoCursorHide();
```

```
        vrl_DisplayUpdate();
        vrl_VideoSetViewPage(pagenum);
        vrl_VideoCursorShow();
        last_render_ticks = vrl_TimerRead() - render_start;
        need_to_redraw = 0;
        return stat;
        }
```

First, the current time is stored in the variable *render_start*; this is later used to compute the frame rate.

If the list that the vrl_SystemRender() routine is given is NULL (as was the case in our example vrl_ApplicationMouseUp() routine in input.c) then the last list of objects rendered is used. If the palette has changed since the last frame, it gets copied to the hardware palette and the "changed" flag is cleared.

The system uses the concept of a "draw" page (on which drawing takes place) and a "view" page (which is the one the user is currently viewing). The vrl_SystemRender() routine gets the current drawing page number, and increments it (so we start drawing on the next page). It wraps back to page zero after it's drawn the last available display page.

The ambient light level is set according to that of the current world. If it's necessary to clear the screen, the system does so (or draws a horizon, as appropriate). Then the vrl_ApplicationDrawUnder() routine we looked at earlier is called.

A check is made to see if we're configured for stereoscopic viewing. If we are, and if both the left and right eye cameras exist, and if it's a "two-eye" system (i.e., not Chromadepth or SIRDS) then the left eye image is drawn followed by the right eye image. If we're not doing two-eye stereoscopic rendering, a single image is drawn.

To draw an image, we start by selecting an eye and setting a corresponding horizontal offset. Next, we tell the display subsystem to get ready for a new frame, tell the rendering engine about our camera and lights (using vrl_RenderBegin()), and call vrl_RenderObjlist() to actually draw the objects. Finally, we tell the display subsystem that the frame is complete.

The vrl_ApplicationDrawOver() routine is then called to put any additional information on the display. The cursor is hidden, and vrl_DisplayUpdate() is called; this is necessary, since some display systems don't have multiple pages and instead use an off-screen buffer which the vrl_DisplayUpdate() routine copies to the screen. For systems that have multiple-page displays, the current view page is set to the (now finished) drawing page. The mouse cursor is then revealed again, the time it took to do all this is noted, and the *need_to_redraw* variable (which was set by vrl_SystemRequestRefresh()) is cleared.

The final vrl_System routine is vrl_SystemCommandLine(). It just goes through the command-line parameters and calls the appropriate routines to read the various types of files:

```
void vrl_SystemCommandLine(int argc, char *argv[])
        {
        int i;
        vrl_Camera *cam;
```

```
for (i = 1; i < argc; ++i)  /* i = 1 to skip argv[0] */
        {
        FILE *in = fopen(argv[i], "r");
        if (in == NULL) continue;
        if (strstr(argv[i], ".wld"))
                vrl_ReadWLD(in);
        else if (strstr(argv[i], ".fig"))
                vrl_ReadFIG(in, NULL, NULL);
        else if (strstr(argv[i], ".plg"))
                vrl_ReadObjectPLG(in);
        /* ignore anything else */
        fclose(in);
        }
if (!vrl_WorldGetCamera())   /* need to have a camera */
        vrl_CameraCreate();
vrl_WorldUpdate();
}
```

After all the files on the command line have been processed, the vrl_SystemCommandLine() routine checks to see if a current camera has been set. If not, a new camera is created.

Configuration Files

AVRIL supports the loading of configuration files; the format of these files is given in Appendix B of the technical reference manual. The functions that support loading configuration information are

```
int vrl_ReadCFG(FILE *in);
int vrl_ReadCFGfile(char *filename);
```

The vrl_ReadCFG() function reads a configuration file and stores the information from it in a set of internal data structures. Any devices specified in the file are opened, and the serial ports they use are opened as well. If a display driver is specified, the display is initialized using that driver. The vrl_ReadCFGfile() routine does the same thing, but uses the name of a file rather than a pointer to an already-opened file. The filename is processed to prepend the current loadpath (unless the filename begins with a slash). If the filename is NULL, then "avril.cfg" is used.

There are routines for reading, setting and toggling the various flags that the user can specify in the configuration file:

```
void vrl_ConfigSetCompassDisplay(vrl_Boolean flag);
vrl_Boolean vrl_ConfigGetCompassDisplay(void);
void vrl_ConfigToggleCompassDisplay(void);
void vrl_ConfigSetPositionDisplay(vrl_Boolean flag);
vrl_Boolean vrl_ConfigGetPositionDisplay(void);
void vrl_ConfigTogglePositionDisplay(void);
void vrl_ConfigSetFramerateDisplay(vrl_Boolean flag);
vrl_Boolean vrl_ConfigGetFramerateDisplay(void);
void vrl_ConfigToggleFramerateDisplay(void);
```

If you wanted to let the user specify a configuration file to load by setting the AVRIL environment variable, you would make the following call:

```
vrl_ReadCFGfile(getenv("AVRIL"));
```

If no AVRIL environment variable is found, getenv() will return NULL and avril.cfg will be used.

That's All, Folks!

For more detailed information about AVRIL, check out the AVRIL reference manual. It contains an in-depth description of all the AVRIL functions and data types.

If you have problems using AVRIL, drop me a line via email. My address is broehl@sunee.uwaterloo.ca; be sure to put AVRIL in the subject line so I know what it's about, otherwise it might take me days to get back to you. (It might anyway...)

Support for AVRIL

There are two electronic mailing lists for discussing AVRIL. The first list is called *avril-announce*, and is used for announcements of new releases, utilities, applications and so on. The second list is called *avril-developers*, and is used as a way for people who are developing applications using AVRIL to communicate and exchange ideas.

To subscribe to either or both lists, send mail to majordomo@sunee.uwaterloo.ca with the following line(s) in the body of the message:

```
subscribe avril-announce YourName
subscribe avril-developers YourName
```

To unsubscribe from either or both lists, do the exact same thing but with the word "unsubscribe" instead of the word "subscribe".

Future Features

Features that will be added in future releases include routines for sound, networking, and an application language of some sort. I also hope to add Z-buffering and texture mapping.

The latest release of AVRIL can always be found on sunee.uwaterloo.ca, in the pub/avril directory. I will also try to put the latest version on major sites such as ftp.wustl.edu, oak.oakland.edu, x2ftp.oulu.fi and possibly others. Please feel free to make it available to everyone; the only restrictions are that you can't sell it (since it's free!) and you can't develop commercial applications without properly licensing it.

There should be a new release of AVRIL every few months; starting with version 2.5, AVRIL should be ported to several other platforms.

One sad note: after many years of cheerfully using Borland C, I've decided to move on. All the new features I want to add (Z-buffering, texture mapping, higher resolutions, support for 16- and 24-bit color) all require lots of memory. The old 640k barrier is just too much of a limitation, so I'm going to protected mode. The best protected mode compiler is Watcom C, and I've already started the conversion.

What does this mean for users of AVRIL? Well, it might mean buying a new compiler. However, I may try to port AVRIL to DJGPP, a freeware C compiler based on GNU C. No promises, but if I have time I'll give it a shot.

In the meantime, I hope you enjoy using AVRIL.