AVRIL Technical Reference

Version 2.0
March 28, 1995

Bernie Roehl

Note: This is the technical reference manual for version 2.0 of AVRIL. The tutorial is a separate document. The appendices for this technical reference manual are also stored separately.

This document describes AVRIL from a technical perspective, and explains the data types and functions that are available. Anything that is not documented here should not be used, since it's subject to change in future releases of AVRIL. Also keep in mind that some of the routines described below may be implemented as macros, and that this may also change in future releases; none of your code should assume that any particular routine is either a macro or a real function.

There are a number of important concepts that are essential to an understanding of how AVRIL works. Let's start by examining them.

Basic Data Types

AVRIL uses a left-handed coordinate system; if the X axis points to the right, and the Y axis points up, then Z points straight ahead. If you're looking at the origin from the positive end of an axis, a clockwise rotation is a positive rotation angle.

Distances in AVRIL are represented by vrl_Scalars, and rotation angles by vrl_Angles. AVRIL can be compiled to use either floating point or fixed point, so it's important to use the vrl_Scalar and vrl_Angle types for portability. vrl_Scalars should always be treated as (long) integer values, regardless of whether floating or fixed point is used. vrl_Angles are always measured in degrees (converted to the internal vrl_Angle format, of course). A third fundamental data type is vrl_Factor, which is used for things like the return value of trig functions; a special constant called VRL_UNITY is #defined to be a vrl_Factor of 1.

In a floating point implementation, all three types (vrl_Scalars, vrl_Angles and vrl_Factors) are stored as floats; in a fixed-point implementation, they're all 32-bit integers (which will be "long" on many systems). The following macros are provided for converting between floating point (or regular integers) and the three special types:

```
float2scalar(float);
scalar2float(vrl_Scalar);

float2angle(float);
angle2float(vrl_Angle);

float2factor(float);
factor2float(vrl_Factor);
```

Several routines are provided to support portable multiplication and division of the types:

```
vrl_Factor vrl_ScalarDivide(vrl_Scalar a, vrl_Scalar b);
vrl_Scalar vrl_ScalarMultDiv(vrl_Scalar a, vrl_Scalar b, vrl_Scalar c);
vrl_Scalar vrl_FactorMultiply(vrl_Factor a, vrl_Scalar b);
```

The first of these routines simply divides two vrl_Scalars and returns a vrl_Factor result; the absolute value of *a* should be less than or equal to the absolute value of *b*. The second routine multiplies two vrl_Scalars and divides by a third, using a 64-bit intermediate result; in other words, it computes *(a*b)/c*. The third routine multiplies a vrl_Factor by a vrl_Scalar and returns a vrl_Scalar result; it can also be used for multiplying two vrl_Factors, or an integer or long value by a vrl_Factor. The order of the operands is significant, because C automatically promotes ints to longs.

In floating-point implementations of AVRIL, there may be occasions where the computed value of a vrl_Scalar has a fractional part; in such cases you should use the following function:

```
vrl_Scalar vrl_ScalarRound(vrl_Scalar value);
```

to round to the nearest valid vrl_Scalar value. To take the absolute value of a vrl_Scalar, use the function

```
vrl_Scalar vrl_ScalarAbs(vrl_scale value);
```

There are currently two trig routines, vrl_Sine() and vrl_Cosine(); they both take vrl_Angles as parameters and return vrl_Factors:

```
vrl_Factor vrl_Sine(vrl_Angle angle);
vrl_Factor vrl_Cosine(vrl_Angle angle);
```

The routine vrl_MathInit() should be called before calling any of the trig functions; it pre-computes the trig tables. This is done in the vrl_SystemStartup() routine (found in system.c).

Positions on the screen (i.e., pixel coordinates) are represented using the vrl_ScreenPos type, for future portability. Fractional screen positions (used in scan-converting polygons) are represented using the vrl_ScreenCoord type. This type is currently used only by the vrl_Display family of routines.

There are several other basic types used in AVRIL: vrl_Time is a measure of elapsed time in ticks, vrl_Color is used to represent colors (both 8-bit and 24-bit) and vrl_Boolean is a true/false type value (non-zero being true). The types vrl_32bit and vrl_unsigned32bit are used for signed and unsigned 32-bit numbers, and vrl_16bit and vrl_unsigned16bit are used for signed and unsigned 16-bit numbers. The function

```
vrl_32bit abs32(vrl_32bit value);
```

will return the absolute value of a 32-bit number, independent of whether 32-bit values are of type int or type long.

Vectors

A vrl_Vector is a three-element array, which can be indexed by the #defined constants X, Y and Z; for example, if *v* is a vector then *v*[X] is the X-component of the vector. In general, vrl_Vectors are made up of three vrl_Scalars; however, a normalized vector (such as a facet normal, a basis vector, or a vector that's been normalized using the vrl_VectorNormalize() function) will actually have vrl_Factors as elements. The following functions perform fundamental operations on vrl_Vectors:

```
void vrl_VectorCreate(vrl_Vector result, vrl_Scalar x, vrl_Scalar y, vrl_Scalar z);
void vrl_VectorCopy(vrl_Vector destination, vrl_Vector source);
void vrl_VectorAdd(vrl_Vector result, vrl_Vector v1, vrl_Vector v2);
void vrl_VectorSub(vrl_Vector result, vrl_Vector v1, vrl_Vector v2);
void vrl_VectorNegate(vrl_Vector v);
vrl_Factor vrl_VectorDotproduct(vrl_Vector v1, vrl_Vector v2);
vrl_Scalar vrl_VectorCrossproduct(vrl_Vector result, vrl_Vector v1, vrl_Vector v2);
vrl_Scalar vrl_VectorMagnitude(vrl_Vector v);
void vrl_VectorNormalize(vrl_Vector v);
vrl_Scalar vrl_VectorDistance(vrl_Vector v1, vrl_Vector v2);
void vrl_VectorScale(vrl_Vector v, vrl_Scalar newmag);
void vrl_VectorRescale(vrl_Vector v, vrl_Scalar newmag);
void vrl_VectorPrint(FILE *out, char *str, vrl_Vector v);
vrl_Boolean vrl_VectorEqual(vrl_Vector v1, vrl_Vector v2);
void vrl_VectorZero(vrl_Vector v);
```

The vrl_VectorCreate() function takes three vrl_Scalars and assembles them into a vrl_Vector. The vrl_VectorCopy(), vrl_VectorAdd() and vrl_VectorSub() routines do element-by-element copies, additions and subtractions of vrl_Vectors. The vrl_VectorNegate() function reverses the direction of a vrl_Vector by flipping the sign of each of its components. The vrl_VectorDotproduct() routine computes the dot product (inner product) of two vectors; at least one of the vectors should be normalized for this to work properly.

The vrl_VectorCrossproduct() routine computes the vector cross product (outer product) of two vectors. This is likely to be slow, since it normalizes the result (which involves doing a square root operation). It returns the magnitude of the vector prior to normalization. The vrl_Magnitude() routine returns the magnitude of a vector, and the vrl_VectorNormalize() routine scales a vector so that it has a magnitude of 1.

The vrl_VectorDistance() routine takes two vrl_Vectors (each representing a point in space) and computes the distance between those two points. The vrl_Scale() function takes a normalized vrl_Vector and scales all its components by the given amount; the vrl_Rescale() function takes a non-normalized vector and re-scales it to have the specified magnitude.

The vrl_VectorPrint() routine prints out a message followed by the values of each of the components of the vrl_Vector, enclosed in square brackets. Do not attempt to write to the screen with this routine, since it will not work well in graphics mode.

The vrl_VectorEqual() routine returns a non-zero value if the two vrl_Vectors are identical, and vrl_VectorZero() sets the components of a vrl_Vector to zero. The [0,0,0] vector is sometimes needed, so a global vrl_Vector variable called vrl_VectorNULL is defined.

Matrices

A vrl_Matrix is a 4 by 3 array that stores location and orientation information. All AVRIL matrices are homogeneous; the upper 3 by 3 submatrix stores rotation information and the last 3-element row stores a translation vector. You should never have to deal with the vrl_Matrix type directly. However, in case you do have a need to deal with actual matrices, the following routines are provided:

```
void vrl_MatrixIdentity(vrl_Matrix m);
void vrl_MatrixCopy(vrl_Matrix result, vrl_Matrix m);
void vrl_MatrixMultiply(vrl_Matrix result, vrl_Matrix m1, vrl_Matrix m2);
void vrl_MatrixInverse(vrl_Matrix result, vrl_Matrix m);
void vrl_MatrixRotX(vrl_Matrix m, vrl_Angle angle, vrl_Boolean leftside);
void vrl_MatrixRotY(vrl_Matrix m, vrl_Angle angle, vrl_Boolean leftside);
void vrl_MatrixRotZ(vrl_Matrix m, vrl_Angle angle, vrl_Boolean leftside);
void vrl_MatrixRotVector(vrl_Matrix m, vrl_Angle angle, vrl_Vector vector,
        vrl_Boolean leftside);
void vrl_MatrixResetRotations(vrl_Matrix m);
void vrl_MatrixGetBasis(vrl_Vector v, vrl_Matrix m, int axis);
void vrl_MatrixSetBasis(vrl_Matrix m, vrl_Vector v, int axis);
void vrl_MatrixTranslate(vrl_Matrix result, vrl_Scalar x, vrl_Scalar y, vrl_Scalar z);
void vrl_MatrixSetTranslation(vrl_Matrix result,
        vrl_Scalar x, vrl_Scalar y, vrl_Scalar z);
void vrl_MatrixGetTranslation(vrl_Vector v, vrl_Matrix m);
void vrl_MatrixGetRotations(vrl_Matrix m, vrl_Angle *rx, vrl_Angle *ry, vrl_Angle *rz);
```

The vrl_MatrixIdentity() function sets the matrix to zeroes, except for the diagonal elements which are set to VRL_UNITY. The vrl_MatrixCopy() and vrl_MatrixMultiply() routines are used to copy and multiply matrices, and the vrl_MatrixInverse() routine computes the matrix inverse. The various rotation functions apply a rotation around X, Y, Z or a specified vector by a given angle; the vrl_MatrixResetRotations() routine sets all the rotations to zero. Several of the vrl_Matrix routines use a *leftside* parameter; a non-zero value for this parameter specifies that the transformation should be applied as a pre-multiplication instead of a post-multiplication.

The function vrl_MatrixGetBasis() gets one of the basis vectors of the rotation part of the matrix; this is equivalent to (but faster than) transforming an axis-aligned unit vector by the matrix. In other words, vrl_MatrixGetBasis(v, m, X) is equivalent to transforming the vector [1,0,0] by the rotation part of the matrix *m* and storing the result in the vector *v*.

The vrl_MatrixTranslate() routine applies a translation to the matrix, and vrl_MatrixSetTranslation() sets the actual translation part of the matrix. The vrl_MatrixGetTranslation() routine fills the given vector with the current translation part of the matrix, and vrl_MatrixGetRotations() gets the angles which, when applied in the order Y, X, Z, produces the rotation part of the matrix.

Transforms

You should never have to use any of the transform functions directly; this is all handled for you by AVRIL. A vector can be transformed by a matrix, or each component of the transform (X, Y or Z) can be computed separately:

```
void vrl_Transform(vrl_Vector result, vrl_Matrix m, vrl_Vector v);
vrl_Scalar vrl_TransformX(vrl_Matrix m, vrl_Vector v);
vrl_Scalar vrl_TransformY(vrl_Matrix m, vrl_Vector v);
vrl_Scalar vrl_TransformZ(vrl_Matrix m, vrl_Vector v);
```

Coordinate Systems

AVRIL allows objects to be translated or rotated in five different coordinate systems. This may seem like a lot, but they're easy to get used to. An object can be moved in its own local coordinate system, the coordinate system of the object it's attached to, the "world" coordinate system, the viewer's coordinate system, or the coordinate system of another object.

For example, consider a bicycle on an open train car. The bicycle is facing sideways, so that if you were sitting on it you'd be watching the scenery go by on the right side of the train. The train itself is moving northeast. If we translate the bicycle along the positive Z axis in its local coordinate system, it will travel sideways off the train car, in a south-easterly direction. If we move it in the positive Z direction of its parent, it will move to the rider's left, towards the front of the train (northeast). If we move it in the positive Z direction in the world, it will move due north. If we're looking at it from directly above, moving the bicycle in the viewer's positive Z direction would send it through the train car and down into the ground. If a bird is flying due south, then moving the bicycle in the positive Z direction relative to the bird would make the bike move due south.

We represent these various coordinate systems by the constants VRL_COORD_LOCAL, VRL_COORD_PARENT, VRL_COORD_WORLD, and VRL_COORD_OBJREL. The view-relative coordinate system is just a special case of the VRL_COORD_OBJREL coordinate frame, with the viewer as the object that the movement should be relative to.

Worlds

In AVRIL, a virtual world is a collection of objects, light sources, virtual cameras and miscellaneous attributes. You can have any number of worlds within a single AVRIL application; they're distinct from each other, and you can switch between them whenever you like.

When you run an AVRIL program, a default world is created and initialized for you; if you only plan on having one world in your application, you don't have to do anything special. If you want to create additional worlds, you can simply declare variables of type vrl_World and initialize them by calling vrl_WorldInit(&yourworld); however, it's probably better to dynamically allocate them using vrl_malloc(). In fact, the simplest way to create a world is with the vrl_WorldCreate() function, which allocates the space and initializes the world for you. To make a given world current, use the vrl_WorldSetCurrent() function; the vrl_WorldGetCurrent() function can be used to get a pointer to the current world.

```
vrl_World *vrl_WorldInit(vrl_World *world);
vrl_World *vrl_WorldCreate(void);
void vrl_WorldSetCurrent(vrl_World *world);
vrl_World *vrl_WorldGetCurrent(void);
```

You can easily add objects, light sources and cameras to the current world, and remove them; you can also count how many of each the current world contains, and get pointers to the linked list of lights, linked list of cameras and the hierarchical tree of objects You can also find lights, cameras and objects by name.

```
void vrl_WorldAddLight(vrl_Light *light);
void vrl_WorldRemoveLight(vrl_Light *light);
vrl_Light *vrl_WorldFindLight(char *name);

void vrl_WorldAddCamera(vrl_Camera *camera);
void vrl_WorldRemoveCamera(vrl_Camera *camera);
vrl_Camera *vrl_WorldFindCamera(char *name);

void vrl_WorldAddObject(vrl_Object *obj);
void vrl_WorldRemoveObject(vrl_Object *obj);
vrl_Object *vrl_WorldFindObject(char *name);

int vrl_WorldCountObjects(void);
int vrl_WorldCountLights(void);
int vrl_WorldCountCameras(void);

vrl_Light *vrl_WorldGetLights(void);
vrl_Light *vrl_WorldGetCameras(void);
vrl_Object *vrl_WorldGetObjectTree(void);
```

If you need to iterate through the linked list of lights or cameras, you can use the functions

```
vrl_Light *vrl_LightGetNext(vrl_Light *light);
vrl_Camera *vrl_CameraGetNext(vrl_Camera *camera);
```

You can also obtain information about the total number of facets in the world, the minimum and maximum bounds of the world, the center of the world and the radius of the world's bounding sphere using these functions:

```
int vrl_WorldCountFacets(void);
void vrl_WorldGetBounds(vrl_Vector v1, vrl_Vector v2);
void vrl_WorldGetCenter(vrl_Vector v);
vrl_Scalar vrl_WorldGetSize(void);
```

Each world has a "current camera" through which the world is seen; you can set the current camera, or get a pointer to it using these routines:

```
void vrl_WorldSetCamera(vrl_Camera *cam);
vrl_Camera *vrl_WorldGetCamera(void);
```

The clearing of the screen prior to each frame, and the use (and colors) of the horizon, are controlled by the following functions:

```
void vrl_WorldSetScreenClear(int n);
int vrl_WorldGetScreenClear(void);
void vrl_WorldToggleScreenClear(void);

void vrl_WorldSetHorizon(int n);
int vrl_WorldGetHorizon(void);
void vrl_WorldToggleHorizon(void);

void vrl_WorldSetGroundColor(int color);
int vrl_WorldGetGroundColor(void);

void vrl_WorldSetSkyColor(int color);
int vrl_WorldGetSkyColor(void);
```

The rate at which the user moves and turns is controlled by the "turn" step and the "move" step. In addition, the movement "mode" can be set to 0 or 1; if it's 1 (the default) then simple movement can move the user vertically, otherwise they stay on the ground. Note that these are really only suggestions, and it's up to the application to make use of them.

```
void vrl_WorldSetMovementMode(int n);
int vrl_WorldGetMovementMode(void);
void vrl_WorldToggleMovementMode(void);

void vrl_WorldSetMovestep(vrl_Scalar distance);
vrl_Scalar vrl_WorldGetMovestep(void);

void vrl_WorldSetTurnstep(vrl_Angle angle);
vrl_Angle vrl_WorldGetTurnstep(void);
```

There's a flag, stored in the world data structure, which indicates whether or not the world is being rendered stereoscopically; the following routines access that flag:

```
void vrl_WorldSetStereo(int n);
int vrl_WorldGetStereo(void);
void vrl_WorldToggleStereo(void);
```

The world data structure also stores a pointer to the stereo configuration being used. That pointer can be accessed using the following routines:

```
vrl_WorldSetStereoConfiguration(conf);
vrl_StereoConfiguration *vrl_WorldGetStereoConfiguration(void);
```

In addition to the standard, "cyclopean" camera, there are cameras in the world for the left and right eyes. The functions to access them are:

```
void vrl_WorldSetLeftCamera(cam);
vrl_Camera *vrl_WorldGetLeftCamera(void);
void vrl_WorldSetRightCamera(cam);
vrl_Camera *vrl_WorldGetRightCamera(void);
```

Finally, additional aspects of the virtual world such as the ambient light level and the "scale factor" (the number of real-world millimeters per unit of distance in the virtual world) can be set and queried using the following functions:

```
void vrl_WorldSetAmbient(vrl_Factor ambient);
vrl_Factor vrl_WorldGetAmbient(void);
```

```
void vrl_WorldSetScale(vrl_Scalar scale);
vrl_Scalar vrl_WorldGetScale(void);
```

Objects

Objects are the most important entities in a virtual world. All objects have a location and orientation, and they can be attached to each other in a tree-structured hierarchy. Each object can have a shape (i.e. geometric description) and a surface map. You can create an object statically (by declaring a variable of type vrl_Object) or dynamically (either by using vrl_malloc() to allocate the space and vrl_ObjectInit() to initialize it, or by simply calling vrl_ObjectCreate()). If you use vrl_ObjectCreate(), you can optionally specify a shape for the object to use; if you don't want to assign a shape, use NULL. You can also destroy objects using vrl_ObjectDestroy().

```
vrl_Object *vrl_ObjectInit(vrl_Object *obj);
vrl_Object *vrl_ObjectCreate(vrl_Shape *shape);
void vrl_ObjectDestroy(vrl_Object *object);
```

You can create an exact copy of an object using the function

```
vrl_Object *vrl_ObjectCopy(vrl_Object *obj);
```

Note that the newly-created object will share all the same properties (including the shape and surface map) as the original and will be in the exact same location as the original; you should probably move it. The copy will have nothing attached to it (it doesn't inherit children from the original), and will be a sibling of the original (sharing the same parent, if any).

Objects can be rotated around any of the axes, in any coordinate frame, using the following function:

```
void vrl_ObjectRotate(vrl_Object *obj, vrl_Angle angle, int axis,
       vrl_CoordFrame frame, vrl_Object *relative_to);
```

The *axis* is one of the defined constants X, Y or Z. The *frame* is one of the coordinate frames discussed earlier. If the *frame* is VRL_COORD_OBJREL, then the *relative_to* parameter points to the object that motion should be relative to. For example, to rotate an object 45 degrees around the viewer's Z axis, you would make the following call:

```
vrl_ObjectRotate(obj, float2angle(45), Z,
       VRL_COORD_OBJREL, vrl_CameraGetObject(vrl_WorldGetCamera()));
```

You can also orient an object to "look" in a particular direction using the function

```
void vrl_ObjectLookAt(vrl_Object *obj, vrl_Vector forward, vrl_Vector up);
```

The object will be rotated so that it's Z axis points along the *forward* vector, and its Y axis points in the general direction of the *up* vector. Note that the actual Y orientation may be different, unless you make sure that *up* is perpendicular to *forward*. The *up* and *forward* vectors are specified in world coordinates, and should both be unit vectors; you may find the vrl_VectorNormalize() function handy for this.

Translations of an object are done with the following function:

```
void vrl_ObjectTranslate(vrl_Object *obj, vrl_Vector v,
        vrl_CoordFrame frame, vrl_Object *relative_to);
```

The object is moved along the vrl_Vector *v* in the specified *frame*. The meaning of the *relative_to* parameter is the same as it was for vrl_ObjectRotate().

The vrl_ObjectRotate() and vrl_ObjectTranslate() routines both apply a rotation to the current state of the object. If you wish to make the rotations absolute, call vrl_ObjectRotReset(obj). If you wish to make translations absolute, call vrl_ObjectVectorMove(obj, vrl_VectorNULL) to set the translations to zero. These should both be done before applying the vrl_ObjectRotate() and vrl_ObjectTranslate() functions.

The vrl_ObjectRotate() and vrl_ObjectTranslate() functions should be used for all object rotation and translation. Some older functions are also provided for rotating and moving objects relative to their parent (one of the more common cases); they are as follows:

```
void vrl_ObjectMove(vrl_Object *obj, vrl_Scalar x, vrl_Scalar y, vrl_Scalar z);
void vrl_ObjectRelMove(vrl_Object *obj, vrl_Scalar x, vrl_Scalar y, vrl_Scalar z);
void vrl_ObjectRotX(vrl_Object *obj, vrl_Angle angle);
void vrl_ObjectRotY(vrl_Object *obj, vrl_Angle angle);
void vrl_ObjectRotZ(vrl_Object *obj, vrl_Angle angle);
void vrl_ObjectRotVector(vrl_Object *obj, vrl_Angle angle, vrl_Vector vector);
void vrl_ObjectRotReset(vrl_Object *obj);
void vrl_ObjectVectorMove(vrl_Object *obj, vrl_Vector v);
void vrl_ObjectVectorRelMove(vrl_Object *obj, vrl_Vector v);
```

An object's current location can be obtained in two ways, either component-by-component for each of X, Y and Z, or copied into a vector:

```
vrl_Scalar vrl_ObjectGetWorldX(vrl_Object *object);
vrl_Scalar vrl_ObjectGetWorldY(vrl_Object *object);
vrl_Scalar vrl_ObjectGetWorldZ(vrl_Object *object);
void vrl_ObjectGetWorldLocation(vrl_Object *object, vrl_Vector v);

vrl_Scalar vrl_ObjectGetRelativeX(vrl_Object *object);
vrl_Scalar vrl_ObjectGetRelativeY(vrl_Object *object);
vrl_Scalar vrl_ObjectGetRelativeZ(vrl_Object *object);
void vrl_ObjectGetRelativeLocation(vrl_Object *object, vrl_Vector v);
```

The World versions of the functions return the absolute world coordinates; the Relative versions return the coordinates relative to the object's parent.

The rotation angles of objects, either relative to the world or to their parent, can be obtained using the following routines:

```
void vrl_ObjectGetWorldRotations(vrl_Object *object,
        vrl_Angle *rx, vrl_Angle *ry, vrl_Angle *rz);
void vrl_ObjectGetRelativeRotations(vrl_Object *object,
        vrl_Angle *rx, vrl_Angle *ry, vrl_Angle *rz);
```

The current world-space orientation of an object's "forward", "up" and "right" vectors can be obtained using the following routines:

```
void vrl_ObjectGetForwardVector(vrl_Object *object, vrl_Vector v);
void vrl_ObjectGetRightVector(vrl_Object *object, vrl_Vector v);
void vrl_ObjectGetUpVector(vrl_Object *object, vrl_Vector v);
```

The vectors filled in by these routines will all be normalized.

An object can be attached to another object, or detached from whatever object it is currently attached to; you can also find out the "parent" of the object:

```
vrl_Object *vrl_ObjectAttach(vrl_Object *obj, vrl_Object *newparent);
vrl_Object *vrl_ObjectDetach(vrl_Object *obj);
vrl_Object *vrl_ObjectGetParent(vrl_Object *obj);
```

The vrl_ObjectAttach() and vrl_ObjectDetach() functions return a pointer to the object's previous parent if any. Note that movement and rotation in the VRL_COORD_PARENT system (including that performed using vrl_ObjectRotX() and other similar functions) is carried out relative to the object's parent. In other words, if the object is attached to another object, its location and orientation will depend on that of its parent; if the parent moves, the child will move with it. However, if the child moves the parent will stay where it is.

You can find the "root" of an object tree using the following function:

```
vrl_Object *vrl_ObjectFindRoot(vrl_Object *obj);
```

You can walk an entire object tree, executing a function on each node of the tree, using the following routine:

```
void vrl_ObjectTraverse(vrl_Object *object, int (*function)(vrl_Object *obj));
```

The function is called once for each object in the hierarchy, and is given a pointer to the object it's being called on; if the function returns a non-zero value at any point, the tree is not processed any further. All parent objects are processed before their descendants.

The distance between two objects can be found using

```
vrl_Scalar vrl_ObjectComputeDistance(vrl_Object *obj1, vrl_Object *obj2);
```

The shape and surface map of an object can be altered at any time, and as often as needed, using the following routines:

```
void vrl_ObjectSetShape(vrl_Object *object, vrl_Shape *shape);
vrl_Shape *vrl_ObjectGetShape(vrl_Object *object);

void vrl_ObjectSetSurfacemap(vrl_Object *object, vrl_Surfacemap *map);
vrl_Surfacemap *vrl_ObjectGetSurfacemap(vrl_Object *object);
```

Objects can be flagged as invisible (in which case they're not drawn) or highlighted (in which case they're drawn with a bright outline). They can also have a "layer" property, and individual layers can be made visible or invisible, as described later in the section on Layers. Note that layer zero is always visible; in effect, an object whose layer is zero will appear on all layers. The following routines are used to set, query and toggle those values:

```
void vrl_ObjectSetVisibility(vrl_Object *object, int vis);
int vrl_ObjectGetVisibility(vrl_Object *object);
void vrl_ObjectToggleVisibility(vrl_Object *object);

void vrl_ObjectSetHighlight(vrl_Object *object, highlight);
int vrl_ObjectGetHighlight(vrl_Object *object);
void vrl_ObjectToggleHighlight(vrl_Object *object);

void vrl_ObjectSetLayer(vrl_Object *object, int layer);
int vrl_ObjectGetLayer(vrl_Object *object);
```

AVRIL supports the idea of "fixed" objects; you can mark an object as fixed or movable, and find out its current status, using the following functions:

```
void vrl_ObjectMakeFixed(vrl_Object *object);
void vrl_ObjectMakeMovable(vrl_Object *object);
vrl_Boolean vrl_ObjectIsFixed(vrl_Object *object);
```

You can also find out the boundaries of an object (in world coordinates) using the functions

```
void vrl_ObjectGetMinbounds(vrl_Object *object, vrl_Vector v);
void vrl_ObjectGetMaxbounds(vrl_Object *object, vrl_Vector v);
```

The vectors returned by these two functions can be thought of as the opposite corners of the object's bounding box.

AVRIL will normally select a level of detail for an object automatically; however, you can override this mechanism on an object-by-object basis using two routines to set and get the current "forced" rep for an object:

```
void vrl_ObjectSetRep(vrl_Object *object, vrl_Rep *rep);
vrl_Rep *vrl_ObjectGetRep(vrl_Object *object);
```

If you want automatic representation selection to be re-enabled for the object, just use vrl_ObjectSetRep(obj, NULL). See the section on Representations for details.

Whenever an object moves, all the objects "descended" from that object must be updated. The following function will update the object and all its descendants:

```
vrl_Object *vrl_ObjectUpdate(vrl_Object *object);
```

You should generally only call this once per frame, on the object tree for the current world; the macro vrl_WorldUpdate() can be used to do this more concisely.

A vrl_Object can have several other properties associated with it. These include a name, a function, and some application-specific data. The function associated with an object gets called whenever the object is processed during the tree-walking that vrl_ObjectUpdate() performs. The following routines allow you to get and set these additional properties:

```
void vrl_ObjectSetName(vrl_Object *obj, char *str);
char *vrl_ObjectGetName(vrl_Object *obj);
void vrl_ObjectSetFunction(vrl_Object *obj, vrl_ObjectFunction fn);
vrl_ObjectFunction *vrl_ObjectGetFunction(vrl_Object *obj);
void vrl_ObjectSetApplicationData(vrl_Object *obj, void *data);
void *vrl_ObjectGetApplicationData(vrl_Object *obj);
```

Shapes

As described earlier, AVRIL keeps shape information separate from object descriptions, so that shapes can be re-used by multiple objects. Shapes (entities of type vrl_Shape) are generally read from PLG files using the vrl_ReadPLG() function, described later. You can also create them using the vrl_Primitive family of functions, also described later in this document. The syntax for PLG files is described in Appendix C.

You can modify a shape after it's been loaded; bear in mind that any changes you make to a shape will affect all objects using that shape! To re-scale a shape, or shift all the vertices in the shape relative to the shape's origin point, use the following functions:

```
void vrl_ShapeRescale(vrl_Shape *shape, float sx, float sy, float sz);
void vrl_ShapeOffset(vrl_Shape *shape, vrl_Scalar tx, vrl_Scalar ty, vrl_Scalar tz);
```

After making changes to a shape (or any representation within a shape), you should call vrl_ShapeUpdate() to recompute the shape's bounds.

```
void vrl_ShapeUpdate(vrl_Shape *shape);
```

The vrl_ShapeRescale() and vrl_ShapeOffset() routines call vrl_ShapeUpdate() automatically, so you don't have to do it again; it's only when you move individual vertices that you need to worry about it.

Shapes can have a default surface map, which is used for objects that don't set one of their own. A pointer to a shape's default surface map can be obtained, or a pointer to a new surfacemap for a shape set, by calling the functions

```
vrl_Surfacemap *vrl_ShapeGetSurfacemap(vrl_Shape *shape);
void vrl_ShapeSetSurfacemap(vrl_Shape *shape, vrl_Surfacemap *map);
```

To get a pointer to the representation of a shape that will be used at a given on-screen size, use following function:

```
vrl_Rep *vrl_ShapeGetRep(vrl_Shape *shape, vrl_Scalar size);
```

To add an additional representation to an existing shape, use the function

```
void vrl_ShapeAddRep(vrl_Shape *shape, vrl_Rep *rep, vrl_Scalar size);
```

The *size* parameter gives the apparent on-screen size in pixels at which the shape should be used. You can find out how many representations a shape has using the function

```
int vrl_ShapeCountReps(vrl_Shape *shape);
```

A shape's name can be set or obtained using the functions

```
void vrl_ShapeSetName(vrl_Shape *shape, char *str);
char *vrl_ShapeGetName(vrl_Shape *shape);
```

Shapes are kept internally in a singly-linked list; if you need to iterate through the list, the following two functions can be used

```
vrl_Shape *vrl_ShapeGetList(void);
vrl_Shape *vrl_ShapeGetNext(vrl_Shape *shape)
```

You can also locate a shape by name using

```
vrl_Shape *vrl_ShapeFind(char *name);
```

Representations

A shape can have any number of representations, at various levels of detail. Each representation (vrl_Rep) has a set of vertices (each of type vrl_Vector) and a set of facets (each of type vrl_Facet). A representation can also have a "sorting type" field; this will be explained in more detail in future releases of this documentation.

You can traverse the list of representations for a shape, calling a function on each representation, by using the following routine:

```
void vrl_ShapeTraverseReps(vrl_Shape *shape, int (*function(vrl_Rep *rep)));
```

The function is called once for every representation, and is given the representation as a parameter. If the function returns a non-zero value, the processing of the representation list stops at that rep.

The other approach is to iterate through the linked list of representations using the functions

```
vrl_Rep *vrl_ShapeGetFirstRep(vrl_Shape *shape);
vrl_Rep *vrl_RepGetNext(vrl_Rep *rep);
```

You can set and get a vrl_Rep's sorting type, find out the approximate size (in pixels) at which a rep becomes effective, as well as count the number of vertices and facets in a rep using the following functions:

```
void vrl_RepSetSorting(vrl_Rep *rep, int type);
int vrl_RepGetSorting(vrl_Rep *rep);

int vrl_RepGetSize(vrl_Rep *rep);
```

```
int vrl_RepCountVertices(vrl_Rep *rep);
int vrl_RepCountFacets(vrl_Rep *rep);
```

There are also "traversal" functions for vertices and facets:

```
void vrl_RepTraverseVertices(vrl_Rep *rep, int (*function)(vrl_Vector *vertex));
void vrl_RepTraverseFacets(vrl_Rep *rep, int (*function)(vrl_Facet *facet));
```

If you need to get or set the values of a vertex's coordinates, you can use the functions

```
void vrl_RepGetVertex(vrl_Rep *rep, int n, vrl_Vector v);
void vrl_RepSetVertex(vrl_Rep *rep, int n, vrl_Vector v);
```

Be careful when using vrl_RepSetVertex(); it's easy to move a vertex and create non-planar or non-convex facets, which confuse the renderer. You can only move vertices safely if you know the vrl_Rep is composed entirely of triangles, since by their nature triangles are always planar and convex. In any case, be sure to call vrl_ShapeUpdate() after moving any vertices.

To support Gouraud shading, functions are provided to compute vertex normals by averaging the normal vectors of all the polys which share the vertex in a vrl_Rep or a vrl_Shape:

```
void vrl_RepComputeVertexNormals(vrl_Rep *rep);
void vrl_ShapeComputeVertexNormals(vrl_Shape *shape);
```

Make sure that the polygons normals have already been computed (by the vrl_ShapeUpdate() function) before calling vrl_RepComputeVertexNormals().

There's also a function for computing edge information for a representation; currently, this information is not used:

```
void vrl_RepBuildEdges(vrl_Rep *rep);
```

Facets

AVRIL's terminology is slightly different from some other VR systems; a "facet" is a flat three-dimensional entity, whereas a "polygon" is a two dimensional area on the screen. The job of the graphics pipeline is to turn facets into polygons.

Facets in AVRIL have an array of integers that specify which vertices in the representation should be connected (and in what sequence) to form the outline of the facet. Facets also have an index into the surface map for an object, to determine what the surface properties of the facet should be. They also have a flag that indicates whether or not the facet should be highlighted.

The surface index of any facet can be set or queried at any time using the following two routines:

```
void vrl_FacetSetSurfnum(vrl_Facet *facet, int n);
int void vrl_FacetGetSurfnum(vrl_Facet *facet);
```

The highlighting of the facet can be set, queried or toggled using the following routines:

```
void vrl_FacetSetHighlight(vrl_Facet *facet, int high);
int vrl_FacetGetHighlight(vrl_Facet *facet);
void vrl_FacetToggleHighlight(vrl_Facet *facet);
```

The number of points in the facet, the index of any particular point, or a pointer to the vertex for a particular point, can all be obtained using these routines:

```
int vrl_FacetCountPoints(vrl_Facet *facet);
int vrl_FacetGetPoint(vrl_Facet *facet, int n);
vrl_Vector *vrl_FacetGetVertex(vrl_Rep *rep, vrl_Facet *facet, int n);
```

vrl_Facets can be identified by an ID number. The ID number for a vrl_Facet can be set and read, and a vrl_Facet with a particular ID can be found, using the following functions:

```
void vrl_FacetSetId(vrl_Facet *facet, vrl_unsigned16bit n);
vrl_unsigned16bit vrl_FacetGetId(vrl_Facet *facet);
vrl_Facet *vrl_RepFindFacet(vrl_Rep *rep, vrl_unsigned16bit id);
```

Surfaces

AVRIL surfaces are designed for expandability. At the moment, each vrl_Surface consists of a type, a hue and a brightness. The types are SURF_SIMPLE (no lighting, just a fixed color), SURF_FLAT (for flat shading), SURF_GOURAUD (for Gouraud shading), SURF_METAL (for a pseudo-metallic effect) and SURF_GLASS (for a partially transparent effect). Surfaces are initialized and modified using the following routines:

```
vrl_Surface *vrl_SurfaceInit(vrl_Surface *surf);
vrl_Surface *vrl_SurfaceCreate(vrl_unsigned8bit hue);

void vrl_SurfaceSetType(vrl_Surface *surf, vrl_LightingType type);
vrl_LightingType vrl_SurfaceGetType(vrl_Surface *surf);

void vrl_SurfaceSetHue(vrl_Surface *surf, unsigned char h);
unsigned char vrl_SurfaceGetHue(vrl_Surface *surf);

void vrl_SurfaceSetBrightness(vrl_Surface *surf, unsigned char b);
unsigned char vrl_SurfaceGetBrightness(vrl_Surface *surf);
```

The hue and brightness values are 8-bit unsigned quantities; the maximum brightness value is therefore 255.

Future versions of AVRIL may support specular shading; the following two functions allow you to set and get the specular exponent value, which controls the "sharpness" of the specular highlights:

```
void vrl_SurfaceSetExponent(vrl_Surface *surf, vrl_Exponent exp);
vrl_Exponent vrl_SurfaceGetExponent(vrl_Surface *surf);
```

For backwards compatibility with REND386, AVRIL includes functions to convert a 16-bit REND386 surface descriptor into a vrl_Surface, and vice-versa:

```
vrl_Surface *vrl_SurfaceFromDesc(vrl_unsigned16bit desc, vrl_Surface *surf);
vrl_unsigned16bit vrl_SurfaceToDesc(vrl_Surface *surf);
```

As surfaces are created, they are added to a list; the following functions allow you to iterate through the list:

```
vrl_Surface *vrl_SurfaceGetList(void);
vrl_Surface *vrl_SurfaceGetNext(vrl_Surface *surf);
```

Surface Maps

Surface maps contain an array of pointers to surfaces; you can create a surface map with room for a particular number of entries, find out how many entries the map contains, and access entries within a map, using the following routines:

```
vrl_Surfacemap *vrl_SurfacemapCreate(int n);
int vrl_SurfacemapCountEntries(vrl_Surfacemap *map);
vrl_Surface *vrl_SurfacemapGetSurface(vrl_Surfacemap *map, int surfnum);
vrl_Surface *vrl_SurfacemapSetSurface(vrl_Surfacemap *map, int surfnum,
        vrl_Surface *surface);
```

As surfacemaps are defined, they are added to a list; the list can be iterated through using the following functions:

```
vrl_Surfacemap *vrl_SurfacemapGetList(void);
vrl_Surfacemap *vrl_SurfacemapGetNext(vrl_Surfacemap *map);
```

Lights

Lights in AVRIL have a number of properties; they can be on or off, they can have an intensity, they can have a "type", and they can be associated with an object. The on/off and intensity properties are similar to a household dimmer; rotating the knob on a dimmer alters the intensity, and pushing it in toggles the light on and off.

The current version of AVRIL only supports ambient lights and directional lights; point sources will be supported soon. Any light that is not associated with an object is considered ambient; this is in addition to the overall ambient light level for the world. A directional light uses the orientation of the object it's associated with to determine which direction the light should come from. A point source light (once implemented) will use the location of the object it's associated with to determine where the light comes from.

As with worlds and objects, lights can be statically or dynamically created and destroyed using the following functions:

```
vrl_Light *vrl_LightInit(vrl_Light *light);
vrl_Light *vrl_LightCreate(void);
void vrl_LightDestroy(vrl_Light *light);
```

The light's *type* value can be one of LIGHT_AMBIENT, LIGHT_DIRECTIONAL or LIGHT_POINTSOURCE, and is set and queried using the following two functions:

```
void vrl_LightSetType(vrl_Light *light, int type);
int vrl_LightGetType(vrl_Light *light);
```

The light's on/off status can be checked and altered, and the intensity set and queried, using these functions:

```
void vrl_LightOn(vrl_Light *light);
void vrl_LightOff(vrl_Light *light);
void vrl_LightToggle(vrl_Light *light);
vrl_Boolean vrl_LightIsOn(vrl_Light *light);

void vrl_LightSetIntensity(vrl_Light *light, vrl_Factor inten);
vrl_Factor vrl_LightGetIntensity(vrl_Light *light);
```

Notice that the intensity values are vrl_Factors; they should never be less than zero or greater than VRL_UNITY.

You can make and break associations between a light source and an object, and determine what object a light source is currently associated with, using the following routines:

```
void vrl_LightAssociate(vrl_Light *light, vrl_Object *object);
void vrl_LightDisAssociate(vrl_Light *light);
vrl_Object *vrl_LightGetObject(vrl_Light *light);
```

Many of the routines that were used for objects earlier have counterparts that are used for light sources; they're implemented as macros that just perform the operations on the object with which the light source is associated.

```
vrl_LightMove(light, x, y, z);
vrl_LightRelMove(light, x, y, z);
vrl_LightVectorMove(light, v);
vrl_LightVectorRelMove(light, v);
vrl_LightRotX(light, angle);
vrl_LightRotY(light, angle);
vrl_LightRotZ(light, angle);
vrl_LightRotVector(light, angle, vector);
vrl_LightRotReset(light);
vrl_LightRotate(light, angle, axis, frame, relative_to);
vrl_LightTranslate(light, v, axis, frame, relative_to);
vrl_LightLookAt(light, forward, up);
vrl_LightAttach(obj, newparent);
vrl_LightDetach(obj);
vrl_LightGetWorldX(light);
vrl_LightGetWorldY(light);
vrl_LightGetWorldZ(light);
vrl_LightGetWorldLocation(light, v);
vrl_LightGetWorldRotations(light, rx, ry, rz);
vrl_LightGetRelativeX(light);
vrl_LightGetRelativeY(light);
vrl_LightGetRelativeZ(light);
vrl_LightGetRelativeLocation(light, v);
vrl_LightGetRelativeRotations(light, rx, ry, rz);
```

It's important to note the difference between attaching and associating light sources. A light source can be associated with an object, which means it will use that object's location and orientation as its own. The object with which the light is associated can be attached to another object, and "inherit" location and orientation information from it. The vrl_LightAttach() and vrl_LightDetach() routines are provided only as a convenience; what you're really attaching and detaching with those routines is the object that the light is associated with. You generally associate a light source with an object once, and leave it that way; you can attach or detach the light however you want after that.

Lights can have other attributes as well, just as objects can; specifically, they can have a name and some application-specific data, which are accessed using the following functions:

```
void vrl_LightSetName(vrl_Light *light, char *str);
char *vrl_LightGetName(vrl_Light *light);
void vrl_LightSetApplicationData(vrl_Light *light, void *data);
void *vrl_LightGetApplicationData(vrl_Light *light);
```

Cameras

AVRIL allows you to have any number of virtual cameras. Each camera is associated with an object, much as lights are. However, unlike lights, cameras *must* be associated with an object; there's no such thing as an "ambient" camera. Cameras are initialized and destroyed just like objects or light sources:

```
vrl_Camera *vrl_CameraInit(vrl_Camera *camera);
vrl_Camera *vrl_CameraCreate(void);
void vrl_CameraDestroy(vrl_Camera *camera);
```

Cameras have only a few properties that are important; in particular, a zoom factor, an aspect ratio, and hither and yon clipping plane distances. These are all set and queried using the following routines:

```
void vrl_CameraSetZoom(vrl_Camera *camera, float zoom);
float vrl_CameraGetZoom(vrl_Camera *camera);

void vrl_CameraSetAspect(vrl_Camera *camera, float asp);
float vrl_CameraGetAspect(vrl_Camera *camera);

void vrl_CameraSetHither(vrl_Camera *camera, vrl_Scalar hither);
vrl_Scalar vrl_CameraGetHither(vrl_Camera *camera);

void vrl_CameraSetYon(vrl_Camera *camera, vrl_Scalar yon);
vrl_Scalar vrl_CameraGetYon(vrl_Camera *camera);
```

Notice that the zoom factor and aspect ratio are floats; this may change in a future release of AVRIL. The zoom factor works like the zoom on a camera; the higher the zoom, the more the image is magnified. The zoom is the tangent of half the field of view. The aspect ratio is the ratio between the horizontal and vertical zoom factors. The hither clipping distance is the distance in virtual space from the camera to the invisible plane at which objects will be "clipped". The "yon" distance is like an invisible wall; any object entirely on the far side of the wall will not be seen.

The routines for associating a camera with an object and for determining what object a camera is currently associated with are as follows:

```
void vrl_CameraAssociate(vrl_Camera *camera, vrl_Object *object);
vrl_Object *vrl_CameraGetObject(vrl_Camera *camera);
```

There's no vrl_CameraDisAssociate() function, as there was for lights; cameras must be associated with an object in order to have any meaning.

Again, routines are provided for manipulating and querying the location and orientation of a virtual camera; these are macros, just as they were for lights:

```
vrl_CameraMove(camera, x, y, z);
vrl_CameraRelMove(camera, x, y, z);
vrl_CameraVectorMove(camera, v);
vrl_CameraVectorRelMove(camera, v);
vrl_CameraRotX(camera, angle);
vrl_CameraRotY(camera, angle);
vrl_CameraRotZ(camera, angle);
vrl_CameraRotVector(camera, angle, vector);
vrl_CameraRotReset(camera);
vrl_CameraRotate(camera, angle, axis, frame, relative_to);
vrl_CameraLookAt(camera, forward, up);
vrl_CameraTranslate(camera, v, axis, frame, relative_to);
vrl_CameraAttach(obj, newparent);
vrl_CameraDetach(obj);
vrl_CameraGetWorldX(camera);
vrl_CameraGetWorldY(camera);
vrl_CameraGetWorldZ(camera);
vrl_CameraGetWorldLocation(camera, v);
vrl_CameraGetWorldRotations(camera, rx, ry, rz);
vrl_CameraGetRelativeX(camera);
vrl_CameraGetRelativeY(camera);
vrl_CameraGetRelativeZ(camera);
vrl_CameraGetRelativeLocation(camera, v);
vrl_CameraGetRelativeRotations(camera, rx, ry, rz);
```

Camera can have other attributes as well, just as objects and lights can; specifically, they can have a name and some application-specific data, which are accessed using the following functions:

```
void vrl_CameraSetName(vrl_Camera *camera, char *str);
char *vrl_CameraGetName(vrl_Camera *camera);
void vrl_CameraSetApplicationData(vrl_Camera *camera, void *data);
void *vrl_CameraGetApplicationData(vrl_Camera *camera);
```

In addition, there are three routines that obtain the current "forward", "right" and "up" vectors for a camera:

```
vrl_CameraGetForwardVector(camera, v)
vrl_CameraGetRightVector(camera, v)
vrl_CameraGetUpVector(camera, v)
```

Layers

Layers were described earlier, in the section on objects. The routines for dealing with layers are as follows:

```
void vrl_LayerOn(int n);
void vrl_LayerOff(int n);
void vrl_LayerToggle(int n);
int vrl_LayerIsOn(int n);
void vrl_LayerAllOn(void);
void vrl_LayerAllOff(void);
```

The last two routines, vrl_LayerAllOn() and vrl_LayerAllOff(), turn all the layers on or off at once. By default, all layers are on.

Stereoscopic Rendering

Stereoscopic rendering consists of generating two images, one for the left eye and one for the right. The world data structure maintains a flag indicating whether stereoscopic rendering should be used, as well as pointers to the left-eye and right-eye cameras and the stereo configuration information; see the section on Worlds for more information.

There are a number of ways of transferring the separate left and right images to the viewer's eyes; among them are anaglyph techniques that use red and blue filters, field sequential techniques that use shutter glasses, optical techniques that use a split screen, alternate-scanline techniques, and systems that use more than one VGA card.

In addition, there are systems that provide stereoscopic depth without using separate images: the patented "Chromadepth" technique used in some comic books and laser light shows is one, and SIRDS (Single-Image Random Dot Stereograms, or "Magic Eye" pictures) are another.

In AVRIL, information about the current stereo configuration is stored in a structure of type vrl_StereoConfiguration. Such a structure can be created (or initialized) using the following functions:

```
vrl_StereoConfiguration *vrl_StereoCreateConfiguration(void);
vrl_StereoConfiguration *vrl_StereoInitConfiguration(vrl_StereoConfiguration *conf);
```

The type of stereoscopic viewing is defined by a constant; the possible values are listed in Appendix I. Setting and getting the type of stereoscopic viewing is done using the following functions:

```
void vrl_StereoSetType(vrl_StereoConfiguration *conf, vrl_StereoType st_type);
vrl_StereoType vrl_StereoGetType(vrl_StereoConfiguration *conf);
```

Some systems, like Chromadepth and SIRDS, use a single "eye"; most others use two eyes. There's a function for determining the number of "eyes" used by the current stereo type:

```
int vrl_StereoGetNeyes(vrl_StereoConfiguration *conf);
```

You can alter the amount by which each eye's image is shifted on the screen using the following functions:

```
void vrl_StereoSetLeftEyeShift(vrl_StereoConfigurtion *conf, int shift);
int vrl_StereoGetLeftEyeShift(vrl_StereoConfigurtion *conf);
void vrl_StereoSetRightEyeShift(vrl_StereoConfigurtion *conf, int shift);
int vrl_StereoGetRightEyeShift(vrl_StereoConfigurtion *conf);
```

This shift is necessary, due to differences between HMDs. The total shift (the sum of that which is computed and that which is explicitly set by the functions above) can be obtained using the following functions:

```
int vrl_StereoGetTotalLeftShift(vrl_StereoConfiguration *conf);
int vrl_StereoGetTotalRightShift(vrl_StereoConfiguration *conf);
```

It may be necessary, because of the way an HMD's optics are constructed, to rotate the virtual eyes inwards or outwards. This is done using the following functions:

```
void vrl_StereoSetLeftEyeRotation(vrl_StereoConfiguration *conf, vrl_Angle rot);
vrl_Angle vrl_StereoGetLeftEyeRotation(vrl_StereoConfiguration *conf);
void vrl_StereoSetRightEyeRotation(vrl_StereoConfiguration *conf, vrl_Angle rot);
vrl_Angle vrl_StereoGetRightEyeRotation(vrl_StereoConfiguration *conf);
```

The eye spacing and convergence distance (both in world units) can be manipulated using the following functions:

```
void vrl_StereoSetEyespacing(vrl_StereoConfiguration *conf, float spacing);
float vrl_StereoGetEyespacing(vrl_StereoConfiguration *conf);
void vrl_StereoSetConvergence(vrl_StereoConfiguration *conf, float conv);
float vrl_StereoGetConvergence(vrl_StereoConfiguration *conf);
```

By using various values for eyespacing and convergence distance, a strong or weaker stereo effect can be achieved.

The Chromadepth technique is unusual, in that it uses distance to compute a color between extreme red and extreme blue. Since only a finite number of palette entries is available, it's necessary to optimize their use over a range of distance; this range is represented by the "ChromaNear" and "ChromaFar" values, which can be altered using the following functions:

```
void vrl_StereoSetChromaNear(vrl_StereoConfiguration *conf, vrl_Scalar val);
vrl_Scalar vrl_StereoGetChromaNear(vrl_StereoConfiguration *conf);
void vrl_StereoSetChromaFar(vrl_StereoConfiguration *conf, vrl_Scalar val);
vrl_Scalar vrl_StereoGetChromaFar(vrl_StereoConfiguration *conf);
```

After altering any of the values in a vrl_StereoConfiguration structure, the following function should be called:

```
int vrl_StereoConfigure(vrl_StereoConfiguration *conf);
```

In order for the system to render a scene stereoscopically, both the left-eye and right-eye cameras must exist in the current world; to create them initially, use the following function:

```
int vrl_StereoSetup(void);   /* creates a pair of cameras */
```

There are a number of routines in the scan-conversion module (i.e., the display driver) that relate to stereoscopic viewing; they are described in the section on display drivers.

For more information about stereoscopic rendering and its implementation, see the source code in system.c (which makes all the appropriate calls).

File I/O Routines

AVRIL supports the PLG file format, the FIG file format, and most of the WLD file format; these formats are described in the Appendices. The library contains routines for reading each of those formats:

```
vrl_Shape *vrl_ReadPLG(FILE *in);
vrl_Object *vrl_ReadObjectPLG(FILE *in);
int vrl_ReadWLD(FILE *in);
vrl_Object *vrl_ReadFIG(FILE *in, vrl_Object *parent, char *rootname);
```

The vrl_ReadPLG() routine reads a shape from the specified file and returns a pointer to it. The vrl_ReadObjectPLG() routine is similar, but it also allocates an object and assigns the shape to it.

The vrl_ReadWLD() function reads a world description from the file into the current world; you can make as many calls to this routine as you like, combining a number of WLD files. While reading a WLD file, statements may be encountered which the WLD parser doesn't recognize; these are passed to an application-defined function called

```
void vrl_ReadWLDfeature(int argc, char *argv[], char *rawtext);
```

The *argc* and *argv[]* parameters are just like the ones passed to a main() function in C; the *rawtext* parameter is the original input line.

The vrl_ReadFIG() routine lets you specify a "parent" object to which the newly-read object tree should be attached, as well as the name of the root object. Any segment names (segnames) that are assigned in the FIG file will be added to the current world's list of objects as *rootname.segname*.

All the routines listed above assume you've already opened the file; there are also routines for loading objects, figures and entire worlds from files, given the filename:

```
vrl_Object *vrl_ObjectLoadPLGfile(char *filename);
vrl_Object *vrl_ObjectLoadFIGfile(char *filename);
int *vrl_LoadWLDfile(char *filename);
```

While loading a PLG file, a scale factor and offset can be applied. The vertices read from the file are multiplied by the scaling factors, and then the offsets are added to them. The scale factors and offsets are set using:

```
void vrl_SetReadPLGscale(float x, float y, float z);
void vrl_SetReadPLGoffset(float x, float y, float z);
```

FIG files can also have a scale factor applied to them. In addition, parts of a figure that have a "segnum" value set can have pointers to their objects placed into a parts array specified by the user:

```
void vrl_SetReadFIGscale(float x, float y, float z);
void vrl_SetReadFIGpartArray(vrl_Object **ptr, int maxparts);
```

If the *ptr* is not NULL, then any parts in the FIG file that have a segnum will create an entry in the array, indexed by the segnum value. The *maxparts* value is the number of elements the caller has provided space for in the array.

There are several other routines that support file operations. Two routines maintain a kind of "current directory" for file loading; they are

```
void vrl_FileSetLoadpath(char *path);
char *vrl_FileFixupFilename(char *fname);
```

The first sets the given path (if not NULL) to be the directory that subsequent filename fixups should use. The second routine is used to generate a full filename, with the current loadpath prepended. Note that filenames beginning with '/' or '\' are not modified. Also note that vrl_FileFixupFilename() returns a pointer to an internal buffer, which will be rewritten on the next call to vrl_FileFixupFilename(). If you really need to keep the fixed-up filename around, you should strcpy() it to another buffer or strdup() it.

System and Application routines

These routines are described in detail in the Tutorial, but here's a quick summary:

```
vrl_Boolean vrl_SystemStartup(void);
void vrl_SystemRun(void);
vrl_RenderStatus *vrl_SystemRender(vrl_Object *list);
vrl_Time vrl_SystemGetRenderTime(void);
vrl_Time vrl_SystemGetFrameRate(void);
void vrl_SystemCommandLine(int argc, char *argv[]);

void vrl_SystemRequestRefresh(void);
vrl_Boolean vrl_SystemQueryRefresh(void);

void vrl_SystemStartRunning(void);
void vrl_SystemStopRunning(void);
vrl_Boolean vrl_SystemIsRunning(void);

void vrl_ApplicationDrawUnder(void);
void vrl_ApplicationDrawOver(vrl_RenderStatus *stat);
void vrl_ApplicationInit(void);
void vrl_ApplicationKey(vrl_unsigned16bit c);
void vrl_ApplicationMouseUp(int x, int y, unsigned int buttons);
```

These are not really part of AVRIL's "guts", since you don't need to use anything in system.c (which is the only module that knows about the vrl_Application functions).

User Interface

The current version of AVRIL has a few primitive user interface routines for you to use. A better user interface needs to be designed; in the meantime, here are the routines:

```
void vrl_UserInterfaceBox(int width, int height, int *x, int *y);
void vrl_UserInterfacePopMsg(char *msg);
void vrl_UserInterfacePopText(char *text[]);
```

```
int vrl_UserInterfaceDismiss(void);
int vrl_UserInterfacePopMenu(char *text[]);
int vrl_UserInterfaceMenuDispatch(char *text[], int (**funcs)(void));
vrl_unsigned16bit vrl_UserInterfacePopPrompt(char *prompt, char *buff, int n);
```

The vrl_UserInterfaceBox() routine puts up a nice bordered box, centered on the screen. The *width* and *height* determine the size of the box. When the routine returns, *x* and *y* will contain the screen coordinates of the top-left corner of the box. Either can be NULL, indicating that you don't care about the values.

The vrl_UserInterfacePopMsg() routine displays a one-line text message. The vrl_UserInterfacePopText() routine puts up a multi-line message; the array of string pointers has to have a NULL pointer entry at the end.

The vrl_UserInterfaceDismiss() routine is useful after you've called either vrl_UserInteracePopMsg() or vrl_UserInterfacePopText(); it waits for the user to press a key or click the mouse.

The vrl_UserInterfacePopMenu() routine displays a menu and waits for the user to select an item. If the user clicks on an item with the mouse, the index of that item will be returned as the value of the function. If the user presses a key, the menu is searched item by item until one is found that has an uppercase letter matching the key the user entered; the index of that entry is returned. If the user clicks outside the menu, or presses ESC, the value -1 is returned.

The vrl_UserInterfaceMenuDispatch() routine is similar to vrl_UserInterfacePopMenu(), but it takes an array of pointers to functions as a second parameter. When an item in the menu is selected, the corresponding function is called. No parameters are passed to that function.

The vrl_UserInterfacePopPrompt() box displays a prompt to the user and lets them enter a text response. The backspace key is supported. The user can end their input using either ENTER or ESC; the key they press to end their input is returned as the value of the function.

There are two other routines which are not really part of the user interface; they're used to overlay text on the screen or display the compass. They're typically called from vrl_ApplicationDrawOver().

```
void vrl_UserInterfaceDrawCompass(vrl_Camera *camera, int x, int y, int armlen);
void vrl_UserInterfaceDropText(int x, int y, vrl_Color, char *text);
```

In vrl_UserInterfaceDrawCompass(), the *x* and *y* values are the location of the "origin" of the compass and *armlen* is the length of each arm. (The arms will of course seem shorter because of perspective). The *x*, *y* and *armlen* values are in screen coordinates (i.e., pixels). The *camera* is used to obtain orientation information about the user's viewpoint.

The vrl_UserInterfaceDropText() routine displays the text message at the given screen coordinates in the given color, with a black (i.e., color 0) drop shadow.

Tasks

The pseudo-tasking mechanism is described in the Tutorial. Tasks are added using vrl_TaskCreate(), which takes a pointer to the function, a pointer to the data, and the period. The tasks should be run periodically by calling vrl_TaskRun(), which is normally done in vrl_SystemRun(). The tasks can obtain a pointer to their data by calling vrl_TaskGetData(), the elapsed time since they last ran by calling vrl_TaskGetElapsed(), and the current time by calling vrl_TaskGetTimeNow(). Note that for any given call to vrl_TaskRun(), all the tasks will receive the same value from vrl_TaskGetTimeNow(); this is different from vrl_TimerRead(), since the timer runs independently of the tasks. You may want to use one or the other of those two functions depending on the circumstances.

```
vrl_Boolean vrl_TaskCreate(void (*function)(void), void *data, vrl_Time period);
void vrl_TaskRun(void);
void *vrl_TaskGetData(void);
vrl_Time vrl_TaskGetElapsed(void);
vrl_Time vrl_TaskGetTimeNow(void);
```

Primitives

AVRIL currently provides five utility routines for creating simple geometric primitives. Each takes a surface map pointer; if the value is NULL, the default color for geometric primitives is used.

```
vrl_Shape *vrl_PrimitiveBox(vrl_Scalar width, vrl_Scalar height, vrl_Scalar depth,
        vrl_Surfacemap *map);

vrl_Shape *vrl_PrimitiveCone(vrl_Scalar radius, vrl_Scalar height, int nsides,
        vrl_Surfacemap *map);

vrl_Shape *vrl_PrimitiveCylinder(vrl_Scalar bottom_radius, vrl_Scalar top_radius,
        vrl_Scalar height, int nsides, vrl_Surfacemap *map);

vrl_Shape *vrl_PrimitivePrism(vrl_Scalar width, vrl_Scalar height, vrl_Scalar depth,
        vrl_Surfacemap *map);

vrl_Shape *vrl_PrimitiveSphere(vrl_Scalar radius, int vsides, int hsides,
        vrl_Surfacemap *map);
```

The box and sphere have their origin at their geometric centers, the cone and the cylinder have their origin at the center of their bases, and the prism has its origin at one corner. You can use vrl_ShapeOffset() to change these choices if you wish.

Rendering

The rendering "engine" needs to be initialized before any actual rendering is done. The renderer needs to know how much memory to allocate for itself, as well as the maximum number of objects, facets, vertices and lights it will have to contend with. When the program is ready to exit, vrl_RenderQuit() should be called to cleanly shut down the engine.

```
vrl_Boolean vrl_RenderInit(int maxvert, int maxf, int maxobjs, int maxlights,
        unsigned int mempoolsize);
void vrl_RenderQuit(void);
```

The routines in system.c normally handle the calling of vrl_RenderInit(), and the setting up of an atexit() function for vrl_RenderQuit().

Two functions are used to give the renderer a pointer to the current camera and list of lights, and to set the current ambient lighting level (usually that for the current world):

```
void vrl_RenderBegin(vrl_Camera *camera, vrl_Light *lights);
void vrl_RenderSetAmbient(vrl_Factor amb);
```

Finally, two functions do the actual drawing; one draws a horizon, the other renders a list of objects (such as that returned by vrl_ObjectUpdate() or vrl_WorldUpdate()).

```
void vrl_RenderHorizon(void);
vrl_Status *vrl_RenderObjlist(vrl_Object *objects);
```

The vrl_RenderObjlist() function returns a pointer to a status struct, which is described in the Tutorial.

Since the renderer can highlight objects and draw them in wireframe, there are functions for setting and getting the colors used for highlighting and wireframe:

```
void vrl_RenderSetWireframeColor(vrl_Color color);
vrl_Color vrl_RenderGetWireframeColor(void);
void vrl_RenderSetHighlightColor(vrl_Color color);
vrl_Color vrl_RenderGetHighlightColor(void);
```

You can also set and get the rendering mode:

```
void vrl_RenderSetDrawMode(int mode);
int vrl_RenderGetDrawMode(void);
```

At the moment, the only values supported for the *mode* are 0 (normal) and 1 (wireframe).

For stereoscopic rendering, a horizontal shift factor is required:

```
void vrl_RenderSetHorizontalShift(int npixels);
```

There are two functions that allow you to monitor a particular point on the screen, render the world, and then see what objects and facets were under the cursor (and nearest the viewer).

```
void vrl_RenderMonitorInit(int x, int y);
vrl_Boolean vrl_RenderMonitorRead(vrl_Object **obj, vrl_Facet **facet, int *vertnum);
```

The *x* and *y* values are coordinates in the current screen window (such as those passed to vrl_ApplicationMouseUp()). The *obj* pointer (if not NULL) gets set to point to the object the cursor was over; similarly, the *facet* pointer (if not NULL) gets set to point to the facet the cursor was over. The *vertnum* pointer is not currently used. If nothing was under the cursor, vrl_RenderMonitorRead() returns zero. Remember that you must call vrl_RenderObjlist() between the call to vrl_RenderMonitorInit() and vrl_RenderMonitorRead(); typically, you would call it as vrl_RenderObjlist(NULL) to just re-render the last object list that was used.

If you need to find out where on the screen a specific vertex will be drawn, you can use the following function:

```
void vrl_TransformVertexToScreen(vrl_ScreenCoord *x, vrl_ScreenCoord *y, vrl_Object *obj,
       vrl_Vector vertex);
```

The vertex is assumed to be in the object's coordinate system. The vertex is transformed, projected, scaled and shifted and the screen coordinates are stored in x and y. Note that the values are of type vrl_ScreenCoord, so you should right shift the results by VRL_SCREEN_FRACT_BITS. The vrl_ObjectToScreen() function uses information computed during the most recent call to vrl_RenderBegin(), so the values will not reflect any changes made since then.

The Timer

AVRIL has a set of routines which deal with the timer; these will vary from one platform to another, but they should all provide the same high-level interface to application software.

```
vrl_Boolean vrl_TimerInit(void);
void vrl_TimerQuit(void);
vrl_Time vrl_TimerRead(void);
vrl_Time vrl_TimerGetTickRate(void);
void vrl_TimerDelay(vrl_Time milliseconds);
```

These routines let you initialize, de-initialize and read the timer. The vrl_TimerGetTickRate() routine returns the number of ticks per second that the timer runs at. The higher this number is, the more accurate the frames/second calculations will be (among other things). It is expected that all future versions of AVRIL will use 1000 ticks per second, so that each tick is one millisecond; however, to be on the safe side, always use vrl_TimerGetTickRate().

The Mouse

Like the timer routines, the mouse routines will differ from platform to platform, but the high-level interface should remain the same.

```
vrl_Boolean vrl_MouseInit(void);
void vrl_MouseQuit(void);
vrl_Boolean vrl_MouseReset(void);
vrl_Boolean vrl_MouseRead(int *x, int *y, unsigned int *buttons);
void vrl_MouseSetUsage(int u);
int vrl_MouseGetUsage(void);
void vrl_MouseSetPointer(void *u);
void *vrl_MouseGetPointer(void);
```

These routines let you initialize and read the mouse. Since the mouse can be used either as a screen-oriented pointer or as a 6D input device (see the Devices section), there has to be some way of toggling between those two functions. That's what the vrl_MouseSetUsage() and vrl_MouseGetUsage() functions are for; a non-zero value means the mouse is a 6D device, a zero value means it's a screen pointer.

When the mouse is a 6D device, it's useful to be able to obtain a pointer to the vrl_Device which is using it; similarly, the vrl_Device function (described later) must be able to set that pointer. That's what the vrl_MouseSetPointer() and vrl_MouseGetPointer() calls do; they set and get a pointer to the vrl_Device that's using the mouse for 6D input.

The Keyboard

Like the timer and the mouse, the keyboard routines will be implemented very differently on different platforms, but they should provide a consistent high-level interface.

```
vrl_Boolean vrl_KeyboardCheck(void);
unsigned int vrl_KeyboardRead(void);
```

The vrl_KeyboardCheck() routine returns non-zero if a key has been pressed, and vrl_KeyboardRead() returns the actual key. Most keys just return their ASCII values; see the file avrilkey.h for definitions of special keys (like arrows, function keys, etc).

Memory Allocation Routines

AVRIL has three functions which are (at the moment) just wrappers around the standard malloc(), calloc() and free() functions. You should always use these functions rather than the system equivalents, for compatability with future versions of AVRIL.

```
void *vrl_malloc(unsigned int nbytes);
void *vrl_calloc(unsigned int nitems, unsigned int item_size);
void vrl_free(void *ptr);
```

Raster Routines

AVRIL uses the notion of a "raster", a rectangular array of pixel values. In most cases, all rendering is done into a raster, which may or may not correspond to an actual physical screen; in cases where it doesn't, it's necessary to copy ("blit") the raster to the display.

Each raster has a height, a width, a depth (number of bits per pixel) a window (the rectangular region within the raster into which rendering is done) and a "rowbytes" value (the number of bytes per horizontal row of pixels).

The following routines are used to deal with rasters:

```
vrl_Raster *vrl_RasterCreate(vrl_ScreenPos width, vrl_ScreenPos height,
        vrl_unsigned16bit depth);
void vrl_RasterDestroy(vrl_Raster *raster);
void vrl_RasterSetWindow(vrl_Raster *raster,
        vrl_ScreenPos left, vrl_ScreenPos top, vrl_ScreenPos right, vrl_ScreenPos bottom);
```

```
void vrl_RasterGetWindow(vrl_Raster *raster, vrl_ScreenPos *left, vrl_ScreenPos *top,
        vrl_ScreenPos *right, vrl_ScreenPos *bottom);
vrl_ScreenPos vrl_RasterGetHeight(vrl_Raster *r);
vrl_ScreenPos vrl_RasterGetWidth(vrl_Raster *r);
vrl_ScreenPos vrl_RasterGetDepth(vrl_Raster *r);
vrl_ScreenPos vrl_RasterGetRowbytes(vrl_Raster *r);
void vrl_RasterSetRowbytes(vrl_Raster *r, vrl_ScreenPos n);
```

The reason for setting the number of bytes per row of pixels has to do with alternate scan-line encoding; if you render the left and right images into the left and right halves of a 640-pixel wide screen, and then set the rowbytes value to 320, you wind up with a raster that has the left-eye image on the even scanlines and the right-eye image on the odd scanlines.

You can read scanlines from a raster into a buffer, or write them from a buffer into the raster. You can also obtain a pointer to the actual data for the raster (i.e. the raw array of pixel values).

```
void vrl_RasterReadScanline(vrl_Raster *r, int n, unsigned char *buff);
void vrl_RasterWriteScanline(vrl_Raster *r, int n, unsigned char *buff);
unsigned char *vrl_RasterGetData(vrl_Raster *r);
```

Even though you can obtain a pointer to the actual data, it’s generally a bad idea to do anything with it.

Palettes and Huemaps

AVRIL makes use of a "palette", a collection of (up to) 256 colors. Each of those colors has three 8-bit components -- one for red, one for green and one for blue. On the PC’s VGA card, only 6 bits are actually used for each component.

The vrl_Color values that AVRIL uses are (in a paletted implementation) used to index the palette to select an actual color. In order to do shading of facets and vertices, AVRIL divides the palette into a number of "hues", with a number of shades for each hue. By default, the first 16 entries of the 256-color palette are simple non-shaded colors (for use in menus and overlaid text); the remaining 240 colors are treated as 15 hues with 16 shades each.

However, this is not etched in stone. AVRIL supports the use of a "hue map", which relates a hue index to a start color in the palette and a count of the number of shades. For example, by using the hue map, you could choose to have 64 shades of flesh tone (instead of 16) in order to represent human beings more accurately.

A hue is represented by a vrl_Hue type, and a palette by a vrl_Palette type. Palettes have a flag that indicates that they’ve been changed; this is important, since it forces the system to update the physical palette stored in the video hardware.

The following functions will initialize a palette, read a palette and huemap data from a file, get and set individual entries in the palette, get a pointer to a palette’s huemap, and read and check the "changed" status of the palette:

```
void vrl_PaletteInit(vrl_Palette *pal);
int vrl_PaletteRead(FILE *in, vrl_Palette *pal);
vrl_Color vrl_PaletteGetEntry(vrl_Palette *pal, int n);
void vrl_PaletteSetEntry(vrl_Palette *pal, int n, vrl_Color color);
vrl_Boolean vrl_PaletteHasChanged(vrl_Palette *pal);
void vrl_PaletteSetChanged(vrl_Palette *pal, vrl_Boolean flag);
vrl_Hue *vrl_PaletteGetHuemap(vrl_Palette *pal);
```

Each world structure has a palette built into it; you can obtain a pointer to it using the following function:

```
vrl_Palette *vrl_WorldGetPalette(void);
```

Bear in mind that the palettes we're discussing here are independent of the actual, physical palette that's maintained by the video hardware; see the section on Video routines for more information about the hardware palette.

Video Routines

AVRIL takes care of all the "high-level" functions required of a VR library, including the handling of input devices, reading files, maintaining data structures, doing transforms and lighting calculations and so on. AVRIL makes calls to lower-level routines to do things like accessing the display device; in fact, it has two separate levels of interface to the display hardware.

The lowest-level display interface is the video driver. The video driver is responsible for such things as entering and exiting graphics mode, hiding and displaying the cursor and loading the hardware palette. Video drivers are easy to replace, and you can write your own to support whatever graphics modes you wish to use.

Internally, the video driver is just a function; see Appendix G for information about how to write one. The application-visible functions that access that driver are described here.

```
void vrl_VideoSetDriver(vrl_VideoDriverFunction driver);
```

This function sets the video driver function to use; all subsequent calls to the vrl_Video family of routines will ultimately be passed to the specified function. See Appendix G for more details.

```
int vrl_VideoGetVersion(void);
char *vrl_VideoGetDescription(void);
```

These two functions allow you to find out which version of the video driver specification the currently-selected driver supports, and to get a textual description of the driver.

```
int vrl_VideoSetup(int mode);
int vrl_VideoGetMode(void);
```

The vrl_VideoSetup() function puts the system into graphics mode; the *mode* parameter specifies the graphics submode to use (for drivers that support multiple graphics modes). The meaning of the *mode* parameter is specific to the driver being used; 0x1234 may mean completely different things to different video drivers. The function returns a non-zero value if it was unable to enter the specified graphics mode. The vrl_VideoGetMode() function returns the current graphics mode; this may or may not be the one that was requested.

```
void vrl_VideoShutdown(void);
```

This function shuts down the video subsystem and returns the hardware to whatever mode it was in prior to the last call to vrl_VideoSetup().

The video subsystem provides a vrl_Raster describing the actual physical framebuffer. To obtain a pointer to the raster (from which information such as the height, width and depth of the video adapter may be obtained), use the following function:

```
vrl_Raster *vrl_VideoGetRaster(void);
```

Many video adapters support multiple pages. AVRIL has the notion of a "current drawing page" (to which all output is written) and a "current view page" (the one that's being displayed on the physical screen). The following functions allow you to find out how many pages the adapter has, and to set and query the value of each of those pages:

```
int vrl_VideoGetNpages(void);
void vrl_VideoSetDrawPage(int page);
int vrl_VideoGetDrawPage(void);
void vrl_VideoSetViewPage(int page);
int vrl_VideoGetViewPage(void);
```

Most current video adapters have 8 bits per pixel, and a "palette" of 256 colors. You can find out whether the video adapter being used has a palette by using the function

```
vrl_Boolean vrl_VideoHasPalette(void);
```

You can read and write a range of values within the palette using the following functions:

```
void vrl_VideoSetPalette(int start, int end, vrl_Palette *palette);
void vrl_VideoGetPalette(int start, int end, vrl_Palette *palette);
```

On some systems, you actually render to an off-screen vrl_Raster in system memory and then copy (or "blit") the image onto the screen. This is done by the function

```
void vrl_VideoBlit(vrl_Raster *raster)
```

You may only want to update the physical display during the vertical blanking interval; to monitor the vertical retrace, use the following function:

```
vrl_Boolean vrl_VideoCheckRetrace(void);
```

The video subsystem is responsible for the on-screen cursor. The following functions allow you to hide the cursor, show it again, reset it to its initial state, move it and set its appearance:

```
void vrl_VideoCursorHide(void);
void vrl_VideoCursorShow(void);
void vrl_VideoCursorReset(void);
void vrl_VideoCursorMove(vrl_ScreenPos x, vrl_ScreenPos y);
void vrl_VideoCursorSetAppearance(void *app);
```

Whenever you write to the currently visible page (the viewpage) you should first call vrl_VideoCursorHide() to prevent the mouse from being "squashed" under a falling polygon. When you're finished updating the display, call vrl_VideoCursorShow() to let the rodent run free again. The vrl_SystemRender() routine, found in system.c, shows how these routines are used. The user interface routines do the calls to vrl_VideoCursorHide() and vrl_VideoCursorShow(), so you don't need to worry about them when you use those functions.

Note that the system keeps a count of the number of times you've called vrl_VideoCursorHide() and vrl_VideoCursorShow(); for example, if you hide the mouse cursor twice, you have to show it twice before it actually appears. The vrl_VideoCursorReset() routine resets the "hidden" count.

The vrl_VideoCursorMove() routine moves the cursor to a particular spot on the screen, and the vrl_VideoCursorSetAppearance() routine sets a new visual appearance for the cursor (the data that is pointed to by the *app* parameter varies from driver to driver).

Display Routines

The display subsystem is the "back end" of the rendering pipeline; it's responsible for actually drawing polygons (and lines, and text) into a raster.

Internally, the display driver is just a function (much like the video driver, or any of the input device drivers); see Appendix H for information about how to write one. The application-visible functions that access that driver are described here.

```
void vrl_DisplaySetDriver(vrl_DisplayDriverFunction driver);
```

This function sets the display driver function to use; all subsequent calls to the vrl_Display family of routines will ultimately be passed to the specified function.

```
int vrl_DisplayInit(vrl_Raster *raster);
void vrl_DisplayQuit(void);
```

These two routines are responsible for initializing and de-initializing the display subsystem. The *raster* parameter specifies a raster to use; this may be an off-screen buffer in system memory, or the actual physical framebuffer returned by vrl_VideoGetRaster().

You can change the raster being used; to set or query the raster, use the following functions:

```
void vrl_DisplaySetRaster(vrl_Raster *r);
vrl_Raster *vrl_DisplayGetRaster(void);
```

You can directly obtain the height, width and depth of the current display raster using the following routines:

```
int vrl_DisplayGetWidth(void);
int vrl_DisplayGetHeight(void);
int vrl_DisplayGetDepth(void);
```

You can obtain the current version of the display driver, and a string describing the driver, using the following functions:

```
int vrl_DisplayGetVersion(void);
char *vrl_DisplayGetDescription(void);
```

The display subsystem contains routines for clearing the screen, drawing individual points (i.e. pixels), drawing lines, drawing boxes, and drawing text:

```
void vrl_DisplayClear(vrl_Color color);
void vrl_DisplayPoint(vrl_ScreenPos x, vrl_ScreenPos y, vrl_Color color);
void vrl_DisplayLine(vrl_ScreenPos x1, vrl_ScreenPos y1, vrl_ScreenPos x2,
        vrl_ScreenPos y2, vrl_Color color);
void vrl_DisplayBox(vrl_ScreenPos x1, vrl_ScreenPos y1, vrl_ScreenPos x2,
        vrl_ScreenPos y2, vrl_Color color);
void vrl_DisplayText(vrl_ScreenPos x, vrl_ScreenPos y, vrl_Color color,
         char *message);
```

When drawing a string of text, it's sometimes necessary to find out how wide and high it will appear on screen; the following two functions provide that information:

```
vrl_ScreenPos vrl_DisplayGetTextWidth(char *string);
vrl_ScreenPos vrl_DisplayGetTextHeight(char *string);
```

The rendering engine (or the application) may need to find out what capabilities the display driver has; in particular, whether it can do things like Gouraud shading and X-Y clipping. The following routines provide the answers:

```
vrl_Boolean vrl_DisplayCanGouraud(void);
vrl_Boolean vrl_DisplayCanXYclip(void);
```

When rendering, certain tasks have to be performed at the beginning and end of every frame; the application informs the display driver when a frame begins and ends using the following two functions:

```
void vrl_DisplayBeginFrame(void);
void vrl_DisplayEndFrame(void);
```

Some display drivers are capable of Z-buffering, either in hardware or in software. The following routines are provided to support Z-buffers:

```
vrl_DisplayUseZbuffer(flag)
void vrl_DisplaySetZbuffer(vrl_Raster *r);
vrl_Raster *vrl_DisplayGetZbuffer(void);
void vrl_DisplayClearZbuffer(depth);
```

These functions allow you to set and get pointers to the vrl_Raster that will be used as a Z-buffer (if software Z-buffering is used). The vrl_DisplayClearZbuffer() routine clears the currently-set Z-buffer (hardware or software) to the specified depth value. The vrl_DisplayUseZbuffer() routine tells the display driver whether or not to use the Z-buffer; the flag is non-zero if Z-buffering should be enabled, and the return value is 0 if no Z-buffer is present, 1 if a software Z-buffer is available, or 2 if a hardware Z-buffer is available. Note that as of version 2.0, AVRIL does not yet support Z-buffering.

Some shading algorithms have greater computational expense than others do. It's possible to limit the complexity of the shading that the display driver uses by calling the following function:

```
void vrl_DisplaySetShading(int value);
```

The higher the *value* parameter, the more time is spent on shading (e.g. Gouraud versus flat, dithering enabled or disabled, etc). The exact meaning is up to the display driver.

Updating the actual, physical display from the off-screen buffer is done by calling the routine

```
void vrl_DisplayUpdate(void);
```

For drivers that write straight to the physical framebuffer, the vrl_DisplayUpdate() function does nothing; for those that use off-screen buffers, it does the blit.

Some display drivers need to perform certain re-calculations whenever the palette changes; to inform the display driver of a new palette, call the following routine:

```
void vrl_DisplayUpdatePalette(vrl_Palette *palette);
```

You can specify a rectangular on-screen window, to limit the area of the screen that the display driver will write to. You can also find out the location and size of that window. The functions to do so are as follows:

```
void vrl_DisplaySetWindow(vrl_ScreenPos x1, vrl_ScreenPos y1, vrl_ScreenPos x2,
        vrl_ScreenPos y2);
void vrl_DisplayGetWindow(vrl_ScreenPos *x1, vrl_ScreenPos *y1,
        vrl_ScreenPos *x2, vrl_ScreenPos *y2);
```

Note that stereoscopic rendering may have an impact on the "real" windows being used.

The display subsystem also has support for stereoscopic viewing. It understands certain types of viewing, and must be informed when the stereo type changes. You can set and query the stereo type using the following two functions:

```
void vrl_DisplayStereoSetType(VRL_STEREO_TYPE stype);
VRL_STEREO_TYPE vrl_DisplayStereoGetType(void);
```

You can specify which "eye" is being drawn to, and which should be displayed, using the following functions:

```
void vrl_DisplayStereoSetDrawEye(VRL_STEREO_EYE eye);
VRL_STEREO_EYE vrl_DisplayStereoGetDrawEye(void);
void vrl_DisplayStereoSetViewEye(VRL_STEREO_EYE eye);
VRL_STEREO_EYE vrl_DisplayStereoGetViewEye(void);
```

The *eye* parameter is one of the values VRL_STEREOEYE_LEFT or VRL_STEREOEYE_RIGHT.

Some stereoscopic viewing systems subdivide the screen, putting the left-eye image in one area and the right-eye image in another; the two are then optically "shuffled" to the appropriate eye. The specific screen areas to use for each eye are set using the following functions:

```
void vrl_DisplayStereoSetLeftWindow(vrl_ScreenPos x1, vrl_ScreenPos y1, vrl_ScreenPos x2,
        vrl_ScreenPos y2);
void vrl_DisplayStereoSetRightWindow(vrl_ScreenPos x1, vrl_ScreenPos y1, vrl_ScreenPos x2,
        vrl_ScreenPos y2);
```

And finally, some stereoscopic viewing systems use more than one display card. The following function is used to specify a "callback" or "upcall" function that should be called to select a specific card:

```
void vrl_DisplayStereoSetCardFunction(void (*function)(VRL_STEREO_EYE));
```

The callback function is told which card should be written to; the value is one of the constants VRL_STEREOEYE_NEITHER, VRL_STEREOEYE_LEFT, VRL_STEREOEYE_RIGHT or VRL_STEREOEYE_BOTH. The function will be called by the video subsystem in order to select either, neither or both of the video cards.

PCX File Routines

There are two functions that deal with files in PCX format:

```
vrl_Boolean vrl_ReadPCX(FILE *in);
vrl_Boolean vrl_WritePCX(FILE *out);
```

The first one reads a PCX file into the current drawing page, the other writes the current drawing page out to disk as a PCX file.

Devices

AVRIL has support for input devices providing multiple "degrees of freedom" (DOF). In fact, AVRIL's devices are even more general; each device can have an arbitrary number of input and output channels.

To use a device in AVRIL, you must first "open" it; this is analogous to opening a file. When you're finished with the device, you should "close" it; you can close all open devices using vrl_DeviceCloseAll(), which is set as an atexit() function by vrl_SystemStartup().

```
vrl_Device *vrl_DeviceOpen(vrl_DeviceDriverFunction fn, vrl_SerialPort *port);
void vrl_DeviceClose(vrl_Device *device);
void vrl_DeviceCloseAll(void);
```

Most input devices communicate over a serial port; the *port* parameter is a pointer to such a port that's been opened with vrl_SerialOpen(). See the section on Serial Ports for more details. Devices (such as the keyboard) that don't use a serial port should pass NULL as the *port* parameter. The *fn* parameter is a function that operates the device; there are a number of these already defined for popular devices, and it's easy to write your own if you have unusual devices you wish to support. They are listed in avrildrv.h, and in the cfg.c file; you should update those files as you add drivers.

If you need to get a pointer to the serial port associated with a device, just call the following function:

```
vrl_SerialPort *vrl_DeviceGetPort(vrl_Device *device);
```

Devices can have a "mode" associated with them, whose meaning is specific to each device. You can set and get a device's current mode using these two functions:

```
void vrl_DeviceSetMode(vrl_Device *device, int mode);
int vrl_DeviceGetMode(vrl_Device *device);
```

Devices can also have "nicknames"; for example, your application might deal with a head-tracker called "headtrack" which would be defined (in the configuration file, most likely) to be the device used for head tracking (e.g. a Polhemus Isotrak or a Logitech Red Baron). You can set and query the nickname of a device, or find a device with a particular nickname, using the following functions:

```
char *vrl_DeviceGetNickname(vrl_Device *device);
void vrl_DeviceSetNickname(vrl_Device *device, char *nickname);
vrl_Device *vrl_DeviceFind(char *nickname);
```

Once a device has been opened, you can easily find out how many input channels it has, and how many two-state buttons are on the device, using the following two functions:

```
int vrl_DeviceGetNchannels(vrl_Device *device);
int vrl_DeviceGetNButtons(vrl_Device *device);
```

Sometimes devices can get into a strange state, or drift from their initial settings; the function vrl_DeviceReset() resets a device to the state it was in just after it was opened.

```
int vrl_DeviceReset(vrl_Device *device);
```

For many devices, resetting a device also marks the current values of all its channels as the "zero" value for that channel. To mark the current values as being the "maximum" values, call the function

```
void vrl_DeviceSetRange(vrl_Device *device);
```

Devices should be periodically "polled" to see if they have anything to report; this is normally done in the vrl_SystemRun() function. An individual device can be polled using vrl_DevicePoll(), which returns a non-zero value if new data was acquired. All the devices can be polled by calling vrl_DevicePollAll(), which returns a non-zero value if any of the devices had new data.

```
int vrl_DevicePoll(vrl_Device *device);
void vrl_DevicePollAll(void);
```

You can get the current value of a channel's input by calling vrl_DeviceGetValue(), and you can read the button status on the device using vrl_DeviceGetButtons():

```
vrl_Scalar vrl_DeviceGetValue(vrl_Device *device, int channel);
vrl_unsigned32bit vrl_DeviceGetButtons(vrl_Device *device);
```

Each bit corresponds to a single button on the device.

The first six channels are special, since they correspond to the six basic degrees of freedom a device can have. The first three, whose channel numbers are the #defined values X, Y, and Z, provide the three-dimensional location of the input device in its private coordinate system. The next three, whose channel numbers are the #defined values XROT, YROT and ZROT, provide the rotation of the device around each of the three axes. Every device is expected to provide at least those six values; others, such as glove-like input devices, may have a separate channel for the flexion of each finger.

You can determine whether a channel's value has changed since the previous poll by using vrl_DeviceChannelGetChanged(), and you can use vrl_DeviceGetChangedButtons() to obtain a vrl_unsigned32bit word whose bits indicate whether the corresponding buttons have changed state since the previous poll.

```
vrl_Boolean vrl_DeviceGetChanged(vrl_Device *device, int channel);
vrl_unsigned32bit vrl_DeviceGetChangedButtons(vrl_Device *device);
```

Note that "previous poll" means the one prior to the most recent one; in other words, you would check the changed flags immediately after a call to vrl_DevicePoll() or vrl_DevicePollAll() in order to see if they've changed since the last time through.

Some devices (such as the Logitech Cyberman and the Global Devices Controller) are capable of output as well as input. You can find out the number of output channels a device has using the following function:

```
int vrl_DeviceGetNOutputChannels(vrl_Device *device);
```

There are some devices that shouldn't be polled too frequently (possibly because the polling takes a long time). Devices drivers typically set their own polling frequency when they're initialized, but you can read and set the polling period using these two functions:

```
void vrl_DeviceSetPeriod(vrl_Device *device, vrl_Time period);
vrl_Time vrl_DeviceGetPeriod(vrl_Device *device);
```

Some devices provide fewer than six degrees of freedom; in particular, some devices (such as sourceless head trackers) provide only rotational information, while others might provide only positional information. In addition, some axes are absolute, while others are relative; for example, a magnetic tracker provides absolute rotation, whereas a joystick usually provides a *rate* of rotation. Two functions are used to determine what a device's suggested modes of operation are for translation and rotation:

```
vrl_DeviceMotionMode vrl_DeviceGetRotationMode(vrl_Device *device);
vrl_DeviceMotionMode vrl_DeviceGetTranslationMode(vrl_Device *device);
```

Each of these two functions returns one of the values VRL_MOTION_NONE (i.e., this type of motion is not reported by this device), VRL_MOTION_RELATIVE (i.e. this device provides relative information) or VRL_MOTION_ABSOLUTE (this device provides absolute information).

It's important to note that these are all *suggestions* from the device driver as to how it should be used; you can still choose, in your application, to treat any device as either absolute or relative.

To understand what these next few functions do, it's important to understand what kind of processing the system does on the values once it receives them from the device. Each channel can be in either "accumulate" or "non-accumulate" mode. For accumulating devices, the value is first checked to see how close it is to zero; if it's less than a channel-specific *deadzone* value, then the value is considered to be zero. For non-accumulating devices, the deadzone value is treated as a minimum change from the most recently read value for this channel; a device that moves by less than the deadzone amount between consecutive polls will not change in value.

The value is then scaled so that its maximum value is less than a channel-specific scale value. For channels with the *accumulate* flag set, the value is also scaled by the elapsed time; the *scale* for such channels is treated as the maximum rate of change per second.

The *accumulate*, *scale* and *deadzone* values can be set and read using the following calls:

```
vrl_Scalar vrl_DeviceGetDeadzone(vrl_Device *device, int channel);
void vrl_DeviceSetDeadzone(vrl_Device *device, int channel, vrl_Scalar value);
vrl_Scalar vrl_DeviceGetScale(vrl_Device *device, int channel);
vrl_DeviceSetScale(vrl_Device *device, int channel, vrl_Scalar value);
vrl_Boolean vrl_DeviceGetAccumulate(vrl_Device *device, int channel);
```

```
void vrl_DeviceSetAccumulate(vrl_Device *device, int channel, vrl_Boolean value);
```

If you like, you can bypass all that processing and obtain the actual, "raw" value being reported by the device by calling

```
vrl_Scalar vrl_DeviceGetRawValue(vrl_Device *device, int channel);
```

Devices are kept internally in a linked list; if you want to iterate over the list, you can do it with the following two functions:

```
vrl_Device *vrl_DeviceGetFirst(void);
vrl_Device * vrl_DeviceGetNext(vrl_Device *device);
```

You can retrieve a human-readable description of a device by calling the function

```
char *vrl_DeviceGetDesc(vrl_Device *device);
```

You can produce output on any of the device's channels by calling

```
void vrl_DeviceOutput(vrl_Device *device, int channel, vrl_Scalar value);
```

The *channel* and the *value* (which ought to be in the range 0 to 255) are passed along to the device; if it's capable of outputting that value on that channel, it does. Not all devices are capable of producing output, and those that can have a variety of different ways of doing it (including sound and vibration). All you can be sure of is that a value of zero will turn the output off, and a non-zero value will turn it on.

Note that it's possible to have an "output-only" device; it will report zero for all its input values, but still respond to output requests. This might be one approach to supporting motion platforms, for example.

Some 2D devices can map their two axes into 6 degrees of freedom using combinations of buttons. There are two functions to get and set the mapping tables they use:

```
void vrl_DeviceSetButtonmap(vrl_Device *device, vrl_DeviceButtonmap *b);
vrl_DeviceButtonmap *vrl_DeviceGetButtonmap(vrl_Device *device);
```

Buttonmaps are a fairly complex topic, and are discussed in more detail in Appendix F.

Serial Ports

Since many input devices use serial communications, AVRIL contains a library of serial port routines. These will mostly be of interest to people writing device drivers, but they might also be used for modem communications.

To some extent, serial port support will be platform-dependent; the meaning of the various parameters in the vrl_SerialOpen() call will be different on different systems. However, all the other routines should be the same regardless of platform.

A serial port can be opened and closed using vrl_SerialOpen() and vrl_SerialClose() respectively; all the serial ports can be closed at once using vrl_SerialCloseAll(), which gets set as an atexit() function by vrl_SystemStartup(). The communications parameters can be set using vrl_SerialSetParameters().

```
vrl_SerialPort *vrl_SerialOpen(unsigned int address, int irq, unsigned int buffsize);
void vrl_SerialClose(vrl_SerialPort *port);
void vrl_SerialCloseAll(void);
void vrl_SerialSetParameters(vrl_SerialPort *port, unsigned int baud,
        vrl_ParityType parity, int databits, int stopbits);
```

The *address* parameter to vrl_SerialOpen() is interpreted differently on different platforms; on PC-compatible machines, it's the base address of the UART chip (usually 0x3F8 for COM1, 0x2F8 for COM2). The *irq* parameter is also system-dependent; on PC-compatible machines, it's the hardware interrupt level the serial port uses (usually 4 for COM1, 3 for COM2).

The *buffsize* parameter is the size of buffer to use for incoming data. If it's set to zero, the port will be in a non-buffered mode; this may mean that characters get lost. Such a mode would only be used if you're doing your own handshake with the device; in other words, you send it a byte to poll it, and then sit in a tight loop receiving the resulting data. In this case, the *irq* value is ignored.

The *baud* parameter to the vrl_SerialSetParameters() function is the baud rate; this is usually 9600 for most input devices. The *parity* parameter is one of VRL_PARITY_NONE, VRL_PARITY_EVEN or VRL_PARITY_ODD; most devices use VRL_PARITY_NONE. The *databits* field is the number of data bits per transmitted byte; this is either 7 or 8, and most devices use 8. The number of stop bits can be 1 or 2, and is usually 1. Newly-opened serial ports are set up to be 9600 baud, VRL_PARITY_NONE, 8 data bits and 1 stop bit.

The vrl_SerialCheck() routine will return a non-zero value if there are unread characters in the input buffer (or if there's a character waiting at the UART, if the port is in unbuffered mode). The vrl_SerialGetc() routine reads and returns a byte. It should not be called unless you know there's a character waiting; in buffered mode, such a call will return zero, while in unbuffered mode the call will block until a character arrives! The vrl_SerialFlush() routine will get rid of any characters waiting in the input buffer.

```
vrl_Boolean vrl_SerialCheck(vrl_SerialPort *port);
unsigned int vrl_SerialGetc(vrl_SerialPort *port);
void vrl_SerialFlush(vrl_SerialPort *p);
```

The vrl_SerialPutc() and vrl_SerialPutString() routines put out single characters and null-terminated strings of characters respectively. The terminating null byte is not sent by vrl_SerialPutString().

```
void vrl_SerialPutc(unsigned int c, vrl_SerialPort *port);
void vrl_SerialPutString(unsigned char *s, vrl_SerialPort *p);
```

There are also two routines for controlling the state of the DTR and RTS lines:

```
void vrl_SerialSetDTR(vrl_SerialPort *port, vrl_Boolean value);
void vrl_SerialSetRTS(vrl_SerialPort *port, vrl_Boolean value);
```

and one for setting the size of the serial FIFO, if applicable:

```
void vrl_SerialFifo(vrl_SerialPort *p, int n);
```

Packet Routines

Many serial devices communicate by sending "packets" of data. There are four routines in AVRIL to support the reception of packets:

```
vrl_DevicePacketBuffer *vrl_DeviceCreatePacketBuffer(int buffsize);
void vrl_DeviceDestroyPacketBuffer(vrl_DevicePacketBuffer *buff);
vrl_Boolean vrl_DeviceGetPacket(vrl_SerialPort *port, vrl_DevicePacketBuffer *buff);
unsigned char *vrl_DevicePacketGetBuffer(vrl_DevicePacketBuffer *buff);
```

The vrl_DeviceCreatePacketBuffer() function creates a packet buffer with room for the specified number of bytes; vrl_DeviceDestroyPacketBuffer() destroys such a buffer. The vrl_DevicePacketGetBuffer() routine returns a pointer to the actual data packet.

The vrl_DeviceGetPacket() routine is designed to handle a particular type of packet, a fixed-size one in which the leading byte has the top bit set and none of the other bytes do. This format is used by the Logitech Cyberman, among other devices. You can use this routine directly, or you can write your own (with a different name, of course); the source code is found in packet.c, and the vrl_DevicePacketBuffer data structure is in avril.h (and it's not expected to change, unlike many other internal data structures). The vrl_DeviceGetPacket() routine returns a non-zero value if a complete packet has been received (i.e. exactly *buffsize* bytes have been received, starting with a byte that has the top bit set).

More information

Remember that the appendices to this document are in a separate file, as is the Tutorial.