An Introduction to the *vlib* Volume Graphics API

December 10, 2002

1 Introduction

The *vlib* volume graphics API is the result of a research project undertaken at the University of Wales (Swansea, UK). This introductory document covers the basics of the API and is intended reading for those wishing to create images using the principles of volume graphics. It is assumed that the reader has a knowledge of computer graphics, and preferably some knowledge of the field of volume graphics also.

A complete implementation of the interface can be downloaded from the official *vlib* Web site at

`http://vg.swan.ac.uk/vlib`

This is in addition to a number of example programs which use the API, the specification, sample images and animation. The full code for all the images presented in this tutorial is available on both the official *vlib* distribution CD and the Web site.

2 Configuring and Using the Library

The UWS (University of Wales Swansea) implementation of *vlib* acts a programmatical interface between your modeling programs and a lower level volume graphics and voxelization system. After downloading the software, decompress it, change to the directory where you decompressed it to and type

```
./configure
```

Then, log in as "root" and type

```
make install
```

If you cannot log in as root (and cannot find anybody who can), then type

```
make DESTDIR=/yourhomedir/vlib install
```

where `/yourhomedir` represents your home directory (note that you cannot use ~ to represent your home directory—instead, you must type the path in full). This process will create the files:

```
~/vlib/usr/local/lib/libvlib.a
~/vlib/usr/local/lib/libvlaux.a
~/vlib/usr/local/include/vlib.h
~/vlib/usr/local/include/vlaux.h
```

Now both *vlib* and a library of helper functions, *vlaux*, are installed.

You will need to include both headers (`vlaux.h` and `vlib.h`) in your programs and link both libraries (`libvlib.a` and `libvlaux.a`) into your object code in order to produce an executable file. If the libraries were installed by the root user then the necessary files are now a part of your system. Simply include the header files with

```
#include <vlib.h>
#include <vlaux.h>
```

at the top of your programs and compile the executables using (for instance),

```
gcc -o prog prog.c -lvlaux -lvlib -lm
```

If the files were installed in your home directory then you will have to inform the compiler of their location. Do this with (for instance),

```
gcc -o prog prog.c
    -I/yourhomedir/vlib/usr/local/include
    -L/yourhomedir/vlib/usr/local/lib
    -lvlaux -lvlib -lm
```

Note: If you're running on SunOS then you might have to compile with `-lnsl` as well.

This should be typed on one line and `yourhomedir` should be replaced as described above. If *vlib* detected the PVM library while it was building then you will also have to append `-lpvm3` to the above statement. Note the use of `-lm` to specify that the program should also be linked against the math library. All *vlib* programs require this under Linux. Additionally, under Solaris, `-ldl -lsocket -lnsl` is also required.

3 Rendering the First Image

In this section, we shall tackle the basics and discover how to render a simple image using the *vlib* API.

In order to produce a picture, we must first construct a scene containing objects and light sources. This process is referred to as *modeling*. In some instances, it may also be necessary to supply *vlib* with some information on how the graphics system should render the environment. A piece of such information is referred to as a *rendering parameter*.

To initialize the *vlib* system, enter the following into your C program:

```
vlInit();

vlEnd();
```

In all circumstances, these respectively indicate the first and last *vlib* commands that must be called. If you try compiling and executing your program as it is now, you won't notice much happening in the way of rendering. This is because no scene has yet been defined.

Extend your program as follows:

```
vlInit();
    vlScene();

    vlEnd();
vlEnd();
```

Now we have a scene. Unfortunately, this program will not yet run. The scene is what we could describe as "invalid"—not because it contains no objects or light sources but because the graphics system doesn't know what to do with it. For the computer to produce an image, two essential ingredients must be provided: a description of the output and a virtual camera.

Inside the scene block (that is, between the calls to **vlScene** and **vlEnd**), type the following:

```
vlImage("output.ppm", 4.0/3.0, 400, 300);
```

When the system eventually produces its output, this line will ensure that the image (called `output.ppm`) will be 400 pixels across by 300 pixels high. The second parameter represents the aspect ratio of the image and is usually set to $width/height$.

Let us now define the camera. As this is the introductory section, we shall make use of the auxiliary library to make the camera definition that little bit easier. We will see how more complex camera models are defined later on. Enter the following inside the scene block:

```
vluCamera(-100.0, 150.0, -200.0,
    0.0, 0.0, 0.0);
```

This will create a camera which is located at world coordinates $(-100, 150, -200)$ and looking towards the world origin $(0, 0, 0)$. In other words, we are 100 units to the left, 150 units above and 200 units behind the origin looking towards it.

Don't worry about the order in which you call **vlImage** and **vluCamera**. Providing you have called them both, the system will not have any problems. In fact, unless otherwise stated, the order in which you issue *vlib* commands is irrelevant.

Now try running the program. Although it now produces output, the image is somewhat less than interesting. In fact, it's completely black. This is because we need both an object *and* a light source if we wish to make anything visible.

An ambient light source would be a good place to start. Enter the following into your program:

```
vlAmbient(1.0, 1.0, 1.0, 1.0);
```

The first three parameters respectively define the red, green and blue color components of the light source. The fourth parameter is an intensity value. So by entering the above into our C program, we are requesting a bright white ambient light.

Inside the scene block, enter the following two commands:

```
vlObject();

vlEnd();
```

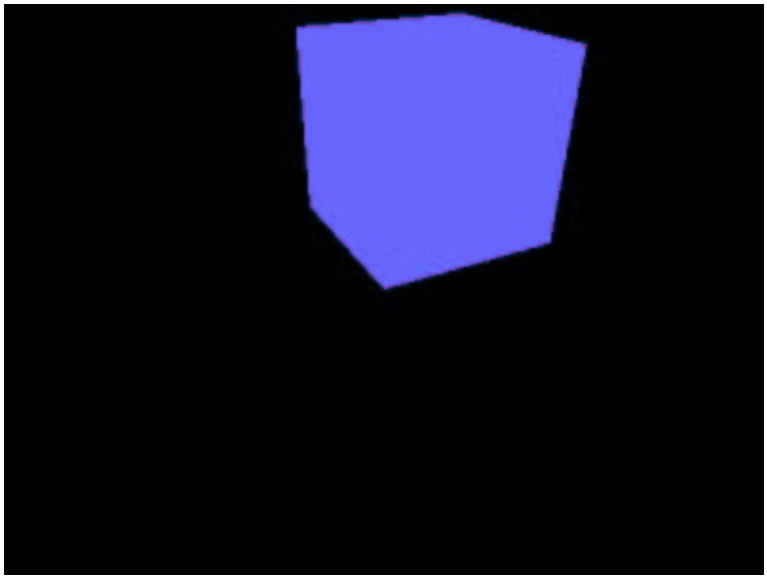

Figure 1: A plain, flat-looking, poorly-centered, blue cube.

We have just created a very simple object. It is quite small relative to our camera position. In fact, it is bounded by a unit cube with vertices at $(0, 0, 0)$ and $(1, 1, 1)$. Let us scale it up by calling

```
vlScale(100.0, 100.0, 100.0);
```

So now our object specification looks like this:

```
vlObject();
    vlScale(100.0, 100.0, 100.0);
vlEnd();
```

Although the object is large enough to be visible, and we have a light source present in the environment, the system would still produce a black image if we were to execute the program now. This is because neither reflection properties, nor any opacity, have been assigned to the object.

Try entering the following into the object definition:

```
vlSet(VL_O, 1.0);
vlSet(VL_KA | VL_B, 1.0);
vlSet(VL_R | VL_G, 0.4);
```

The first of these commands assigns the maximum opacity to the object. The next command assigns the maximum value to the ambient reflection coefficient and the blue color component simultaneously. Likewise, the third command assigns 0.4 (just below half intensity) to the red and green color components.

Try compiling and executing the program now (this was discussed in Section 2). If all has gone according to plan then you should now be rewarded with your first image (Figure 1).

4 Achieving a 3D Appearance

The image looks somewhat flat and rather boring. The pixels are uniform in color and no sense of depth is presented. This section builds on the information presented so far and describes how to make an object look three dimensional.

A scene should contain a non-ambient light source if the objects it contains are to look 3D. Let us insert a point light source:

```
vlPointLight(-100.0, 150.0, -200.0,
    VL_NOATTENUATION,
    1.0, 1.0, 1.0, 1.0);
```

This line must be entered in the scene block but outside the object block. We now have a bright white point light source located at $(-100, 150, -200)$, which is currently at the same position as the camera. Don't worry about the `VL_NOATTENUATION` parameter—it is used to indicate no light attenuation.

We are not quite done. For an object to achieve a 3D appearance, it must have suitable normals. Try entering the following lines into the object block:

```
vlSet(VL_F, 1.0);
vlEstimateNormal();
```

The symbol `VL_F` represents the so-called geometry field of an object. This field does not, strictly speaking, represent a physical property. Instead it provides the system with information on how to estimate normal vectors. We will describe the process of normal estimation a little later on. The **vlEstimateNormal** command instructs *vlib* to actually compute normal vectors; by default *vlib* will not compute them. This command should always follow any commands to set up the `VL_F` field.

Lastly, our object should reflect a little bit of diffuse light cast by the point light source. Try setting the diffuse reflection coefficient to 0.8 by entering the following into the object block:

```
vlSet(VL_KD, 0.8);
```

Also, reduce the amount of ambient light reflected by the object by setting `VL_KA` to 0.5. You will require separate **vlSet** statements to set the blue color and ambient reflection properties.

Just before we render the image, let us make a couple of modifications in order to gain a better view of the object. Try moving the view reference point (the location of the camera) to $(0, 150, -300)$ and setting it so that we look 20 units below the origin. This is done by modifying the arguments of **vluCamera** to

```
vluCamera(0.0, 150.0, -300.0,
    0.0, -20.0, 0.0);
```

Also, let's move the object a little bit. Consider the object transformations that we would like to perform. Firstly, let us translate the object so that it is centered at the origin. As we mentioned earlier, the initial boundary of a volume object has unit dimensions and vertices at $(0, 0, 0)$ and $(1, 1, 1)$. To center the object at the origin, translate it back by 0.5 in all three dimensions by calling:

```
vlTranslate(-0.5, -0.5, -0.5);
```

Then, scale it up with:

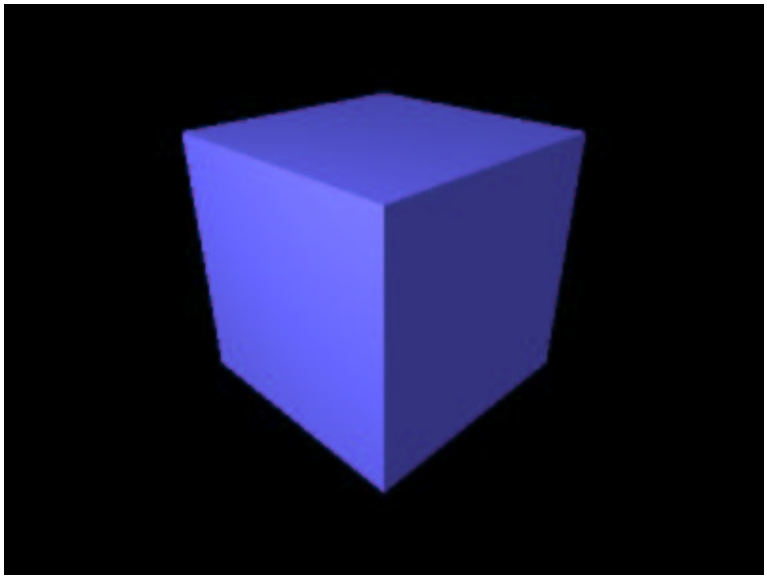

Figure 2: A three-dimensional cube.

```
vlScale(150.0, 150.0, 150.0);
```

It is important to issue these instructions in this order as matrix operations are generally non-commutative. Now the volume object has a bounding box which extends 150 units in all dimensions and has vertices at $(-75, -75, -75)$ and $(75, 75, 75)$. Lastly, rotate the object by 45 degrees about the y-axis:

```
vlRotate(0.0, 45.0, 0.0);
```

Now if you compile and execute the program you will be presented with a solid, three-dimensional object (Figure 2).

5 Applying a Solid Texture

I'm sure you would agree that the second image is considerably better than the first. But it still lacks something that's considered to be very important to many volume-defined objects: a solid texture. The RGB color components of the cube are presently set to $(0.4, 0.4, 1.0)$, which equates to a fairly light blue. We can improve on this hugely by making use of the **vlFunction** command and some of the textures included with the auxiliary library, *vlaux*.

Try to remove the commands in your program that set the object's color. If done correctly, you should now have something like this:

```
vlObject();
    vlTranslate(-0.5, -0.5, -0.5);
    vlScale(150.0, 150.0, 150.0);
    vlRotate(0.0, 45.0, 0.0);

    vlSet(VL_O, 1.0);
    vlSet(VL_F, 1.0);
    vlEstimateNormal();
    vlSet(VL_KA, 0.5);
    vlSet(VL_KD, 0.8);
vlEnd();
```

Applying a solid texture is extremely easy. Just add the following line of code to the object definition:

```
vlFunction(VL_R | VL_G | VL_B, &VLU_CHESS, 0);
```

And that's it. You have just applied a 3D chess-board texture to the cube. Don't worry about the last parameter. It can be used to pass additional information to the field function but we don't describe this functionality here.

Unlike traditional texture mapping, this texture is solid and extends right the way through the object. Suppose, for instance, that you were to cut the cube open. You would see texture on the inside and no seams anywhere. We will take a closer look at the **vlFunction** command later on.

Let us also change the background color of the environment. Now that our object is black and white, we shall use a very light blue color to establish a contrast. Enter the following line into the scene block (that is, not inside the object block):

```
vlBackground(0.8, 0.8, 1.0);
```

Finally, it is worth noting that often aliasing (the jaggy effect) is much more apparent when using certain types of solid textures. We can reduce this problem by instructing the system to super-sample. Enter the following command into the scene block:

```
vlSuperSample(3);
```

This tells the system to fire nine (3*3) rays per pixel and to average the color derived by each. If you render the image now then it should appear as Figure 3.

6 Improving Camera Control

If we take a very critical view of Figure 3 then we may say that the object doesn't fill enough of the image. In other words, we can see too much background and not enough cube. There are three ways to alleviate this problem. We could

- enlarge the object;
- move the object closer to the camera; or
- define a better camera model.

The first two options are effectively equivalent and aren't good solutions. If an object is placed too close to the camera then it may appear to deform, and this is not always a good thing. The best

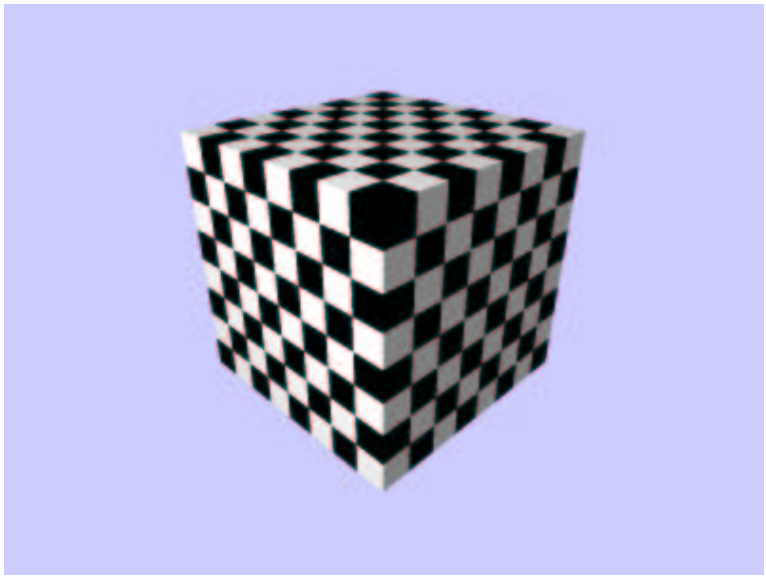

Figure 3: A solid checker texture.

solution is to reduce the size of the viewport and requires a little more work. A variable must be declared of the `vlcamera` type. For the sake of this example, let's call it `camera`.

At any point in your program, you may fill the camera structure with the relevant information. In order to keep the view reference point and *lookat* vector the same as we've been using until now, you should set the `vrp` and `lookat` components of the `camera` structure as follows:

```
camera.vrp[0] = 0.0;
camera.vrp[1] = 150.;
camera.vrp[2] = -300.0;

camera.lookat[0] = 0.0;
camera.lookat[1] = -20.0;
camera.lookat[2] = 0.0;
```

A view up vector must also be supplied. Unless you want to simulate camera tilt (e.g., rolling the head onto the shoulders) the `up` component should be set using

```
camera.up[0] = 0.0;
camera.up[1] = 1.0;
camera.up[2] = 0.0;
```

The camera definition is finished by setting the viewport dimensions, the distance from the viewport to the lens, and the camera type. Let's keep a unit distance and a perspective projection but reduce the viewport from 1.0 to 0.8 in both dimensions.

```
camera.distance = 1.0;
camera.type = VL_PERSPECTIVE;

camera.viewport[0] = 0.8;
camera.viewport[1] = 0.8;
```

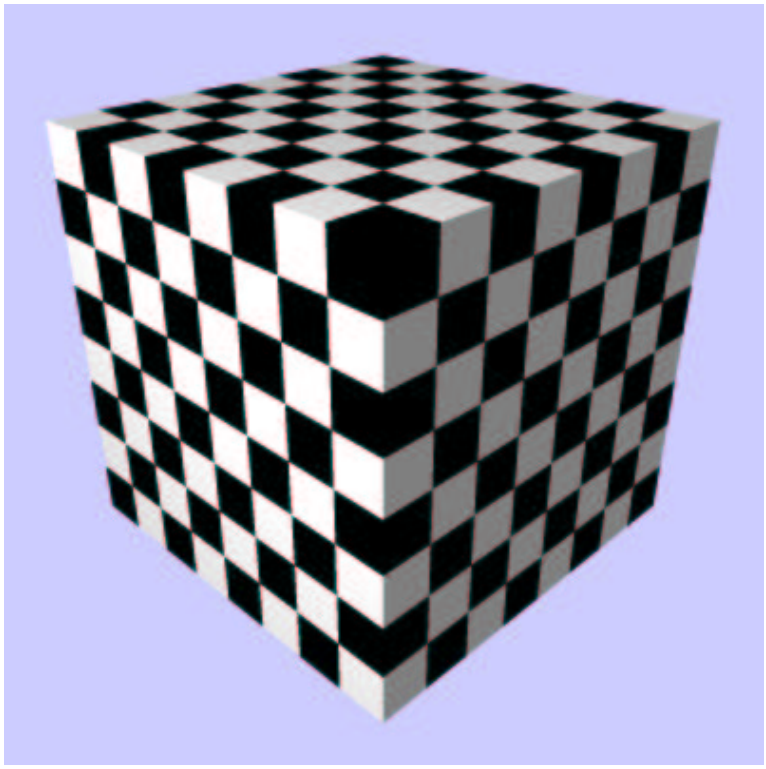

Figure 4: A better view of the cube.

The camera model is created by replacing the call to **vluCamera** with

```
vlCamera(&camera);
```

It may also be nice to make the resulting image square, as there is a lot of emptiness between the object and the edge of the picture on both sides. Modify the call to **vlImage** as follows:

```
vlImage("output.ppm", 1.0, 400, 400);
```

The image resulting from your work should appear as Figure 4.

7 Non-Cuboid Objects

Our object is not strictly-speaking a cube, even though it looks like one. It is a bounded volume object with a constant unit opacity. Outside the cuboidal boundary, the system sets the opacity property (as well as all other properties) to zero. Hence the cuboidal appearance of our objects so far!

In order to achieve other shapes, it is necessary to manipulate the opacity property of an object in a very particular fashion. Let us set about switching the cube for a sphere.

Earlier on, we made use of a field function to set the color components of our object. It just so happened that the function we applied (`VLU_CHESS`) assigned an alternating series of unit and zero values to the red, green and blue fields in the three dimensions. We could easily define other functions to assign a different arrangement of values to one or more fields.

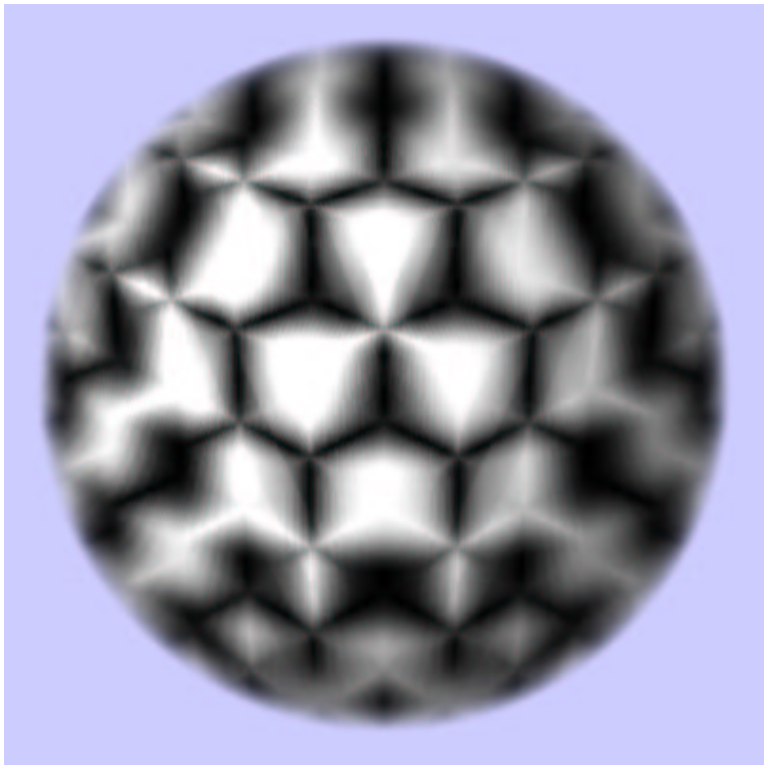

Figure 5: A fuzzy, textured sphere.

The VLU_SPHERE function is an example of what we call a *distance field*[1]. This particular function assigns values to all fields of an object such that they are unity at the center, and fall off linearly to zero at a distance of 0.5.

Let us apply this function to the VL_F and VL_O fields of our object:

```
vlFunction(VL_O | VL_F, &VLU_SPHERE, 0);
```

You may wish to modify the camera slightly before you render so that the object appears directly in the center of the image. Try reducing the viewport to 0.5 in both dimensions and modifying the `lookat` vector to $[0, 0, 0]$. This image may take a couple of minutes to render so you may also want to turn super-sampling off, either by removing the call to **vlSuperSample**, or by setting its argument to 1. The output of this program should look like Figure 5.

The sphere is fuzzy which makes it hard to recognize the checkered texture. This is because the opacity doesn't demonstrate a sudden transition from zero to unity (or anything near unity). We will address this point in a second. Beforehand, type the following command into the scene block (i.e., outside the object block):

```
vlStepSize(1.0, 50.0);
```

Also, increase the ambient reflection component (VL_KA) to 1.0 and reduce the diffuse reflection component (VL_KD) to zero. If you run the program now then you should see an image that looks like Figure 6.

[1] Several distance fields are included in the auxiliary library—you may want to try a few.

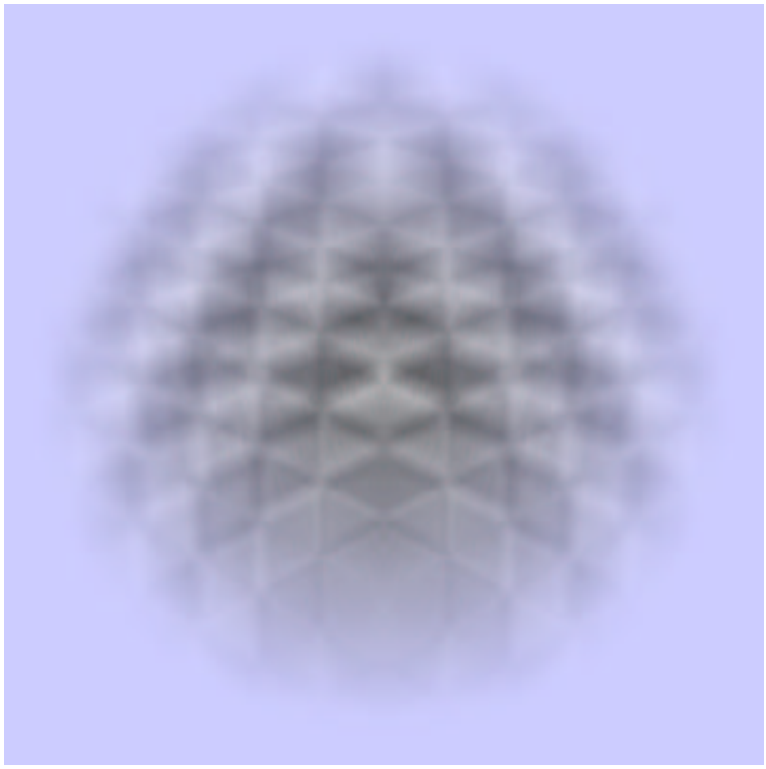

Figure 6: An almost-transparent sphere.

The **vlStepSize** command affects how the renderer samples objects. The first argument specifies the distance between sampling points, while the second argument regulates color opacity. More specifically, this second argument says "*if the sampling distance were this value then the color accumulation would be correct*." The value we've specified above is quite large, relative to the dimensions of the object. Therefore, color will accumulate slowly and so the object will look almost transparent.

The reason behind modifying the ambient and diffuse reflection coefficients is that amorphous matter (semi-transparent, voluminous objects) tend to look much darker than they ought with too much diffuse reflection, so we reduce this and increase the amount of ambient reflection to compensate.

8 Implementing Transfer Functions

In reality, there are a number of ways in which we can increase the sharpness of the sphere using the *vlib* API. The method we shall employ here is the transfer function, or *mapping*. In the context of *vlib*, a mapping transforms scalar data into scalar data using one of two approaches. We shall employ the "lookup table" approach in this section.

In your code, declare an array of `vlfloat` values and a simple constant (for clarity) as follows:

```
vlfloat map[] = {
    0.001, 0.0,
    1.000, 1.0
};

#define MAPPING_ID 1
```

This array provides the data for the mapping, which shall be explained in a moment. Before we can use the mapping, we must declare the data to the system. This is achieved by calling the following from within the scene block (not inside an object block):

```
vlMapping(MAPPING_ID, VL_LOOKUP, 1, 2, map);
```

This mapping is subsequently referenced by its identifier `MAPPING_ID`. Don't worry about the second argument just yet. If you look at the mapping array, you will see that it has been arranged into two rows and two columns. The leftmost column defines a series of transfer function *control points* (input values), while the right column is what we can call *output*. The usage of these shall be explained shortly. The API permits any number of output columns. However, in this case, we have a single output column and so the third argument to **vlMapping** is 1. The fourth argument is the number of rows, which in this case is 2. The last argument is a pointer to the first element in the data array.

To recap, the spherical distance fields says that the value on a field at a point will be greater than zero if the point is within a 0.5 radius of the center of the sphere. This is in terms of local object space and not world coordinates. Assume that we apply this spherical distance field to the `VL_F` field only. We would like to define the opacity field of our object such that it is unity only when `VL_F` is greater than zero. By the same token, we would like the opacity field to be zero wherever the `VL_F` field is also zero.

The mapping that we've defined is capable of performing just this. The lookup mapping takes an input value (in our case, the value on `VL_F`, as we will soon show), and runs down the left hand column of the mapping array until it locates a value that is greater or equal to the input value. It then steps across to one of the output columns and returns the value at that position in the array[2].

Modify the object definition so that the opacity field is not defined by the spherical distance field. Then, insert the following line *after* the call to **vlFunction** that sets the `VL_F` field:

```
vlMap(VL_O, VL_F, MAPPING_ID, 1);
```

The second argument states that the `VL_F` field will be used as input to the mapping function. The final argument is an index to the output data. As we have only one output column defined in our array, this number can only be 1.

A value of zero on `VL_F` will map to zero using the mapping we've defined. All values of `VL_F` above 0.001 will map to unity. The result is applied to the opacity field of our object. Now, reduce the ambient reflection field (`VL_KA`) to 0.5 and increase the diffuse reflection coefficient field (`VL_KD`) to 0.8. If you run the program as it is now then you should be presented with the image shown in Figure 7.

[2]This description assumes that the rows are sorted into ascending order on the leftmost column. In practice, the user needn't worry about sorting as *vlib* does it automatically.

Figure 7: A solid sphere with some aliasing.

9 Anti-Aliasing

We saw earlier how the amount of aliasing can be reduced by super-sampling with the **vlSuperSample** command. Simply firing multiple rays per pixel is not always sufficient for anti-aliasing in discrete ray-tracing. The interval between successive sampling points must also be taken into account, as should the rendering algorithm in general.

By default, the *vlib* API requests that the underlying rendering system uses the direct volume rendering (DVR) algorithm. DVR is most suited to rendering voluminous (amorphous) matter as it samples rays uniformly, without regard for nature of the scene and its objects. In contrast, the direct surface rendering (DSR) algorithm attempts to locate user-definable isosurfaces and may perform adaptive sampling instead.

For the solid sphere that we now have, DSR will invariably reduce the amount of aliasing apparent between the black and white checkers. However, we shall reserve the use of this algorithm until later. To improve the result of the DVR algorithm, simply reduce the sampling interval from 1.0. Try a value of 0.3. The improvement gleaned by lowering the interval is very dependent on the object. In our case, we would get a similar output if we were to reduce the interval to 0.03 or even 0.003, however the rendering time would eventually increase phenomenally.

```
vlStepSize(0.3, 50.0);
```

The second argument is now redundant as there is no longer any color accumulation in the strictest sense of the term. In our sphere scene, a ray will encounter a fully-opaque voxel and will stop. Or, a

Figure 8: A sphere with virtually no aliasing artefacts.

ray will never reach a voxel with any opacity and will consequently terminate on leaving the scene.

Increase the argument to **vlSuperSample** back to 3 (or reinstate the command if it was removed). If you run your program now then it should produce an image similar to the last but with far fewer aliasing artefacts. The expected output is shown in Figure 8.

10 Incorporating Datasets

We shall now make the scene a little more interesting by placing a chess piece on top of the sphere. Start by scaling and translating the sphere object so that it will make a suitable floor for the chess piece to sit on. Modify the transformations so that they read

```
vlTranslate(-0.5, -1.0, -0.5);
vlScale(500.0, 500.0, 500.0);
vlRotate(0.0, 45.0, 0.0);
```

By translating by -1.0 in the vertical axis initially, the object can be scaled by any amount and it will not rise above $y = 0$ at any point. This is useful as we can now position a second object on top.

In a manner very similar to the mapping (i.e., transfer function) definition, a dataset must be declared to the *vlib* API before it may be employed in the specification of a volume object. For the sake of clarity in this demonstration, declare the following constant

```
#define DATASET_ID 1
```

We shall use this symbolic constant as the identifier of the chess piece dataset through our program. Now, locate the pawn chess piece dataset (`pawn2.vlb`) and load it by calling

```
vlLoad(DATASET_ID, "pawn2.vlb");
```

outside the existing object block. In practice, you should make a habit of passing the full path and file name of the dataset to the **vlLoad** command. For single processing mode however (as we are using now), a relative path is fine. If you intend to make use of the parallel processing functionality of the *vlib* system then a relative path will not suffice as the slave processes will be unable to find the data.

Include a second object in your scene by entering the following specification in the scene block (although permitted, do not enclose it within the first object as we don't wish to create a hierarchical object).

```
vlObject();
    vlTranslate(-0.5, -0.5, -0.5);
    vlRotate(90.0, 0.0, 0.0);
    vlTranslate(0.0, 0.5, 0.0);
    vlScale(90.0, 90.0, 90.0);

    vlDataset(VL_F, DATASET_ID, VL_FALSE);
    vlEstimateNormal();

    vlSet(VL_O, 1.0);
    vlSet(VL_R | VL_G | VL_B, 1.0);
    vlSet(VL_KA, 0.5);
    vlSet(VL_KD, 0.8);
vlEnd();
```

The only unfamiliar field specification command here is **vlDataset**. This command, in the example above, is used to associate the pawn dataset (referenced by its identifier) with the `VL_F` field of the volume object. The third argument to the command specifies whether the values in the dataset should be normalised according to the normalisation values specified in the header of the `pawn2.vlb` dataset file. A dataset alone doesn't offer a total mapping from spatial positions to scalar values so the API implicitly accompanies the association between dataset and field with an interpolation function. By default, a trilinear filter is used but this may be overridden. If you run the program now then the resulting image should look like Figure 9.

11 Direct Surface Rendering

Although our new object, as seen in Figure 9, doesn't look much like a chess piece, the object specification itself is quite correct. With the information presented so far, it will appear as though we've produced a white object with opacity and reflection coefficients very similar to the cube given in earlier examples. The only difference is in the setting of the `VL_F` field, which effectively defines the curvature (or rather the normals) of the object.

Let us render an isosurface defined over the `VL_F` field. To achieve this, it is necessary to switch into DSR mode just before the object declaration of this object (i.e., before the call to **vlObject**), using:

```
vlRenderType(VL_DSR);
```

Figure 9: A multi-object scene. The chess piece is not yet visible.

Of the two objects now present in your scene, if the chess piece is the first object that you declare then be sure to switch back to DVR mode just before the declaration of the second object. Do this by calling:

```
vlRenderType(VL_DVR);
```

If you try rendering the image now then it will seem as though the chess piece object has completely disappeared. This is because we've asked the system to use an isosurface rendering mechanism but we've not yet specified any isosurfaces to render. Try specifying an isosurface of 0.85 for the `VL_F` field of the chess piece object. Somewhere within the object declaration, insert the command:

```
vlSurface(VL_F, 0.85);
```

Now, if you run the program you should produce an image similar to Figure 10.

12 Specular Highlight

Although our new object is now quite recognizable as a pawn chess piece, we can improve its appearance greatly by applying both a solid texture and a specular highlight. You already have the knowledge for the texture: simply remove the **vlSet** command that specifies the color and replace it with:

```
vlFunction(VL_R | VL_G | VL_B, &VLU_WOOD, 0);
```

Figure 10: A DSR rendered chess piece.

This will apply the solid wood texture contained in the auxiliary library.

Fortunately, specifying specular highlights in direct surface rendering is easy. We simply provide additional information regarding the specular reflection VL_KS and exponent VL_N properties. Try changing the reflection coefficients of the chess piece to the following

```
vlSet(VL_KA, 0.3);
vlSet(VL_KD, 0.6);
vlSet(VL_KS, 0.8);
vlSet(VL_N, 30.0);
```

To make the chess piece the main focus of our new image, move the camera in slightly by changing its view reference point to $(0, 200, -200)$. Before you render, be aware that this image will take a few minutes to produce. For now, we shall reduce the waiting time by "cropping" the chess object slightly. If you compare Figures 9 and 10 then you will see just how small the chess piece is in relation to its bounding box. Ironically, the empty space on either side of the pawn takes the most amount of time to render, as discrete ray-tracing involves sampling the whole way through an object from front to back. Minimize this space by adding the following command to the object specification:

```
vlBoundary(0.7, 0.7, 1.0);
```

This command will reduce the bounding box by a factor of 0.7 in the x and y dimensions, without actually scaling the object. An increase in rendering speed may also be gleaned by increasing the sampling interval. However, a sensible method is to employ parallel processing to make the best use of your available computing power. We will not discuss this further just yet.

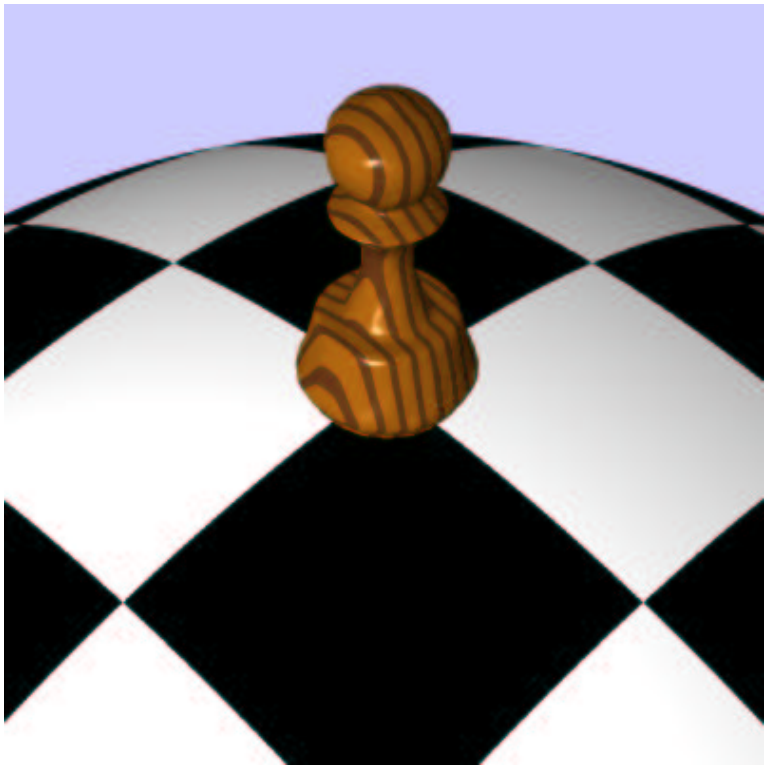

Figure 11: A more realistic looking chess piece with texturing and highlights.

If you render the image now, it should look like Figure 11. Discussions concerning specular highlights in direct volume rendering are presented in Section 14.

13 Shadows

A true feeling of depth is rarely achieved in an image with no shadows. For instance, you may ask yourself exactly how far the chess piece is from the top of the sphere. It may look as though the two objects are physically touching but are they really?

Before we add shadows to our scene, let's once again modify the reflection coefficients of the chess piece. You may agree that now we have a texture on the pawn it looks a little dark. Once we incorporate shadows, this may even be seen as a little confusing. The light source is strong enough to cause a sharp highlight in the object and to cast a shadow on the ground, so the object should really be a little brighter. Push the ambient reflection coefficient of the object (VL_KA) up to 0.4, and the diffuse value (VL_KD) up to 0.8.

By default, the *vlib* API instructs objects to cast shadows but not to receive them. This is for efficiency reasons as shadow computation is usually rather expensive in discrete ray-tracing. We would like the sphere object to receive shadows, so type the following command just before its definition (i.e., just before the sphere's object block):

```
vlToggle(VL_RECEIVESHADOWS, VL_TRUE);
```

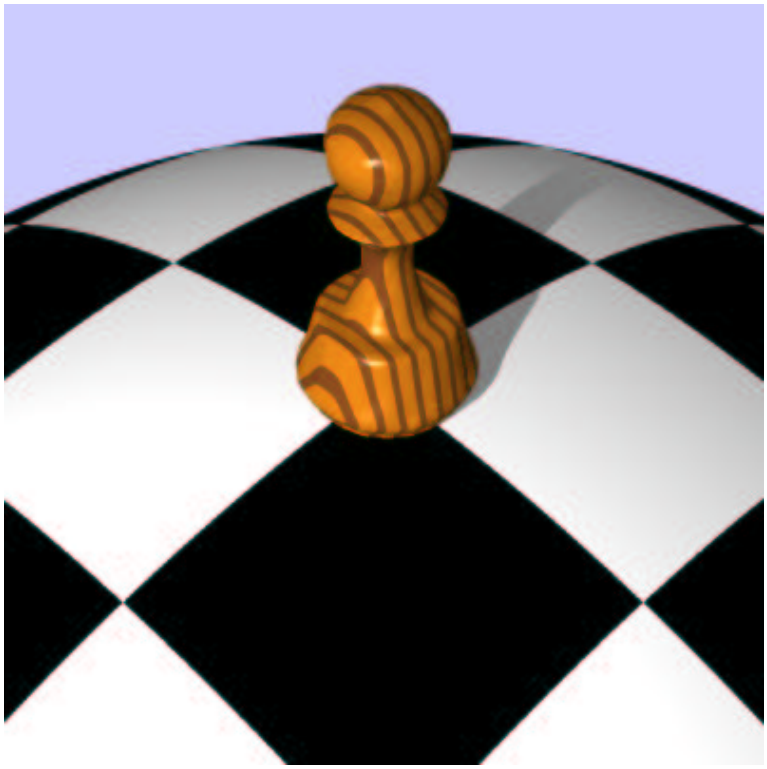

Figure 12: Shadows give a better understanding of depth.

If you have placed the sphere's object block above that of the chess piece, you will have to tell the system to prevent the chess piece from receiving shadows. Do this by calling **vlToggle** just before the chess piece's object block but with VL_FALSE as the second argument.

If you run the program now then it should produce output similar to Figure 12. You will notice that a better understanding of depth is gleaned with the shadow. If you look carefully at the base of the pawn in relation to its shadow, you will notice that the pawn is actually ever so slightly raised above the surface of the sphere.

14 Reflections

Depth cueing may be enhanced further by making good use of reflections. Let's give the sphere a kind of metallic finish so that it reflects the pawn. In a manner similar to shadows, *vlib* allows objects to appear in reflections by default but not to cast reflections. Therefore, we must explicitly make an object reflective with the **vlToggle** command. Add the following code just before the sphere's object block:

```
vlToggle(VL_REFLECTOBJECTS | VL_REFLECTBACKGROUND,
    VL_TRUE);
```

Be sure to turn both the VL_REFLECTOBJECT and VL_REFLECTBACKGROUND rendering parameters off after the object declaration if the chess piece declaration follows that of the sphere.

A metallic object should really reflect the background color of the scene so we should use the VL_REFLECTBACKGROUND parameter. Compare this to a metal object reflecting the color of a room, or the blue color of the sky. However, neither reflection type (object nor background color) will have any affect until specular reflection and specular exponent components (VL_KS and VL_N) are provided in the specification of the sphere.

Be aware that we are still rendering the sphere with the direct volume rendering (DVR) algorithm. Applying the specular component is not as simple as with the direct surface rendering (DSR) algorithm. The *vlib* API doesn't enforce any dependency between an object's opacity and specular reflection properties, so in fact a completely transparent object (such as glass) can still produce a broad, specular shine.

We could use a mapping to specify the specular reflection coefficient in a manner similar to the way in which we specified the opacity property of the sphere. In Section 8, we defined mapping data with two rows and a single output column which represented opacity. Our simplest approach would be to extend the data by adding a new column to represent the specular reflection coefficient. Modify the mapping data so it reads as follows:

```
vlfloat map[] = {
    0.001, 0.0, 0.0,
    1.000, 1.0, 0.4
};
```

You will also need to change the third argument of **vlMapping** from 1 to 2 as there are now two output columns. By applying the mapping to the VL_KS field, and using the VL_F field as input, we can say that wherever the opacity is unity (i.e., VL_F is greater than 0.001), the value on VL_KS will be 0.4.

Incorporate the mapping into the specification of the sphere object by adding the following code to its declaration block:

```
vlMap(VL_KS, VL_F, MAPPING_ID, 2);
vlSet(VL_N, 30);
```

You will notice that the VL_N property is constant across the object. Although it is possible to generate the VL_N value using a mapping, it is unnecessary in this case. The specular exponent field has no effect wherever the specular reflection component is zero.

If you render the image now then it should look like Figure 13. Notice that the sense of depth is even further enhanced as we can clearly see a reflection of the bottom of the chess piece in the sphere. This indicates that the two objects are actually some distance apart.

15 Parallel Rendering

Unless you have a very fast computer, you will have noticed that some of the images have taken more than just a few seconds to render. The *vlib* API supports parallel rendering but this extra feature will work only if you have the Parallel Virtual Machine (PVM) installed on your computer. Assuming you have, the *vlib* implementation should have made a note of this automatically upon being built. If so, then you will have noticed the text PARALLEL MODE appear on the screen as you have been rendering.

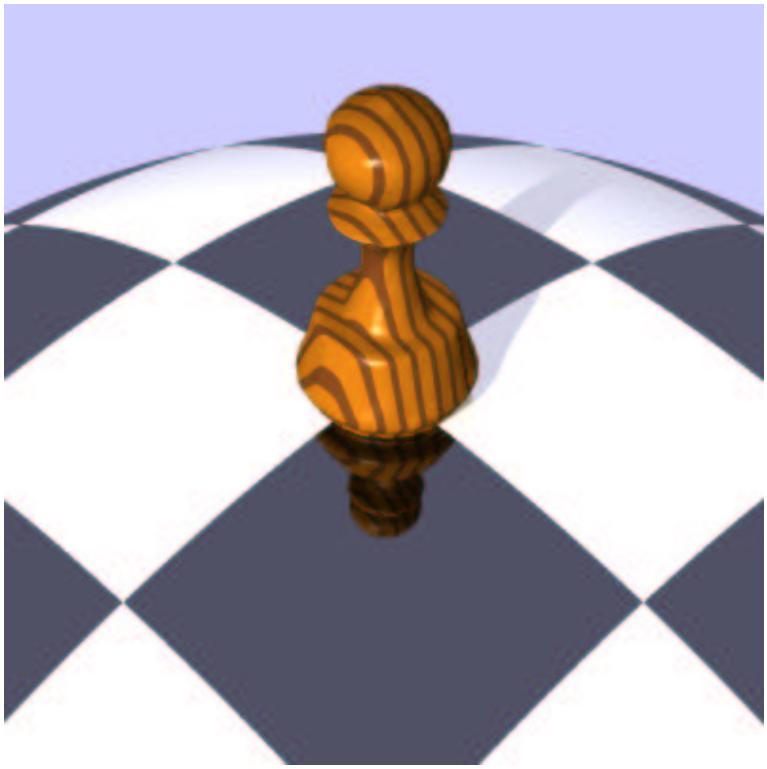

Figure 13: Reflections complement shadows in helping to understand depth.

The idea of parallelism is that the workload of the system is distributed over multiple computers in order to make the rendering process much faster. To actually make use of the parallel feature, you must ensure three things.

1. That PVM is installed and configured on each of your slave machines (the other computers that you wish to make use of).
2. That you have a `.rhosts` file located in your home directory, which contains the name of each slave machine, plus the name of your master computer.
3. Any volume datasets that your program uses are accessible from each of the slave machines. For instance, you may keep them in your account on the main file server, or may duplicate them and store them on the hard disk on each computer.

If these points cannot be satisfied then you will not be able to use parallel rendering.

Assuming you have managed the above successfully you can invoke parallel rendering by modifying the top part of your code as follows:

```
vlInit();
    vlParallel("computer", "file", argc, argv, 0, 1);
    vlScene();
       :
```

Replace `computer` with the name of a slave machine that you wish to use. For example

```
red.colors.org
```

Also, replace `file` with the full path and file name of the executable file that you're about to compile. You may repeat the **vlParallel** command any number of times—once for each slave computer. Be sure to declare your `main` function with the following parameters

```
int main(int argc, char *argv[])
```

16 Conclusion

This document has just scratched the surface of *vlib* and has explained how it may be used to create some interesting effects. The real power offered by the interface is exploitable through writing field functions, which provide a high level of non-uniformity in an object specification.

Several demonstrations programs are available on the *vlib* Web site, which will give you ideas for your own programs. The specification document is suggested further reading for users wishing to gain more power when programming with the interface. This too is downloadable from the Web. The address is given in the first section of this document.