Chapter 8

vlib: A Volume Graphics API

Contents

8.1 Introduction

It may be fair to say that the community is taking an *ad hoc* approach to advancing the field of volume graphics, with different research groups independently devising algorithms and creating in-house software systems to meet their specific needs. However, of these software systems, very few are released publicly so in the most part, volume graphics unfortunately remains exclusive to universities or to those with adequate knowledge, skill and time to develop modelling and rendering systems of their own. Yet, it is our belief that volume graphics will only ever receive the recognition that it deserves through the development and widespread distribution of not only volume graphics images, but also both commercial and freeware volume modelling and rendering software, as is the case in surface graphics.

In an attempt to step forwards towards this goal, we have designed and specified the first generic applications programming interface (API) for volume graphics. An API is a programmatical interface

between two software entities, such as a hardware driver and an operating system (OS), or an OS and a user interface. It provides a standard way for communicating between these entities through (often) many function prototypes and data types. A notable advantage of an API is the abstraction that it offers software developers. For example, consider the popular surface graphics API, OpenGL. A user of OpenGL does not know, or particularly care, how triangle primitives are rasterised or how the stack of transformation matrices is maintained. He knows simply that when an API command is called that the desired effect somehow results. An implementor of OpenGL similarly needs to have little consideration for the applications that the user of OpenGL will develop. He just needs to know that when, for instance, the polygon display function is called, that he should render a triangle. A second important advantage of an API is that it generally has a portable specification and can therefore be implemented on any platform using any language. OpenGL has implementations on several Windows platforms, MacOS and Linux, for example.

In summary, an API is needed for volume graphics for the following reasons:

- It has a portable specification and can be implemented using virtually any configuration of hardware architecture and operating system.
- It would enable field-based object modelling and rendering to become separated by an interface.
- An applications developer would be able to concentrate solely on producing a modelling system that complies with the API specification and would not need to learn about low-level rendering and optimisation details. Similarly, developers of rendering systems would become free from GUI design issues.
- Arbitrary modelling and rendering systems which both comply with the API specification are guaranteed to work together if connected. An illustration of this is shown in Figure 8.1.

It is vital that we make some sort of attempt to separate the modelling and rendering stages of the field-based object model. Consider, for instance, that the diverse requirements and capabilities of computer users in general have spurred the development of many types of user interfaces, from command-line modellers to GUIs. Clearly, it is infeasible to develop a new rendering system for each new modelling interface. Also, consider that users of modelling systems tend to have many rendering requirements, from accurate ray-tracing to real-time visualisation. Again, it is at best difficult to embed all possible types of rendering engine within a single graphics system. An API allows users to interchange modelling and rendering components to suit their particular needs.

The API created as part of this work reflects the principles of the field-based modelling framework, as outlined in Chapter 4. In this framework, objects are modelled largely as a collection of independent bounded scalar fields, which in turn are modelled in whole or in part using discrete or continuous volume data, or a hybrid of the two. We have shown throughout this thesis that this flexible framework not only has the capacity to advance volume graphics and visualisation in terms of graphical effects and visual quality, but is also capable of emulating techniques which are conventionally found in surface graphics only.

The format of this chapter is as follows. To begin, a brief review of relevant volume visualisation and surface graphics systems is given, and is followed by the motivation for this work. In Section 8.3, a number of design principles for the API are established which are later reflected on to explain the rationale behind the interface format. Next, an overview of the API, including a description of its

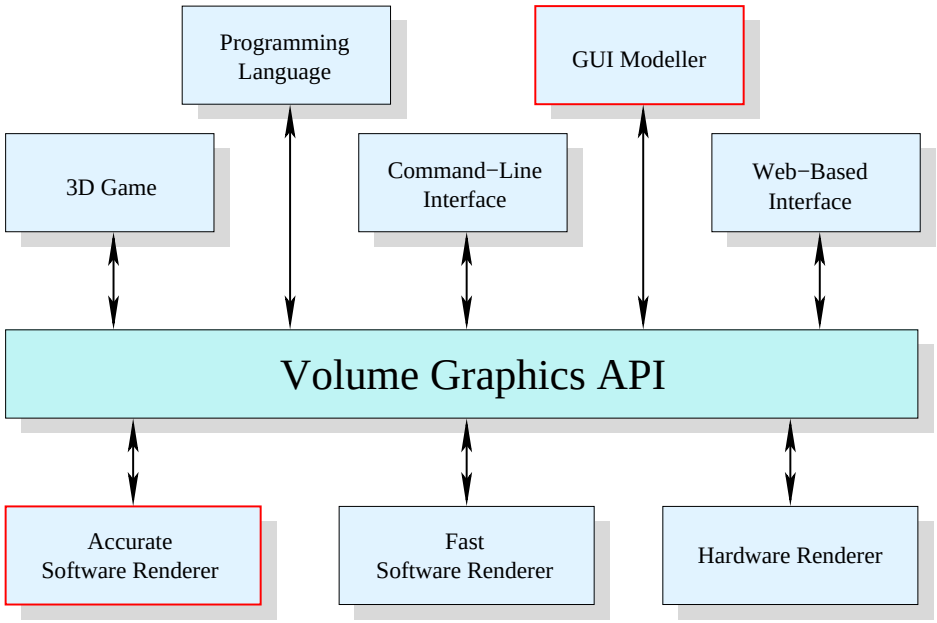


Figure 8.1: Any modeller can be connected to any renderer through the API.

structure, data types, command format and error handling, is given. In Section 8.6, the commands offered by the API for instantiating scalar fields of a volume object are described in detail. Then, normal estimation, object transformation and deformation techniques are discussed. In Section 8.10, an account of the rendering parameters supported by the API for achieving interesting render-time effects is given. Finally, the chapter closes with a few concluding comments.

8.2 Related Work

The popularity of surface graphics might be attributed to the wide range of modellers, renderers, games, APIs, CAD packages and other software systems that are available to meet a variety of demands. In the volume domain, the applications that exist tend to be aimed at scientists, rather than at artists, film makers or children, and often excel only at visualising pre-generated data such as that found in medicine, geology or astrology. In this section, the results of a small investigation into the different genres of software in both surface and volume domains is presented. A summary of these results and the motivation for developing a volume graphics API is then given.

8.2.1 Surface Graphics Systems

A high-level modelling language has been developed in conjunction with the POV-Ray [POV] ray-tracer. A variety of light sources, camera types and object types including quadrics, polygon meshes, bi-cubic parametric patches and swept surfaces can be defined through the language; CSG is also supported for creating compound objects. A few looping constructs, mathematical functions and

arrays are present in the language; these enable the user to define object geometry quickly using procedural methods rather than having to calculate vertices or control points manually. The user can set reflection coefficients homogeneously over an object to simulate a variety of finishes such as metal, plastic, stone and chalk. A surface can be further decorated by texture mapping and bump mapping. In addition to these features, a number of rendering effects are supported including fog, reflection, refraction and shadows, although limited control over these is offered. The scene description language allows the user to define complex scenes quickly and the underlying rendering engine produces very high quality images. However, the present version of POV-Ray does not support parallel processing and is consequently slow when rendering complex scenes.

OpenGL [WND97, KF97] is an API for interfacing between software applications and hardware rendering cards, although a number of compliant software-based rendering engines have also been developed. The API supports only low-level graphics primitives, namely points, lines and polygons, since only these are readily implemented in hardware. However, the coordinates of these primitives can be specified in either two or three dimensions and a significant amount of control over their appearance is offered by the API. An OpenGL rendering engine rasterises primitives into a frame buffer while optionally maintaining the depth of each pixel in a z-buffer. A number of per-fragment operations are offered, such as alpha blending for merging the colours of new geometry with that already present in the frame buffer, and hidden surface removal, where pixels are stored in the frame buffer only if the z-buffer suggests that they are sufficiently close to the viewpoint. The OpenGL API has implementations on a large number of platforms and is consequently used by developers of a wide range of software. However, it does not recognise a significant number of high-level object operations, such as CSG and sweeping, and nor can it render numerous surface data types, such as implicit or parametric surfaces, in their native form. The focus of OpenGL is on real-time rendering so it produces images significantly faster than POV-Ray, although of a substantially lower quality. Direct3D is an API by Microsoft, similar in functionality to OpenGL, but less widely implemented.

The RenderMan Interface [Ups90] is also an API, but one which interfaces between modelling and photo-realistic rendering systems. Unlike OpenGL, RenderMan Interface (RI) natively supports a wide range of geometric primitives including polygons, bi-cubic patches, NURBS, procedural primitives and quadrics, in addition to spatial set operations such as CSG. It also incorporates a powerful camera model to allow traditional photographic effects such as motion blurring and depth of field. A novel feature of the interface is a shading language which allows the user to dictate precisely how individual pixels should react to light. The language provides a number of built-in routines such as common trigonometrical functions, and also allows the user to compute over low-level rendering information including the view vector (from the eye to the pixel) and light vectors (from the pixel to the lights). RI has established itself as an important contribution to surface graphics and has recently been used by PIXAR to create several feature-length computer animated films, including *Toy Story* and *Monsters Inc*. The rendering speed attainable from an implementation of this API is clearly not as fast as OpenGL, however a hardware rendering system implementing the RI specification has been produced [Coo87].

Virtual Reality Modelling (previously *Markup*) *Language* (VRML) [VRM] is a standard for communicating 3D objects, worlds and multimedia across the Internet. It is an extended subset of the Open Inventor file format, facilitating relatively simple geometric modelling, texture mapping, material properties and fog. It supports an incredibly diverse range of application areas from collaborative visualisation to 3D games, with new uses emerging daily as the Internet grows and the VRML standard develops. 3D scenes described in the VRML file format are rendered on the client machine using

either a stand-alone rendering engine or an appropriate Web browser plug-in.

8.2.2 Volume Visualisation and Graphics Systems

A software library featuring an implementation of the shear-warp volume rendering algorithm [LL94] has been developed by Lacroute at Stanford University. The VolPack system, written in C, is capable of rendering regular volume datasets at reasonable quality with near-interactive frame-rates on current entry-level PCs. It supports a few rendering effects, such as depth cueing and shadows, although it does not feature any modelling capabilities, nor does it support many of the recent developments in volume graphics, such as multi-volume rendering. A UNIX implementation of VolPack is freely available to non-profit organisations and can be downloaded from the Stanford University Web site [Lac].

An object-oriented API for volume visualisation has been developed by Kitware. The Visualisation Toolkit (VTK) [SML98] supports an intermix of geometric and volumetric objects within a single environment, and will utilise any surface and volume rendering hardware boards installed in the user's system. A number of features and effects are supported by the API, many of which are not specific to volume graphics and are also facilitated by POV-Ray, such as an advanced camera model supporting depth of field and field of view, and a few types of light sources. In terms of volume datasets, a great number of configurations of topology and geometry are supported. A data object can be instantiated either by loading discrete data (e.g., CT medical data) from disk, or by discretising a continuous mathematical function at run-time. VTK cannot render continuous functions directly, as we can in our field-based modelling framework. Instead, it makes continuous functions discrete, stores them in memory and then reconstructs them at render-time using one of many supported reconstruction filters. Hence, fine detail or rapid changes in a continuous function might not be captured fully, leading to smoothed images and possible mis-representations of objects. The API includes commands for performing iso-surface extraction of volume datasets and also Delaunay triangulation of unstructured point sets. In summary, the VTK API is versatile and widely-used. However, it does not support many effects commonly required in computer graphics, including shadows, reflections and refraction, nor does it allow the user to compute over and perturb normal vectors, which are useful abilities in volume graphics.

A few GUI-based applications, including Open Visualization Data Explorer (DX)[1], AVS from Advanced Visual Systems [AVS92], and IRIS Explorer, use the concept of a *data-flow model* for volume visualisation. A data-flow model involves forming a network (Figure 8.2) to represent the sequence of data transformations and operations required for visualisation by connecting a number of *modules*. A link between two modules indicates that the result of the operation represented by the first module (e.g., rotation, crop, transfer function) should be passed as input to the second. DX and AVS both allow interactive visualisation by updating the image window as soon as a modification to the data-flow network is made. An advantage of the data-flow model is the independence and re-usability of modules. A disadvantage is that the user may have to peruse a large number of modules and learn how they connect together to form the network required to achieve a particular visualisation. DX is freely available to non-profit organisations; AVS is available commercially.

A subset of the RenderMan shading language has been extended by Corrie and Mackerras [CM93b]

[1] Formerly known as IBM Data Explorer.

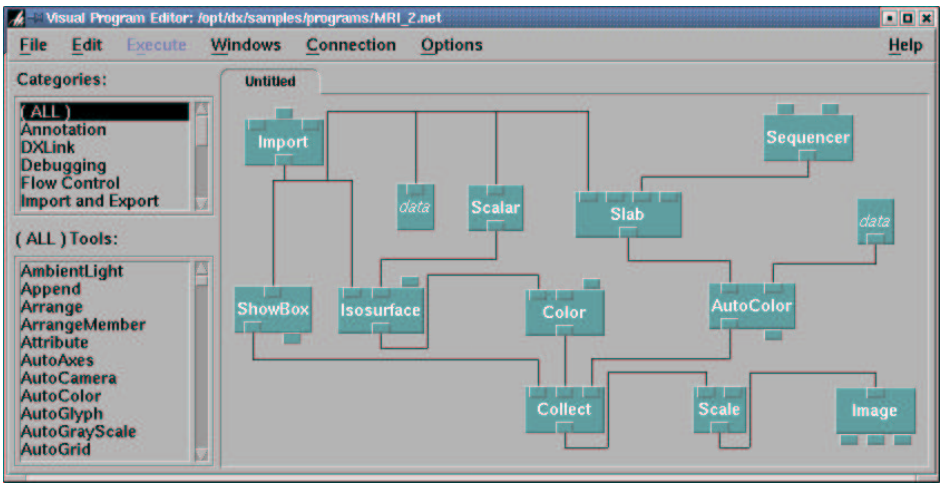


Figure 8.2: A network in a data-flow system.

to support the creation of custom shading and rendering models for visualising volume datasets. A disadvantage of most visualisation systems is that they tend to have hard-wired shading models and a small number of fixed rendering methods, which Corrie and Mackerras point out does not allow an effective visualisation of data from all scientific domains. However, the shading language enables the scientist to create purpose-specific *data shaders* to dictate precisely how light should interact with data at each sample point, and how samples along a ray should combine to form a pixel intensity. A disadvantage of this shading language is that end-users will require some computer programming and computer graphics knowledge in order to create a data shader. Also, the flexibility offered is reported to cause a 70 percent increase in rendering time. A full specification of the data shading language can be found in Corrie and Mackerras's technical report [CM93a].

The group at Stony Brook, New York, have developed a GUI-based rendering application for volume graphics. Their system, VolVis [AHH+94], comprises a series of components for processing, measuring, manipulating and rendering both geometric and volumetric objects. The processing component allows the user to convert an object between geometric and volumetric representations using built-in surface extraction and object filtering functions. A variety of quantitative information about objects in either representation, such as volume or surface area, can be determined with the measurement component. The manipulation component facilitates simple operations including transformation and cutting, which can be performed interactively and viewed using one of several low-quality but real-time rendering algorithms. The final component, the rendering engine, similarly supports a number of rendering algorithms, several of which are parameterised to allow the user to dictate the speed and quality of rendering. VolVis has been implemented on Windows and UNIX platforms and is freely available to non-profit organisations. An on-line registration form must be completed before the software can be downloaded.

8.2.3 Motivation

An API serves an indispensable role in developing a diverse range of reliable graphics applications. Given that an API also encourages teams of programmers to stay within their respective area of expertise (e.g., developers of rendering systems need not concern themselves with the complex issues relating to graphical user interface design), the quality of the software produced is most often enhanced.

To understand precisely the impact of an API, one needs only to look at existing APIs in the surface domain, such as OpenGL, which currently has software implementations on a variety of UNIX, Windows and Macintosh platforms. A great number of software applications, from CAD systems to games, use OpenGL for rendering and are consequently portable since most platform-specific details regarding fast low-level polygon rasterisation are abstracted away from the developer. A second important motivation for developers to utilise OpenGL is the fact that the majority of polygon rendering hardware cards for the PC and Macintosh now support the interface.

The few APIs that exist in the volume domain are concerned largely with data visualisation and not with modelling high-quality volume objects and the graphical effects that traditionally enrich surface rendering such as reflection, refraction and shadows. An overwhelmingly small number of volume graphics (not volume visualisation) applications currently exist, such as VolVis, but these are not at all expandable or useful in developing more advanced software systems, and tend to support only a small number of developments in the field of volume graphics.

The work presented in this chapter is motivated by the lack of available volume graphics software. It is believed that the introduction of a volume graphics API will accelerate the development of general purpose volume modelling and rendering software, and will consequently improve the availability of volume graphics methods to scientists, researchers, software developers, artists, graphics enthusiasts or hobbyists and others.

8.3 Design Principles

An interface of any type is essentially a means for communicating between two or more entities, and a good design is essential if effective communication is to be achieved. A significant proportion of the literature describes the principles of interface design, particularly in the context of graphical user interfaces (GUIs), which relay information between human computer users and any form of backend software application, such as an operating system or an accounting package. Usually, the primary aim of a GUI is to make its users feel "comfortable" in front of the computer and to provide them with a means for navigating the system quickly and with minimal confusion. In order to create an effective interface, a series of smaller *design goals* should first be decided and concepts and general mechanisms for achieving these goals, known as *design principles*, should then be sought.

A common design goal in the development of a GUI is consistency over the interface. An example of a design principle for achieving consistency is to *always use the same hot key for a given task* since the user will become familiar with certain keys, even from unfamiliar areas of a package. (A frequent user of Microsoft Windows, for instance, will know that *control-C* always invokes the copy command.) Another common design goal is simplicity, and is often achieved by employing iconic representations of tasks in the form of metaphors, such as a picture of a dustbin for *delete file*, a padlock for *lock*

desktop, and a filing cabinet for *view files*. Such icons enable user to quickly recognise tasks and may avoid them having to sift through complex menu systems.

A team developing a different type of interface will invariably have a different primary aim. A file format, for instance, is an interface between raw data and a software application, and might be designed with flexibility in mind. In designing the volume graphics API, the following design goals and their subsequent design principles have been decided upon. In the sequel, we show that these design principles are reflected in the specification of the API.

- **Cohesion.** A command should perform a single task. Moreover, the name of the command should provide a good indication of the nature of the task. If this design principle is adhered to then the user will find the API easy to learn and understand.
- **Completeness.** The API should provide all functionality necessary for modelling and rendering a volumetric environment. Whereas the aim of this work is not to produce an exhaustive interface for volume modelling, it should not possess any significant gaps.
- **Consistency.** All commands should follow a consistent naming convention. If groups of commands have similar arguments then the order of arguments should be fixed. In achieving this goal, the amount of time spent by the user referencing documentation, and the liklihood of programming errors arising, will be minimised
- **Encapsulation.** All details concerning the implementation of the API, such as internal variables, data structures and functions, should be hidden away from the user. If internal details are visible then they may be changed inadvertently causing bugs or crashes. If all implementation details are hidden then rendering engines can be interchanged safely without requiring any changes to the user's code.
- **Portability.** The API should not rely on any non-standard features of an operating system, a programming language or a hardware architecture, such as unusual third party libraries. If this design principle is adhered to then the API can be ported to other platforms with minimal effort.

In terms of implementing the API, an entirely different set of goals must be achieved, such as code correctness, efficiency, extendibility, legibility, and so on. However, these points will not be discussed since they relate more to general programming and less to the rationale of the API design with which we are concerned here.

8.4 Conventions Used

A number of constants, data types, commands and particular vocabulary will be employed throughout this chapter while describing the API. Here, we briefly describe the conventions used to avoid the possibility of any ambiguity or confusion arising later on.

Vocabulary	Meaning
must	Used to indicate that something is intrinsic to the correct functioning of the interface or the underlying rendering engine and has to be observed by an implementor at all times. For example, "commands *must* reset the error flag."
should	Used to indicate a recommendation or suggestion to either an implementor or a user of the API. It does not refer to a rule that must be adhered in order for the interface to function correctly. For example, "users *should* periodically check for errors."
will	Used to indicate a consequence of an action. For example, "calling this command from outside the legal scope *will* cause an error."
command	Used to reference a sub-routine of the API. The term *function* is reserved for other purposes and shall not be used synonymously with *command*.

The reader should also observe that all data types will be printed in a lowercase courier font (e.g., `vlcamera`), all constants in an uppercase courier font (e.g., `VL_BOX`), all parameter names in an italic font (e.g., *fields*) and command names in an Arial font (e.g., vlProject).

8.5 System Overview

vlib (short for "volume library") is an interface for communicating information between high-level modelling applications and low-level volume rendering engines. A *vlib implementation* is a library of sub-routines in which all *vlib* functions and data types implemented comply precisely with the specification outlined in the remainder of this chapter. It is not essential for a *vlib* implementation to support the full *vlib* specification, and in some instances, a full implementation might not be possible due to hardware constraints, or may simply be deemed unnecessary. It may contain extra sub-routines or data structures however, to support the use of *vlib*, but only if the names of such additions do not conflict with those listed here.

The *vlib* API comprises several dozen commands and a few data types for controlling the camera, scene lighting, rendering settings, scene graph construction and general system tasks such as error handling. In relation to popular APIs such as OpenGL in the surface domain, the number of commands specified in this interface is relatively small. It should be noted however that the goal of this work is to develop the core elements of a volume graphics API which could be expanded or altered slightly in time as the need arises, and *not* to support numerous image processing features, camera types and general utilities that are not specific to volume graphics.

In Chapter 9, we describe the technical implementation details surrounding several major components of this API. In this chapter however, we provide the specification only and omit all details concerning implementation and low-level algorithms.

A short C program is shown in Figure 8.3 (left) that uses the *vlib* interface to model and render a simple parabolic object with a stone-like texture (right). A consistency should be noticed among the *vlib* commands, shown in bold face: the fact that all are prefixed by vl. A simple naming convention such

```
01 vlfloat myMap[] =
02     {0., 0., 0., 1., 1., 0.3};
03 main() {
04 vlInit();
05    vlScene();
06       vluCamera(175., 250., -350.,
07           175., 140., 70.);
08       vlImage("fig.ppm", 1., 100, 100);
09       vlBackground(WHITE);
10       vlMapping(1, VL_LOOKUP, 2, 2, myMap);
11       vlPointLight(-200., 380., -200.,
12           VL_NOATTENUATION, WHITE, 1.);
13       vlAmbient(WHITE, 1.);
14       vlObject();
15          vlScale(350., 300., 300.);
16          vlFunction(VL_F, &parabola, 0);
17          vlEstimateNormal();
18          vlMap(VL_O, VL_F, 1, 1);
19          vlMap(VL_KS, VL_F, 1, 2);
20          vlFunction(VL_R|VL_G|VL_B, &stone, 0);
21          vlSet(VL_KA, 0.3);
22          vlSet(VL_KD, 0.6);
23          vlSet(VL_N, 20.);
24       vlEnd();
25    vlEnd();
26 vlEnd();
27 }
```

Figure 8.3: A short C program and the image it generates.

as this enables programmers to identify *vlib* commands in their code and also reduces the liklihood of command names conflicting with those in other libraries.

A hybrid procedural/declarative programming paradigm is used. A few elements of the interface are declarative in the sense that the camera, light sources, image properties and (sometimes) volume objects can be defined at any time from anywhere within a program. A few other elements are procedural, as will be shown later. It is evident from the example code that no object-orientation is used; this is so that an implementation of the interface can be used with as many languages on as many platforms as possible. However, we have experimented with writing C++ wrappers around the commands to produce an object-oriented interface, and have achieved successful results.

8.5.1 Constants and Data Types

A *vlib* implementation must define the data types outlined in Table 8.1, and these data types should be used in place of the programming language equivalents wherever possible. This is encouraged to enhance the portability of any application using this API and will consequently make the transferral from one implementation to another by the user a far less menial task.

The `vlbool` data type can be implemented using any appropriate data type provided by the language if no 1-bit integer types exist. The `vlvector` data type can be used to store both a coordinate and a vector. If a coordinate is held then the first, second and third array entries will store the x, y and z positions respectively, otherwise they will store the x, y and z vector components. This will be assumed later in this chapter. `vlfloat` can be represented internally using either 32- or 64-bit floating-point types, according to the rendering speed and accuracy desired by the implementor.

Table 8.1: Data types.

Type	C Equivalent	Description
`vlbool`	`char`	1-bit integer.
`vlenum`	`unsigned char`	8-bit integer.
`vlfloat`	`float` / `double`	32-/64-bit float.
`vlint`	`int`	32-bit integer.
`vlstring`	`char *`	Character string.
`vlvector`	`vlfloat [3]`	Array of three `vlfloat` values.

A number of constants must also be defined. In the most part, constants are intended to be used with only one command or command group and most therefore will be introduced with the appropriate commands as this chapter progresses. An exception is the `vlbool` constants which are returned by some commands and passed as input to others. They are `VL_FALSE` and `VL_TRUE`, the former of which must be defined as zero and the latter non-zero but otherwise arbitrarily.

In later sections, we show that multiple entities (e.g., scalar fields or rendering effects) are referenced simultaneously using a single command argument. By this, it is meant that of the constants that could be passed in that argument, two or more can be unioned together with addition or the bitwise OR operator. Therefore, the appropriate constants must be defined such that they can be encoded into a single value and decoded again unambiguously.

A *vlib* implementor should always define additional user-visible data types using lowercase characters, constants using uppercase characters and command names arbitrarily. The core elements of the API documented herein should be reproduced in an implementation exactly as shown.

8.5.2 Logical Blocks

A consequence of not employing an object-oriented programming paradigm for *vlib* is that users are forced to group calls to particular commands into logical blocks. The sample code in Figure 8.3 emphasises these blocks using indentation. In essence, a block is a group of commands pertaining to a single entity, such as a volume object or a scene. A few of the API commands described herein make sense semantically only when they are called from designated blocks. A transformation command for instance, is applicable only to a volume object and would therefore not make sense if called from outside a volume object block (e.g., line 15 of the sample code could not be reinserted between lines 13 and 14, say). A command called from outside its designated block or "scope" will cause a system error to occur. A discussion of error handling takes place in the following section.

All API commands, without exception, must be made between calls to:

```
void vlInit( void );
```

and

```
void vlEnd( void );
```

An API implementation should use these routines to perform any necessary initialisation and termination tasks, such as allocation and freeing of system memory, and assigning default values to

internal system variables. A number of scenes (zero or more) can be defined between calls to these commands, where a single scene is defined by nesting a scene description between calls to:

```
void vlScene( void );
```

and vlEnd. It should be noted that vlEnd is a general command used to terminate blocks; an implementation of this API should maintain a stack of open blocks and ensure that the correct block is terminated each time vlEnd is called. A number of volume objects (zero or more) can be defined between calls to these commands, where a single volume object is defined by nesting a volume object description between calls to:

```
void vlObject( void );
```

and vlEnd.

A scene description is shown between lines 5 and 25 in the sample code in Figure 8.3. A single volume object is defined in this scene between lines 14 and 24. It should be noted that no command is included in the API to explicitly trigger the rendering (or voxelisation) system. An implementation will produce an image (or a volume dataset, as will be discussed later) as soon as the user terminates a scene description with vlEnd, shown on line 25.

A chart indicating the various command scopes is shown in Table 8.2. A unique number is assigned to each scope; as each interface command is introduced in this chapter, the number of the scope from which it may be called will be given. If, for example, a command is legal from Scope 7 then it can be called from anywhere between vlInit and its corresponding vlEnd statement, including from render-time functions. If, on the other hand, a command is legal only from Scope 4 then it cannot be called from a volume object block, nor from any of the render-time functions. Evidently, Scope 7 contains all other scopes. The exact meaning of the *render-time* scopes will become evident as this chapter proceeds.

8.5.3 Error Handling

If a command is called from anywhere outside its legal scope then a system error will occur. A well-written program will periodically check for system errors at appropriate times (e.g., after loading a dataset to ensure that the file exists and has the correct structure) and will take necessary action if any kind of error is detected (e.g., it will re-prompt the user for the name of a valid dataset file). *vlib* commands do not actually return error values; rather, they set an internal system error flag which the user can access through the interface using the Scope 7 command:

```
vlenum vlError( void );
```

A summary of error codes is given in Table 8.3. We decided that the API commands should not return error codes indicating whether they have completed successfully or have failed since, by doing this, no command will be able to return any other information without introducing inconsistencies to the interface. If a *vlib* command is issued before vlInit then it will have no effect and will not set the internal error flag.

Table 8.2: Chart showing command scopes.

Render-Time Scopes:

field function:	5, 6, 7, 9, 10, 11, 12
STF function:	7, 10, 12
normal function:	7, 8, 10, 11
scale function:	7
up function:	7

Modelling-Time Scopes:

```
vlInit()
                        1, 4, 7
    vlScene()
                        2, 4, 7, 9
        vlObject()
                        3, 6, 7, 9, 11
        vlEnd()
                        2, 4, 7, 9
    vlEnd()
                        1, 4, 7
vlEnd()
```

Table 8.3: Possible error codes returned by the vlError command.

Error	Description
ERR_OK = 0	No error.
ERR_BADID	If *creating* an entity then the identifier already exists. If *referencing* an entity then no entity with the specified identifier exists.
ERR_BADPARAMS	Invalid command parameters.
ERR_FILEERROR	If *loading* a file then either the file could not be found or the contents of the file is invalid. If *creating* a file then the system has been unable to create a file with the specified name.
ERR_INCOMPLETE	Insufficient information has been provided to produce an output.
ERR_INSUFFICIENTMEMORY	The system failed to allocate enough memory for the operation.
ERR_OUTSIDESCOPE	A command has been called from outside its legal scope.
ERR_PVM	The PVM library has reported an error.

8.6 Modelling Scalar Fields

Most visualisation systems define volume objects by first applying a reconstruction filter to a discrete dataset to define a geometry field, and then defining the opacity and colour properties by passing the geometry data through carefully designed transfer functions—one for each visual property. It is our belief that this two-step approach to modelling is insufficient for achieving the flexibility that we frequently require in computer graphics. A key goal of *vlib* is to allow an extremely versatile specification of objects such that their shape is not restricted by discrete volume datasets. After all, discrete data is inherently large, and as new objects are added to a scene, so the memory consumption will grow until such a time when significant tradeoffs between image quality and volume resolution will have to be made. A much better solution would be to allow the geometry, as well as other properties of a volume object, to be defined by compact mathematical or procedural functions, which are not discretised, as in VTK, but are evaluated at render-time, as and when they are required.

The concrete object model described in Chapter 4 forms an integral part of the API. To recap, the model uses ten scalar fields to represent the major visual properties of a volume object, where each field is bounded by the unit domain, $[0,1]^3$. A left-handed coordinate system is assumed. So, part of the role of this API is to allow these fields to be defined over this domain in a flexible yet consistent manner. Since all ten fields are functions of the form $f : [0,1]^3 \longrightarrow \mathbb{R}$, it follows that any techniques supported by the API for instantiating one field should be supported equally for the rest, regardless of the attributes defined by the fields. A total of six core commands are provided by the interface for instantiating fields (Figure 8.4) and will be discussed in this section.

In principle, the commands for modelling scalar fields could be defined in one of two ways. An *explicit* (procedural) method could be used whereby a point, (x, y, z), is passed as an argument to a command to indicate that the result of some computation should be assigned to one or more fields at that location. The advantage of this approach is that fields can be defined in a non-sequential fashion, which is often how we create synthetic volume datasets using code. A drawback is that it is inherently difficult to determine the value on a field quickly without discretising the field beforehand. Alternatively, an *implicit* (functional) method could be used whereby coordinates are implied by the sampling process, and are not passed to any API commands by the user. In practice, this method is much more feasible since it is simple to implement and enables fast evaluation of field data, and the interface commands can be crafted in such a way that the coordinates of sampling points are made available for the user to compute over anyhow. Hence, we have adopted the implicit approach for the *vlib* API.

A consistent naming convention for command parameters will be used throughout this chapter for two reasons: (i) to reduce confusion for the reader; and (ii) so that there is no need to re-explain arguments if they are used with multiple commands. In addition to this, commands are given a consistent parameter ordering where appropriate so that we stay faithful to the *consistency* design principle, discussed earlier. A number of commands discussed in this section will feature a *fields* parameter. The *fields* parameter allows the user to specify which scalar fields (one or more) are allowed to be modified by the command. The ten fields are each referenced in the API by the following symbolic constants:

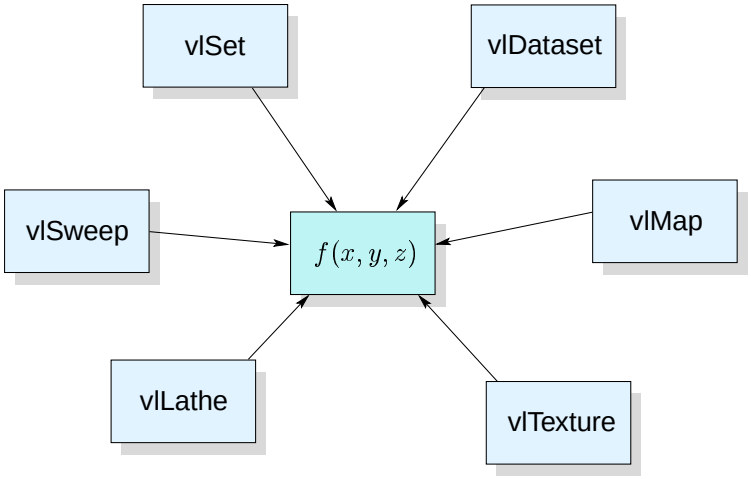


Figure 8.4: A field can be defined using one or more of six core commands.

Field	opacity	red	green	blue	ambient
Symbol	VL_O	VL_R	VL_G	VL_B	VL_KA
Field	diffuse	specular	exponent	IOR	geometry
Symbol	VL_KD	VL_KS	VL_N	VL_RFR	VL_F

The precise values of these integer constants do not need to be defined here, but it is important that each is a unique power of two, 2^n, for $n \geq 0$. This is so that they can be bitwise OR'd so that several fields can be referenced in a single command argument. If a parameter name is *field* (singular) then exactly one field must be referenced.

8.6.1 Explicit Assignments

The most flexible approach to defining an attribute field, A, of a volume object is to determine a scalar value, and assign it to $A(x, y, z)$ at each sampling point (x, y, z) visited by the discrete volume rendering process. The Scope 6 command:

`void` **vlSet**(`vlint` *fields*, `vlfloat` *c*);

performs this task and can be used to set $A(x, y, z) = c$, for all sampling points and all attribute fields referenced by the first argument. If this command is issued from Scope 3 (as in lines 21 to 23 of the sample code in Figure 8.3) then the referenced attribute fields will be completely constant over the bounds of the volume object, unless they are subsequently changed by other commands. However, vlSet can define attribute fields heterogeneously if called from Scope 5. A discussion of this will take place in Section 8.6.7.

8.6.2 Volume Datasets

The conventional approach to instantiating scalar fields in volume visualisation and graphics is to use discrete volume datasets. Interestingly, a number of issues have been encountered in specifying exactly how the API should interact with discrete voxel models. They include, deciding:

- Which topologies and geometries should be supported (e.g., point clouds, rectilinear grids, curvilinear grids, tetrahedral meshes).
- Appropriate sizes and primitive data types for volume elements (e.g., 8-bit unsigned integers, 16-bit signed integers, 32-bit real numbers).
- A logical representation of volume dataset (i.e., a suitable file format).
- How the volume graphics API should interface with the volume data.

We decided that versatility will be gained if volume datasets are self-descriptive. That is, if they contain a header describing their topology, geometry, voxel type and dimensions. The reason for this is twofold: First, if at some point in the future it is decided that a new data format should be supported then no changes to the API specification will need to be made. Second, it eliminates the need for users to interrogate their volume datasets to determine the necessary data attributes.

It is not our place to restrict the volume graphics community to only a selection of dataset formats. However, it has been observed that certain formats are more popular than others in certain applications. A variety of medical scanners (e.g., CT, MRI) and sculpting systems for modelling synthetic volume data generate regular 16-bit unsigned integer volumes. In addition, distance fields are frequently used in volume rendering and some distance transform implementations are known to generate up to 64-bit floating point volumes. Irregular volume data is seldom used in volume graphics, however it is frequently employed in volume visualisation applications (e.g., CFD simulation) and should therefore be supported by this interface.

The API has been designed in such a way that constraints on data formats are imposed only by the underlying rendering engine, and not by the programming interface. This will become evident shortly. The Scope 4 command:

`void` **vlLoad**(`vlint` *id*, `vlstring` *filename*);

loads a volume dataset, where *id* is a unique integer by which the data is subsequently referred. If the dataset contains vector data (e.g., RGB colour) then the first scalar component (e.g., the red channel) will be associated with identifier *id*, the second scalar component (e.g., the green channel) will be associated with identifier *id*+1, and so on. *filename* is the path and name of the file containing the volume data to be loaded. The format and attributes of the data are determined by decoding the contents of the file. If the data file format is recognised and the dataset is valid then it will be read into appropriate internal data structures, otherwise the `ERR_FILEERROR` error will be reported.

Today, very few file formats for storing volume datasets have been publicised but those in common use are either very complicated (e.g., VTK [SML98]) or can only be created using special software tools (e.g., F3D [ŠD02]). A simple file format has been developed as part of this work to represent the regular rectilinear volume datasets commonly used in volume graphics. The "VLB" format consists of a small text header with entries as shown in Table 8.4, followed by raw interleaved data values. Header entries must appear on separate lines. A hash (#) in the first column of a line in the header

indicates an annotation and the line will not be processed — any number of annotations are allowed. An advantage of this file format is that raw volume datasets (i.e., files with no header or footer) can be converted simply by adding a header using a text editor.

Table 8.4: The VLB file format.

Entry	Description
`VLIB.1`	Magic number.
$X\ Y\ Z$	Volume dimensions (positive integers).
W	Number of data elements per voxel (positive integer).
type	Data element type (see the *Type* column in Table 8.5).
`big`|`little`	Endianness of the volume data.
$X'\ Y'\ Z'$	Cell dimensions (positive reals).
$V_1\ V_2$	Minimum and maximum data values in the dataset.
$R_1\ R_2$	Range onto which data values should be normalised ($R_1 \neq R_2$).

A program using this API can use the following Scope 7 command to determine the resolution of a loaded dataset and the dimensions of its cells:

`void` **vlDatasetDim**(`vlint` *id*, `vlvector` *D*, `vlvector` *V*);

where *id* is the identifier associated with the dataset as passed to vlLoad. *D* and *V* are arrays of three scalar values that will be filled with the dimensions of the volume and the cells respectively. If the dataset file format does not have an entry stating the cell dimensions then *V* will be set to $(1, 1, 1)$. The Scope 6 command:

`void` **vlDataset**(`vlint` *fields*, `vlint` *id*, `vlbool` *normalise*);

defines one or more scalar fields by applying a reconstruction filter to a scalar volume dataset (or a scalar channel of a vector volume dataset), where *id* is the identifier associated with the dataset as passed to vlLoad. If *normalise* is `VL_TRUE` then all elements in the dataset will be normalised, either to the range $[0, 1]$, or to the minimum and maximum values specified in the dataset file if such values exist. This is particularly useful in the case of integer colour data since colour values outside the range $[0, 1]$ are illegal and will be clipped.

Table 8.5: Voxel types supported by the VLB file format.

Type	Constant	Description
`uint8`	`VL_UINT8`	8-bit unsigned integer.
`int8`	`VL_INT8`	8-bit signed integer.
`uint16`	`VL_UINT16`	16-bit unsigned integer.
`int16`	`VL_INT16`	16-bit signed integer.
`uint32`	`VL_UINT32`	32-bit unsigned integer.
`int32`	`VL_INT32`	32-bit signed integer.
`float`	`VL_FLOAT`	32-bit float.
`double`	`VL_DOUBLE`	64-bit float.

A reconstruction filter must be selected at any point before associating a dataset with a scalar field, since volume data elements are defined only at volume grid points. A detailed treatment of this subject is given in Section 2.4. A tri-linear reconstruction filter is used by default if no other filter is specified by the user. This choice was made due to its speed and reasonable accuracy. The Scope 9 command:

`void` **vlInterpolationType**(`vlenum` *filter*, `vlfloat` *B*, `vlfloat` *C*);

allows the user to specify which reconstruction filter should be used by subsequent calls to vlDataset. *filter* should be set to either `VL_NN` for nearest-neighbour, `VL_LINEAR` for tri-linear or `VL_CUBIC` for tri-cubic reconstruction. *B* and *C* are the parameters of the BC-spline equation and have no affect when *filter* is `VL_NN` or `VL_LINEAR`. vlDataset can be called inside an object block to allow data values and normal vectors (see later) to be estimated differently between calls to vlDataset.

8.6.3 Transfer Functions

Transfer functions are used almost exclusively for mapping digitised or simulated data onto opacity and colour attributes for the sake of image synthesis. We can see however, that they could potentially serve a far broader role in volume modelling, for example by allowing geometry to be mapped onto opacity, opacity to be mapped onto colour, one colour component to be mapped onto other colour components, and colour to be mapped onto geometry. They could be used to create interesting effects if a field is filtered through a number of different transfer functions repeatedly.

We have decided that the API should natively support the two transfer functions often described implicitly in the literature for volume visualisation. This is primarily because our experiments have suggested that these transfer functions have at least some use in modelling all fields of a synthetic volume object in some circumstances. The transfer functions that we describe here shall be known as the *lookup table* (LUT) transfer function and the *linear ramp* (LR) transfer function—they are defined similarly, both mathematically and in the context of the interface, and consequently the descriptions and discussions of each shall be intermixed to avoid any unnecessary repetition. A mathematical treatment of the underlying transfer function algorithms is given first.

A transfer function normally computes over a scalar input to produce a scalar output, although this need not be the case. In the field-based object model, field attributes are commonly grouped so that colour fields are defined together, reflection coefficients are defined together and so forth. Similarly, in volume visualisation, if a data element has one value then all visual attributes are instantiated with one set of properties, such as a non-shiny skin colour for CT data, and another set of properties, such as a shiny bone colour, for a different data element value. It follows from this that the API should allow concurrent transfer functions for convenience, which compute over a scalar input but yield a k-vector output. Each k output scalar value can be assigned to any volume object field, or could simply be ignored, as will be shown presently.

Assume that a transfer function compute over a series of n pairs:

$$(a_1, \langle b_{1,1}, \ldots, b_{1,k} \rangle), (a_2, \langle b_{2,1}, \ldots, b_{2,k} \rangle), \ldots, (a_n, \langle b_{n,1}, \ldots, b_{n,k} \rangle) \in \mathbb{R} \times \mathbb{R}^k$$

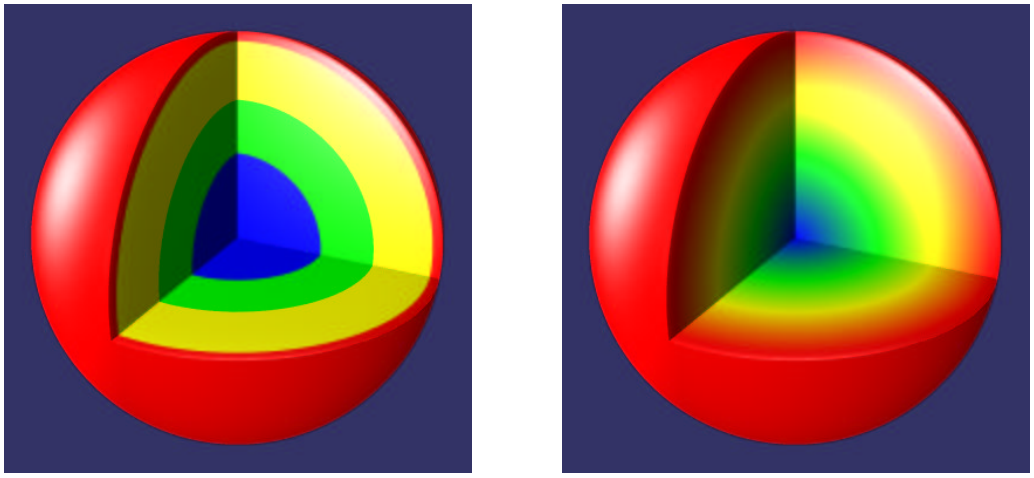
where the pairs are ordered such that $a_1 < a_2 < \ldots < a_n$ and in each pair, the scalar value, a_i, represents the transfer function input and the k-vector, $b_{i,1}, \ldots, b_{i,k}$, represents the transfer function

output. The LR transfer function approximates at most an $(n-1)^{\text{th}}$ degree polynomial using piece-wise linear segments (or ramps, as they are sometimes known), where n is the number of pairs and each pair defines a knot on the approximated curve. The j^{th} component of a k-vector is computed by an LR transfer function as follows:

$$\Phi_{LR}(j,v) = \begin{cases} b_{1,j} & \text{if } v \leq a_1 \\ b_{i-1,j} + \dfrac{v - a_{i-1}}{a_i - a_{i-1}}(b_{i,j} - b_{i-1,j}) & \text{if } \exists i \in \{2,\ldots,n\} \text{ s.t. } a_{i-1} < v \leq a_i \\ b_{n,j} & \text{otherwise.} \end{cases}$$

An LUT transfer function uses the scalar values, a_i, to form $n+1$ ranges of scalar data: $[-\infty, a_1]$, $(a_1, a_2]$, ..., $(a_{n-1}, a_n]$ and $(a_n, \infty]$. It locates the range that contains the input scalar value and returns the corresponding output vector. The output vector for the range $(a_n, \infty]$ is defined implicitly and contains only zeros — this is done for neatness and mathematical simplicity. The j^{th} component of a k-vector is computed by an LUT transfer function as follows:

$$\Phi_{LUT}(j,v) = \begin{cases} b_{1,j} & \text{if } v \leq a_1 \\ b_{i,j} & \text{if } \exists i \in \{2,\ldots,n\} \text{ such that } a_{i-1} < v \leq a_i \\ 0 & \text{otherwise.} \end{cases}$$



(a) (b)

Figure 8.5: Colours defined using (a) lookup table and (b) linear ramp transfer functions.

An example of each of these transfer functions is given in Figure 8.5. A sphere object defined from a spherical distance field is shown with a portion cut away to reveal its internal structures. An LUT transfer function was used to instantiate the colour fields in the left image, where ranges of values on the distance field were mapped to one of four colours (red, yellow, green and blue). In contrast, a LR transfer function was used to instantiate the colour fields in the right image, where the distance field values were used as interpolants to a piece-wise linear spline defined using four colour nodes.

It is important to note that the union of the ranges formed by the pairs in the case of the LUT transfer function is the continuous range $[-\infty, \infty]$. We could have designed the LUT transfer function in such a way that the user must specify ranges explicitly using a pair of input values. For example, the i^{th} range, $[a_{i,1}, a_{i,2}]$, could be defined by the pair $(a_{i,1}, a_{i,2})$. However, if we did this then it would no longer be possible to ensure continuity between two adjacent, non-overlapping ranges. Consider the ranges $[x, y]$ and $[y + \epsilon, z]$, where ϵ is very small. They approximate the continuous range, $[x, z]$, however the value $y + \frac{\epsilon}{2}$ is in neither range and in the context of an LUT transfer function would consequently have no defined output vector.

The pairs over which the transfer functions compute is, in terms of the API, defined as a linear array of $n \times (k+1)$ elements of the `vlfloat` data type. It is not essential that the user sorts every $(k+1)^{\text{th}}$ element since this can be performed by the implementation of the API. A transfer function of either type (LR or LUT) is defined using the Scope 4 command:

`void` **vlMapping**(`vlint` *id*, `vlenum` *type*, `vlint` *k*, `vlint` *n*, `vlfloat` **data*);

where *id* is a unique integer by which the new transfer function will be subsequently referred. *type* must be either `VL_LOOKUP` or `VL_RAMP` to indicate that the user wishes to create either an LUT transfer function or an LR transfer function respectively. *data* is a pointer to the linear array of pairs. An implementation of the API should verify that every $k + 1^{\text{th}}$ value in the array is unique, and should report the `ERR_BADPARAMS` error if a duplicate entry is found.

The Scope 6 command:

`void` **vlMap**(`vlint` *fields*, `vlint` *field*, `vlint` *id*, `vlint` *j*);

passes the value on the field referenced by *field* through the transfer function with identifier *id*, and assigns the j^{th} component of the output vector to all fields referenced by *fields*. It should be noted that exactly one field must be referenced by the second parameter; if multiple fields are referenced then the `ERR_BADPARAMS` error will be reported.

8.6.4 Image-Swept Volumes

In Chapter 5, a novel approach to modelling sweeps in the volume domain, entitled *image-swept volumes*, was presented. An ISV models one or more attribute fields of a volume object as a greyscale image template swept along an arbitrary trajectory in 3D space, and sometimes supports various template transformations including scaling and rotation. Extensions to the basic ISV method are possible such as using 3D volumetric templates in place of 2D image templates, which can lead to some interesting effects. *vlib* provides native support for image-swept volume modelling.

The rotational sweep is particularly common in graphical modelling for synthesising axial-symmetric objects that would in reality be produced on a potter's wheel or a turning lathe. In fact, rotational sweeps are so common that a Scope 6 command:

`void` **vlLathe**(`vlint` *fields*, `vlint` *id*, `vlfloat` *gap*, `vlbool` *normalise*);

has been included in the API specification specifically for modelling them. *id* is the identifier associated with a discrete template loaded with the `vlLoad` command. *gap* is the distance in object space between the right-hand edge of the discrete template and the axis of rotation. If *normalise* is `VL_TRUE` then all elements in the template will be normalised, either to the range $[0, 1]$, or to the minimum and

maximum values specified in the template data file if such values exist. A template will be rotated clockwise around the vertical axis passing through the object's centre.

A variety of other sweeps are possible but are slightly more complicated to specify. First, a sweeping trajectory must be defined using the Scope 4 command:

```
void vlTrajectory( vlint id, vlenum type, vlfloat *data,
   void (*scale)(vlfloat [2] , vlfloat ), void (*up)(vlvector , vlfloat ) );
```

where *id* is a unique integer by which the trajectory is subsequently referred. *type* is a constant referencing the type of trajectory one is defining. An implementation can support any number of trajectories including line segments, circles, quadratic and cubic Bézier curves, B-splines and so on. These types should be referenced using suitable symbolic constants, such as

$$\{\texttt{VL\_LINE}, \texttt{VL\_CIRCLE}, \texttt{VL\_BEZIER2}, \texttt{VL\_BEZIER3}, \texttt{VL\_BSPLINE}, \ldots\}$$

and should be described to the user through suitable documentation. *data* is an array of parameters defining the trajectory. In the case of a line segment for instance, it would contain six values representing the line's start and end coordinates. Again, the intended contents of the data array should be made clear to the user through documentation. *scale* and *up* are pointers to user-defined functions of the form:

```
void scale( vlfloat S[2], vlfloat t );
```

and

```
void up( vlvector U, vlfloat t );
```

which will be executed at every sampling point within the bounds of an image-swept volume. In both cases, $t \in [0, 1]$ is a parameter indicating the distance travelled along the trajectory, computed and passed to the functions by the system. In the case of the *scale* function, *S* is an array of two elements which the user sets to define the horizontal and vertical scale of the template at distance t along the trajectory. If *scale* is assigned a null value or if *S* is not set then it will default to 0.5 in both dimensions. In the case of the *up* function, *U* is a vector which the user sets to define the up vector of the template at distance *t* along the trajectory. If *up* is assigned a null value or if *U* is not set then the up vector will revert to a default up vector.

A volume object's attribute fields can be defined by sweeping a template along a trajectory defined by vlTrajectory with the Scope 6 command:

```
void vlSweep( vlint fields, vlint id, vlint t_id, vlbool normalise );
```

where *id* is the identifier associated with the discrete template and *t_id* is the identifier of a trajectory defined with vlTrajectory. If *normalise* is VL_TRUE then all elements in the template will be normalised, as described above for vlLathe.

8.6.5 Projective Texture Mapping

In Chapter 6, we explained how projective texture mapping can be adapted to the volume domain to define object colour and to support hypertexturing, displacement mapping and bump mapping. It does

```
01 void opacity(vlvector p, void *misc) {
02
03     vlfloat f = vlField(VL_F);
04     if (f >= 300 || (f >= -300 && f <= 50 && p[1] < .5))
05         vlSet(VL_O, 1.);
06 }
07
08 void colours(vlvector p, void *misc) {
09
10     if (vlField(VL_F) < 0) {
11         vlTexture(VL_R, VL_SPHERICAL, 1, VL_TRUE);
12           :
13         return;
14     }
15     vlTexture(VL_R, VL_PLANEZ1, 4, VL_TRUE);
16       :
17 }
```

Figure 8.6: (left) Code applying multiple textures to an object with (right) its output.

this by projecting a greyscale image from either a spherical, cylindrical, cubic or planar intermediate surface bounding the object onto all or part of one or more scalar fields.

A texture map, once loaded with vlLoad, is projected onto a volume object using the Scope 6 command:

`void` **vlTexture**(`vlint` *fields*, `vlenum` *type*, `vlint` *id*, `vlbool` *normalize*);

where *type* indicates the type of intermediate surface and must be one of those listed in Table 8.6. *id* is the identifier assigned to an image map at the time it was loaded. (If a volume dataset is referenced then the first slice of the data along the z-axis will be projected.) If *normalize* is VL_TRUE then the values in the image map will be normalised according to the normalisation parameters specified in the data file, or to the range $[0, 1]$ if no such parameters are available (e.g., in the case of standard bitmap image files). This feature is useful if one intends to project images onto, say, the colour fields of a volume object, whose values are clamped to the unit range.

An example object with colour fields defined using projective texture mapping is shown in Figure 8.6 along with the relevant code. The geometry of the object was defined using the *CThead* dataset (the code for this is not shown). The *opacity* function (shown) was then used to define the opacity field such that the object shows half skull and half skin. Next, the colour fields were defined using the *colours* function. If the geometry field is negative at a given sampling point (line 10) then the point represents skin so the red, green and blue attribute fields at that point were defined by applying a skin texture using spherical texture mapping (lines 15-16). If, however, the geometry field is positive then the point represents bone so a wood texture was applied using planar texture mapping (lines 11-12). This example shows that texture maps can be applied in different ways at different sampling points within the same volume object.

8.6.6 Tiling

Tiling is supported as a convenient method for repeating sequences of values on scalar fields a finite number of times over any of the three spatial dimensions. In this context, tiling is analogous to the

Table 8.6: Texture projection types.

Type	Surface	Type	Surface	Type	Surface
VL_SPHERICAL	Spherical.	VL_PLANEX1	Left face.	VL_PLANEX2	Right face.
VL_CYLINDRICAL	Cylindrical.	VL_PLANEY1	Bottom face.	VL_PLANEY2	Top face.
VL_BOX	Box.	VL_PLANEZ1	Front face.	VL_PLANEZ2	Back face.

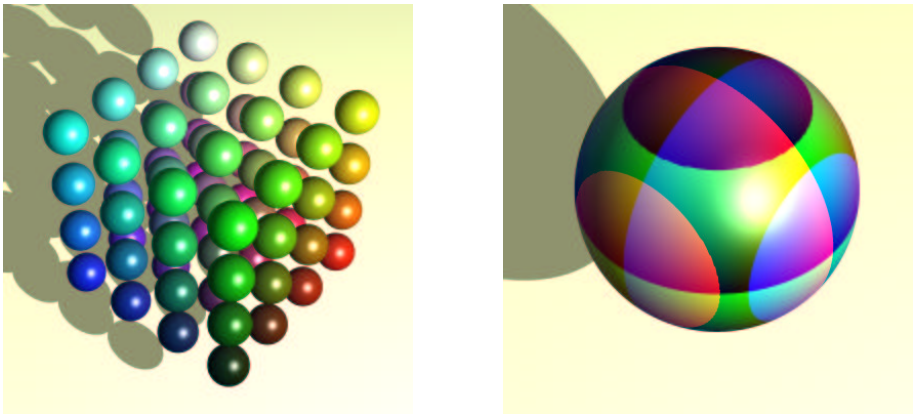
way that tiles on a wall tessellate to repeat a given pattern and should not be confused with *surface tiling* — the process in which cells are fitted with polygons during iso-surface extraction. A scalar field can be tiled at any point during the specification of a volume object with the Scope 6 command:

`void` **vlTile**(`vlint` *fields*, `vlfloat` t_x, `vlfloat` t_y, `vlfloat` t_z);

where t_x, t_y and t_z are positive values indicating the number of times that the values on a field should repeat in the x, y and z dimensions respectively. vlTile is not a "core" field modelling command since it cannot be used to define scalar fields without the use of other field modelling commands such as vlSet or vlDataset. However, it will modify all attribute fields, A, referenced by the first argument such that

$$A(x, y, z) = A(x\,t_x - \lfloor x\,t_x \rfloor, y\,t_y - \lfloor y\,t_y \rfloor, z\,t_z - \lfloor z\,t_z \rfloor).$$

If, for example, vlTile is called three times successively with tiling parameters $(2, 2, 2)$ then the same result will be achieved as if it were called once with tiling parameters $(8, 8, 8)$. There is no limit to the number of times that vlTile can be called.



(a) (b)

Figure 8.7: A sphere object where in (a) the geometry is tiled and in (b) the colours are tiled.

A sphere with continuous colour was used to define the objects shown in Figure 8.7. In the left image, the object's geometry field was tiled four times in each dimension, resulting in 64 spheres, all of which have a unique colour. In the right image, the geometry field keeps its original specification while discontinuous colour was achieved by tiling the three colour fields four times in each dimension. The vlTile command is particularly useful for tiling repeatable textures.

8.6.7 Field Functions

The functionality presented in the previous two sections clearly supports the two-part volume visualisation strategy described earlier. An object's geometry field (VL_F) for example can be defined using a discrete volume dataset with vlDataset before its opacity (VL_O) and colour (VL_R, VL_G, VL_B) fields are defined by passing the geometry field through a transfer function with vlMap. Although this functionality is sufficient in many volume visualisation applications, it has limited use in general-purpose computer graphics. Importantly, three primary deficiencies would result if we were to supply this functionality only. The user would be unable to:

- *compute over sampling coordinates*; this prohibits the emulation of a variety of important object representations, such as implicit surfaces, implicit solids and analytical distance fields, as well as the creation of procedural solid textures.
- *compute arbitrarily over scalar fields*; the built-in transfer functions provide a limited solution to this problem, although it may be the case that the user requires a custom transfer function.
- *apply datasets or transfer functions to only part of a scalar field*; that is, all scalar fields would be homogeneous with respect to their sources.

Field functions alleviate all of these problems. A field function is essentially a user-defined callback function which is passed manually to the API at modelling-time using an interface command. Then at render-time, the rendering engine executes the function at all appropriate sampling points and instantiates scalar fields accordingly. An example of a field function is shown in Figure 8.8; the sample code demonstrates that a simple texture can be defined procedurally by computing over the coordinates of a sampling point.

A field function has the form:

`void` **f**(`vlvector` *p*, `void` **misc*);

where *p* holds the coordinate of the sampling point in object space, and is computed and passed by the system. The user is allowed to change *p*, but any changes made will affect only those scalar field modelling commands called from the field function. The *misc* parameter enables the user to pass additional arbitrary information to the function which may be required for computation, such as an array of additional function parameters. If used then *misc* should be typecast from `void` to the appropriate type.

The Scope 6 command:

```
01 f(vlvector p, void *misc) {
02
03    if (p[0] < 0.5)
04        vlSet(VL_B, 1.);
05    else
06        vlSet(VL_B, 0.);
07
08    vlSet(VL_R, p[1]);
09 }
```

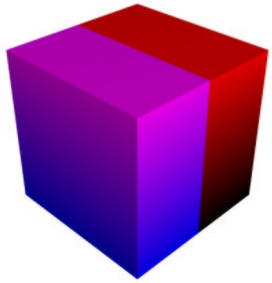


Figure 8.8: A simple texture defined by computing over a point.

`void` **vlFunction**(`void` (*f)(`vlvector` , `void` *), `void` *misc*);

passes a field function to the API, where *misc* is user-defined data which the system stores and passes to the field function at render-time. If no user-defined data needs to be passed then *misc* would typically have a null value. The API could have been designed in such a way that field functions are parameterised not only by the sampling coordinates and user-defined information, but also by various other potentially-useful data such as the values on all scalar fields at the time when the field function is called. The decision not to take this design step was motivated by a variety of factors. In particular, an implementation of the API, such as our own, can be developed in such a way that fields are not evaluated if they are not explicitly utilised in the specification of a volume object, so as not to waste processing time with redundant computation. If all fields must be evaluated and the computed values passed as parameters to a field function then processing time may potentially increase. In addition, parameter passing is particularly slow so to minimise the number of arguments to a field function, additional record data types would be required thereby sacrificing the inherent simplicity of the API.

A few interface commands are supplied to allow the user to access certain information about volume objects from field functions at render-time. The value on any of the object's scalar fields, for instance, can be accessed and consequently computed over, with the Scope 10 command:

`vlfloat` **vlField**(`vlint` *field*);

It should be noted that exactly one scalar field must be referenced by the *field* parameter. If no fields or multiple fields are referenced then the ERR_BADPARAMS error will be reported. A number of other similar commands are also available and will be discussed in subsequent sections in the context of their purposes.

A number of effects implementable using field functions are illustrated in Figure 8.9. They are:

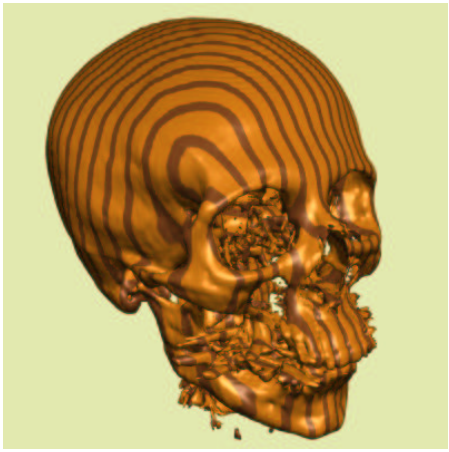
- *Image (a)*– A solid wood texture defined by computing over the sampling coordinates.
- *Image (b)*– A blend of two textures. To create this image, a wood texture was first applied to the colour fields of an object, then the colour values were extracted from the fields using vlField and stored in variables. A checker texture was then applied and the colour values were extracted similarly and stored in a different set of variables. The stored colour values were then averaged and written back to the colour fields using vlSet.
- *Image (c)*– A smoke object in which the opacity field is defined by adding turbulence to a cylindrical distance field.
- *Image (d)*– An implicit surface created by rendering the iso-surface (in spherical coordinates):

$$\sin(3\theta)\sin(4\phi) - r = 0.$$

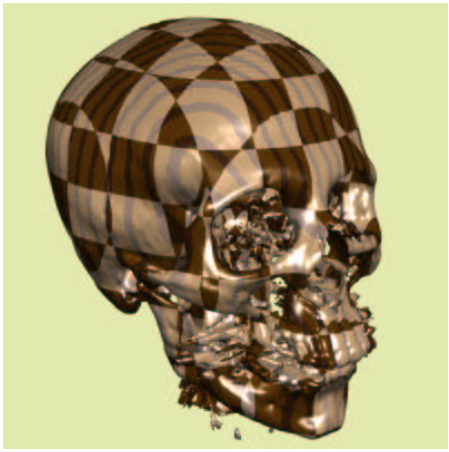
- *Image (e)*– An object in which a dataset is used to define geometry only at selected regions.

8.6.8 Scene Graphs

A *scene graph* provides an explicit means of "combining" multiple volume objects to form compound objects by applying real-domain operators to their constituent scalar fields, such as those described in Section 3.4. The arbitrariness of constructive operators in the volume domain prevents us from

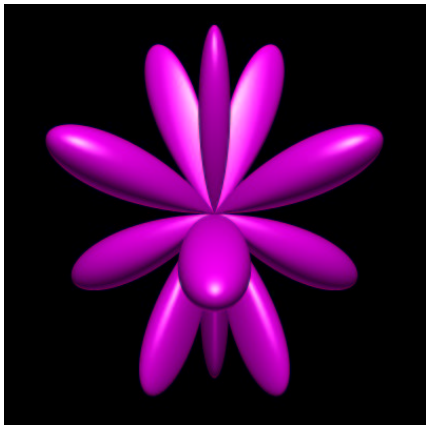


(a) Solid texture

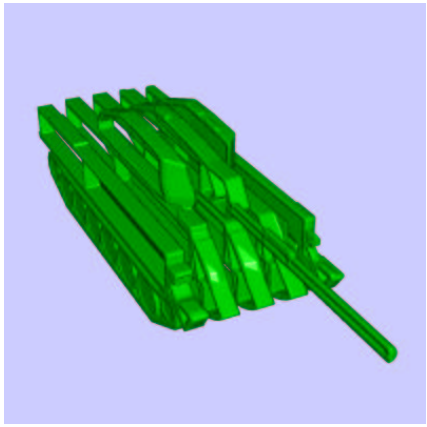


(b) Texture blend

(c) Hypertexture



(d) Implicit surface



(e) Partial dataset

Figure 8.9: Important effects can be implemented using field functions.

allowing objects defined in the API to overlap unless a constructive operator is explicitly applied to their fields. This is due to the confusion arising when, for example, two opacity values or two sets of colours are associated with a single sampling point; it is not known whether the values should be added, averaged, one should be discarded, or an entirely new heuristic should be employed. A consequence of multiple objects implicitly intersecting therefore, is to allow an unpredictable image to result.

No stringent rules are enforced governing precisely how constructive operators should combine scalar fields. If just a few operators, such as union, intersection, difference and blend [CT00] are hard coded or are defined as concrete types within the API then it would be difficult to add new operators as and when they are required. Instead, the API allows its users to create their own constructive operators according to the needs of their specific modelling applications or virtual environments. An operator is implemented as a field function (Section 8.6.7) which allows the user to compute arbitrarily over

```
01 vlObject(); /* obj A */
02     vlScale(100., 100., 100.);
03
04     vlObject(); /* obj B */
05         vlTranslate(0., 0., 1.);
06         vlFunction(VL_R, &f1, 0);
07
08         vlObject(); /* obj C */
09             :
10         vlEnd();
11     vlEnd();
12
13     vlObject(); /* obj D */
14         :
15     vlEnd();
16
17     vlFunction(VL_G, &f2, 0);
18         :
19 vlEnd();
```

Figure 8.10: An object hierarchy.

individual scalar fields, the sampling location or any additional information which the user deems necessary. In this case, the user would also need to compute over separate volume objects to create a compound object, or a *parent object*. The parent object would be represented by a non-leaf node in a scene graph. Although it was not stated previously, this is permitted within the API, but only to a limited degree.

The value on any attribute field of a volume object, $\mathcal{O}_P$, can be computed as a function of one or more attribute field values from another volume object, $\mathcal{O}_C$, but only if $\mathcal{O}_C$ is explicitly declared as a *child* of $\mathcal{O}_P$. $\mathcal{O}_C$ is a child of $\mathcal{O}_P$ if it is an immediate descendent of $\mathcal{O}_P$ in the scene graph. If $\mathcal{O}_C$ is a child of $\mathcal{O}_P$ then $\mathcal{O}_P$ is the *parent* of $\mathcal{O}_C$. An object can have any number of children but only one parent. In *vlib*, a hierarchy of parent and child objects is formed by recursively nesting the declaration of one or more volume objects inside the declaration of another, as illustrated in Figure 8.10. In the example code, A is the parent of B and D, and B is the parent of C. The commands specified in the remainder of this section allow parent objects to establish important information about their children, such as the number of children and the value on a child field at any given sampling point. The children are indexed by the order in which they are declared inside the parent. In the example code therefore, B will be child 1, D will be child 2 and C will be child 1 (since it is the first of B's children).

A facility is provided to allow constructive operators to determine the number of child objects present, and should always be used. If an operator is designed to compute over a fixed number of children then it should verify that the correct number of children are present or report some error otherwise. If, on the other hand, an operator computes over an arbitrary number of children then the total number present would need to be determined. The number of child objects nested inside a parent is determined with the Scope 10 command:

`vlint` **vlChildCount**`( void );`

from a render-time function owned by the parent object. If no children are present then the object is not a parent object and this command will return zero. Another important detail for consideration is whether or not the child objects contain valid data at the current sampling point. Given that all objects are bounded by a finite region of Euclidean space, it may happen that a sampling point does

```
01 void blobby(vlvector p, void *misc) {
02
03     vlfloat sum = 0.;
04     for (i = 0; i < vlChildCount(); i++) {
05         if (vlChildHit(i))
06             sum += blend(ChildField(VL_F, i));
07     }
08
09     vlSet(VL_F, sum);
10 }
```

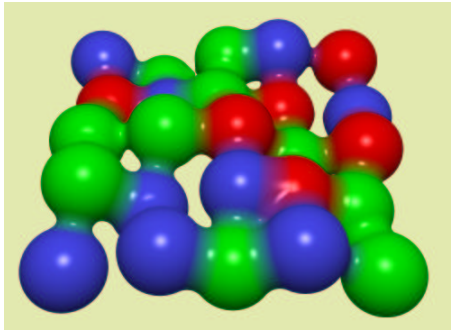


Figure 8.11: (left) An operator computing over an arbitrary number of children with (right) its output.

not lie inside one or more of the children, in which case those children will contain undefined values on all their fields at that point. The user can verify that the current sampling point lies inside child i with the Scope 10 command:

`vlbool` **vlChildHit**(`vlint` *i*);

If child i is known to exist and the current sampling point lies within its bounds, then the value on any of child i's scalar fields can be determined with the Scope 10 command:

`vlfloat` **vlChildField**(`vlint` *field*, `vlint` *i*);

where *field* is a reference to a single scalar field.

A "blobby" constructive operator is shown in Figure 8.11. It computes over an arbitrary number of children (line 4), and for each child, it first determines if the current sampling point is inside the child's bounding box (line 5) and if so, it passes the child's geometry value to a blending function (line 6) and adds the result to a running total. Finally, the resulting total is assigned to the geometry field of the parent object (line 9). An image rendered using this blobby operator is shown alongside the code. A similar constructive operator was used to blend the colours.

8.7 Normal Estimation

Accurate normal vector determination is arguably the most important part of the rendering process. If poor normal vectors are determined from the geometry field of a volume object then not only will diffuse and specular shading be displeasing to the eye, but also reflection and refraction effects will be noticeably wrong. A number of methods have been determined for estimating accurate normals from discrete volume datasets, some of which are known to produce very good results (Section 2.4.2). In our field-based modelling framework however, a volume object's geometry field, $\mathbf{f}$, is not necessarily defined using discrete volume data, and therefore these methods may not always be suitable. In some instances, it may be more appropriate to borrow normal estimation techniques from the implicit surface domain instead (e.g., that described in [Har93]), particularly if a volume object's geometry field is based upon a continuous mathematical specification. If constructive operators are created to mix continuous and discrete geometry data arbitrarily then it may be the case that no suitable methods apply.

The commands included in this specification make normal estimation difficult due to the fact that a volume object's geometry field, upon which the normal estimation is based, can be instantiated:

- many times during a single object specification; and
- using any one of the six core field modelling commands.

The difficulty faced by the first of these points is described using a simple example: Assume two medical CT datasets exist, A and B, where A is an original CT scan and B is a simple volume where each voxel is a label identifying the feature in the corresponding voxel in A. A visualisation of the patient data could be produced using the following sequence of actions:

1. The geometry field, $\mathbf{f}$, is defined by applying a reconstruction filter to dataset B.
2. The colour and opacity fields are then defined by passing $\mathbf{f}$ through a transfer function.
3. The geometry field is re-defined by applying a reconstruction filter to dataset A.
4. Lastly, a normal vector is computed from $\mathbf{f}$.

In this example, there is no need to estimate a normal vector before the geometry field has been instantiated for the second time and so it would be inefficient for a rendering engine to compute a normal vector automatically each time the geometry field is defined. However, one cannot simply assume that a normal vector should be estimated only once the geometry field has been defined for the final time, and not before. Using datasets A and B again, a visualisation could be defined whereby the opacity field is modelled as a function of gradient magnitude (e.g., as in [Lev88]), using the following sequence of steps:

1. The geometry field, $\mathbf{f}$, is defined by applying a reconstruction filter to dataset A.
2. A normal vector is computed from $\mathbf{f}$.
3. The colour and opacity fields are defined as a function of gradient magnitude.
4. The geometry field is re-defined by applying a reconstruction filter to dataset B.
5. A normal vector is computed from the newly-instantiated $\mathbf{f}$.

For this example, the rendering engine must estimate the normal vector twice. A highly efficient implementation of *vlib* will be able to detect precisely when a normal vector must be calculated (see Section 9.5) and could consequently reduce the rendering cost, although there might always be some computational overhead. To reduce this problem, the Scope 3 command:

```
void vlEstimateNormal( void );
```

has been incorporated into the specification. It forces the rendering engine to compute a normal vector immediately based on the current values on the geometry field at the current time. If the geometry field has not been defined at the time this command is called then it will compute the zero vector.

A *vlib* implementation should choose its normal estimation algorithm according to the field modelling command used to define the geometry field before the call to vlEstimateNormal. If the VL_DATASETNORMALS toggle option is turned off (see Section 8.10.2) or the implementation does not support such context-sensitive normal estimation then a normal vector must be determined using

central differencing with the geometry field in world space, where the neighbourhood size (the interval between the primary and secondary sampling points in each dimension) is set using the Scope 2 command:

`void` **vlNeighborhood**(`vlfloat` δ_x, `vlfloat` δ_y, `vlfloat` δ_z);

The user should ensure that the neighbourhood size should be approximately $\sqrt{3}$ times the length of the interval between sampling points along a ray in each dimension to avoid aliasing artefacts and self-occlusion problems (Section 9.4.1.1), although smaller neighbourhood sizes can be used with some object configurations and lead to better images. A secondary point is determined by re-evaluating a volume object up to the point at which the appropriate vlEstimateNormal command, which invoked the normal estimation routine, is called. To avoid infinitely recursive normal estimation, it has been necessary to prevent vlEstimateNormal from computing a normal vector at secondary sampling points. Any calls to this command while the renderer is processing a secondary sampling point will simply result in a zero normal vector being set.

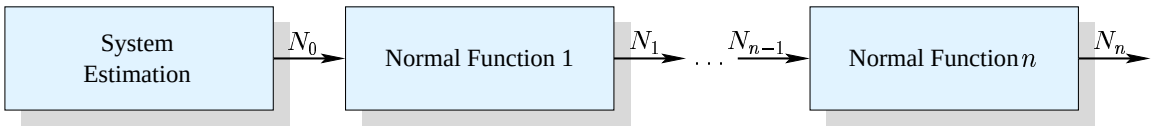


Figure 8.12: A normal vector can be perturbed by a chain of normal functions.

A normal vector can be defined procedurally or analytically in a user-defined normal function (similar to the field function described in Section 8.6.7) of the form:

`void` **nf**(`vlvector` *N*, `vlvector` *p*, `void` **misc*);

where *p* is the current sampling position, computed and passed to the function by the system and *N* is the normal vector to be defined by the user. If a normal vector has already been computed for a volume object then it shall be passed into the normal function as parameter *N* by the system and can be computed over or perturbed to achieve bump mapping. Hence, a chain of normal functions can be established through which normal vectors are propagated, as illustrated in Figure 8.12. If no normal vector has yet been computed then the zero vector will be passed. If the user changes *p* from within the normal function then the change will be discarded when the normal function completes. *misc* is a pointer to user-defined data, as described in Section 8.6.7.

The Scope 3 command:

`void` **vlNormal**(`void` (**nf*)(`vlvector` *N*, `vlvector` *p*, `void` *), `void` **misc*);

associates a normal function with an object specification, in much the same way as vlFunction, discussed earlier, associates a field function with an object specification. It is illegal to instantiate any of the volume object's fields from a normal function since a field function exists for this purpose. However, it is possible to evaluate the object's fields and its children's fields from a normal function with the vlField and vlChildField commands respectively.

A volume object's normal vector can be accessed at any time from other render-time functions with the Scope 12 command:

`void` **vlGetNormal**(`vlvector` *N*);

If no normal vector has yet been computed then the zero vector will be returned. The purpose of this command is to enable visual attributes to be defined as a function of the gradient direction and magnitude, as is commonly done in volume visualisation. The last normal vector to be computed for any of the volume object's children can be determined using the Scope 10 command:

`void` **vlChildNormal**(`vlvector` *N*, `vlint` *i*);

where N is the normal vector of child i, transformed relative to the orientation and scale of its parent. If the current sampling point does not fall inside child i then a undefined normal vector will be returned; the user should always ensure that a valid child number is passed and a valid normal vector is computed by the system with the vlChildCount and vlChildHit commands respectively.

In some instances, volume datasets, especially segmented data and distance fields, are defined such that their normal vectors point inwards, and not outwards from surfaces as one may expect. The Scope 11 command:

`void` **vlFlipNormal**(`void`);

is included in the *vlib* API specification to alleviate this problem by allowing the user to flip a computed normal vector around.

8.8 Transformation

In the field-based object model, all scalar fields comprising a volume object are bounded by the unit domain, $[0,1]^3$. This notion led to the concept of *object space* and *world space* frames of reference in the modelling framework. A transformation matrix, $\mathcal{T}$, is included in the object model to enable multiple volume objects to be scaled and positioned independently in world space.

An object is automatically assigned the identity transformation matrix when it is defined in *vlib*. If children are nested inside the object then $\mathcal{T}$ is multiplied by the transformation matrix:

$$\begin{pmatrix} x_2 - x_1 & 0 & 0 & x_1 \\ 0 & y_2 - y_1 & 0 & y_1 \\ 0 & 0 & z_2 - z_1 & z_1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where x_1 and x_2 are the extrema of the child objects along the world's x-axis, and similarly for y_1, y_2, z_1 and z_2. The purpose of this post-multiplication is to ensure that the parent object always bounds its children regardless of any transformations which these children may undergo. $\mathcal{T}$ can be manipulated further using a series of API commands which:

- scale the object relative to the world's origin:

  `void` **vlScale**(`vlfloat` *x*, `vlfloat` *y*, `vlfloat` *z*);

- rotate the object about the world's origin:

  `void` **vlRotate**(`vlfloat` *x*, `vlfloat` *y*, `vlfloat` *z*);

- translate the object through world space:

  `void` **vlTranslate**(`vlfloat` *x*, `vlfloat` *y*, `vlfloat` *z*);

- flip the object about one or more axis:

  `void` **vlFlipAxis**(`vlbool` x, `vlbool` y, `vlbool` z);

- flip the object about one or more planes:

  `void` **vlFlipPlane**(`vlbool` x, `vlbool` y, `vlbool` z);

All of these are Scope 3 commands. If any of the arguments to vlScale are non-positive then the ERR_BADPARAMS error will be reported. If more than one argument to vlRotate is non-zero (where the arguments are specified in degrees) then the object will be rotated about the x, then the y and lastly the z axis in world space. If more than one argument to vlFlipPlane is VL_TRUE then the object will be mirrored about the yz-plane, then the xz-plane and lastly the xy-plane. If more than one argument to the vlFlipAxis is VL_TRUE then the object will be mirrored about the x, then the y and lastly the z axis.

There is no limit to the number of times that these commands can be called, although the user should ensure that they are called in the correct order given that matrix multiplication is not commutative. If a transformation command is called from a parent object block then the requested transformation will be propagated down recursively through all nested child object blocks. This is so that if a parent object is scaled or moved then its child objects will be scaled or moved accordingly.

8.9 Deformation

In graphical modelling, particularly in animation, deformation is necessary for making objects appear as though they obey the physical laws of motion. If a rubber object falls then it should squash and bounce. If a brittle object falls then it should crack or shatter. A modelling system should make every attempt to support arbitrary object deformation in order to facilitate these and other effects seen in the real world. A number of deformation techniques have been described at various stages throughout this thesis including morphing and displacement mapping. A few are supported natively b the API which we now show.

8.9.1 Spatial Transfer Functions

In Chapter 4, *spatial transfer functions* were introduced with the field-based object model as a method for deforming objects in a procedural manner with a minimal memory overhead. The method described deforms volume objects, which are modelled in the conventional manner, at render-time, therefore avoiding the need for resampling which is known to cause quality degeneration.

A spatial transfer function is implemented in the API as a user-defined function of the form:

`void` **stf**(`vlvector` *p*, `void` **misc*);

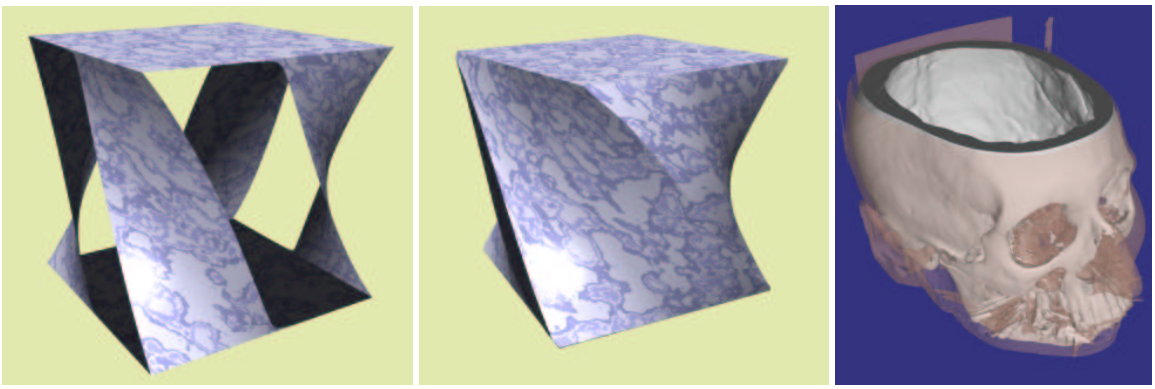
where *p* is the coordinate of the sampling point in object space, computed and passed to the function by the system. A spatial transfer function differs from a field function in two ways. First, attribute

fields cannot be defined using the field modelling commands from within a spatial transfer function, although the current values on these fields can be examined using vlField. Second, a spatial transfer function allows permanent changes to be made to the sampling point, *p*. If the coordinate stored in *p* is changed to *p'* then the rendering engine will evaluate all attribute fields at *p'* but will display the computed properties at *p*. If the new coordinate is outside the unit domain then a sample will not be taken and a blank voxel will be drawn. *misc* is a pointer to user-defined data, as described in Section 8.6.7.

The Scope 3 command:

`void` **vlSTF**(`void` (**stf*)(`vlvector` *p*, `void` *), `void` **misc*);

associates a spatial transfer function with a volume object, where *misc* is user-defined data stored by the system ready for passing to the function at render-time. If the function referenced by *stf* does not compute over any user-defined data then any value could be passed. vlSTF can be called any number of times from an object block and the corresponding spatial transfer functions will be evaluated in order. All field modelling commands will be executed fully before the first spatial transfer function is evaluated.



(a) (b) (c)

Figure 8.13: Clipping can occur with some spatial transfer functions.

A poorly specified spatial transfer function could result in an undesired clipping effect, as shown in Figure 8.13(a), whereby the sampling coordinate, *p*, is changed to one outside the boundary of the volume object. A convenient way to alleviate this problem is to scale the boundary of the volume object without scaling the volume object itself. The Scope 3 command:

`void` **vlBoundary**(`vlfloat` b_x, `vlfloat` b_y, `vlfloat` b_z);

performs this scaling operation, where b_x, b_y and b_z are positive values indicating the boundary scaling factors in the x, y and z dimensions respectively. vlBoundary can also be used to crop volume objects if scaling factors smaller than 1 are specified, as shown in Figure 8.13(c).

8.9.2 Free-Form Deformation

Unfortunately, FFD is at best difficult to render directly in volume graphics without first resampling into a regular grid. However, one possibly approach to overcoming this problem which we described in the previous chapter involves decomposing space into a fixed number of cells and then texture mapping the volume object into these cells at render-time. This FFD technique is supported natively by the API through the Scope 3 command:

`void` **vlFreeForm**(`vlvector` *controls[4][4][4]*, `vlint` *cells*);

where *controls* is a 3D array of 64 control points defining a tri-cubic Bézier volume onto which the volume object is texture mapped. *cells* is a positive integer indicating the number of cells in each dimension into which the Bézier volume should be decomposed. An object, modelled in the usual way, is deformed by issuing a call to vlFreeForm from its object block. There is no limit to the number of times that vlFreeForm may be called for a single object.

8.9.3 Hypertextures

A variety of hypertexturing effects and certain kinds of deformation require a sampling point in the *soft region* of an object to be projected directly onto the nearest *hard region*, where information can be computed and used to determine attribute properties of the sampling point [PH89]. In Chapter 6, we showed how projection can be used in conjunction with projective texture mapping to provide greater control over hair hypertextures and displacement effects. The Scope 5 command:

`vlbool` **vlProject**(`vlvector` *p*);

projects the current sampling point along the reverse direction of the computed normal vector to the closest defined iso-surface. It can only be called from the field function of a top-level parent object given that iso-surfaces cannot be defined for child objects (this will be discussed shortly). The system, once it has projected the point, sets *p* to be the coordinate of the point at the iso-surface. Iso-surfaces are defined using the vlSurface command. It is essential to ensure that a normal vector has been estimated (which in turn will involve defining the object's geometry field) otherwise the system will not know where to project the sampling point. In the event that vlProject is called before a normal is estimated, or if a zero-length normal vector has been computed, then the command will return VL_FALSE to indicate that *p* contains an invalid coordinate due to a failed projection. No error is generated in such instances since zero-length normals are common, particularly in amorphous phenomena. VL_FALSE will also be returned if no iso-surface is located along the reverse normal direction.

A typical use of vlProject is illustrated in Figure 8.14. The code shows how a hair hypertexture is applied to an implicit surface. If the sampling point is on or inside the surface of the implicit surface (line 3) then it is made opaque and assigned a pre-determined colour. If it is outside but within some arbitrary hair length's distance of the surface then it is projected along the reverse normal direction until it intersects with an iso-surface (line 11). If an intersection is determined then a Boolean decision function is evaluated to decide whether p lines on a hair root (line 13). If so then the current sampling point is given opacity (line 15) and is coloured according to a function of p (not shown in the code). It is essential to note that although the variable holding the coordinate of the sampling point is changed

```
01 void hair(vlvector p, void *misc) {
02
03     if (vlField(VL_F) > 0.) {
04
05         vlSet(VL_O, 1.);
06         vlSet(VL_KS, 0.9);
07           :
08         return;
09     }
10
11     if (vlField(VL_F) > -8. && vlProject(p))
12
13         if (IsHair(p)) {
14
15             vlSet(VL_O, 0.8);
16               :
17         }
18 }
```

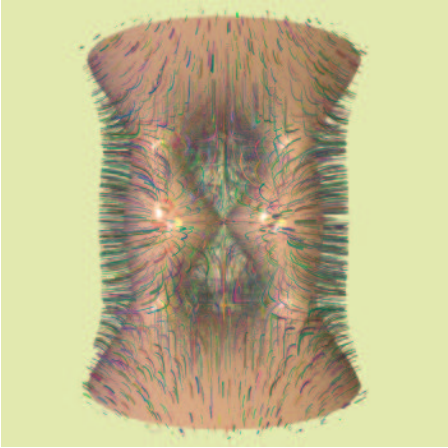


Figure 8.14: Example showing how projection can create hypertextures.

by the vlProject command, all core field modelling commands effect the attribute fields at the original sampling location.

8.10 Rendering Parameters

While developing a rendering engine for volume graphics we have been able to identify a number of rendering features that are not, and should not be, controlled by the field-based object model. In essence, these are features that govern how a rendering algorithm should determine pixel intensities from an instance of the object model, and how it should interpret and process values on a volume object's scalar fields. The object model itself is concerned only with representing an object or the attributes of an object, such as its colour, opacity and reflection coefficients. It is not concerned with, and never should be concerned with, rendering or implementation details of any sort. The geometry field of our concrete object model, for instance, is used to define object curvature. However the precise nature of how pointwise normal vectors are determined from this field is very much dependent on the underlying rendering engine and one of a number of methods could be used. In this section, several commands are presented which affect the way in which volume objects are rendered.

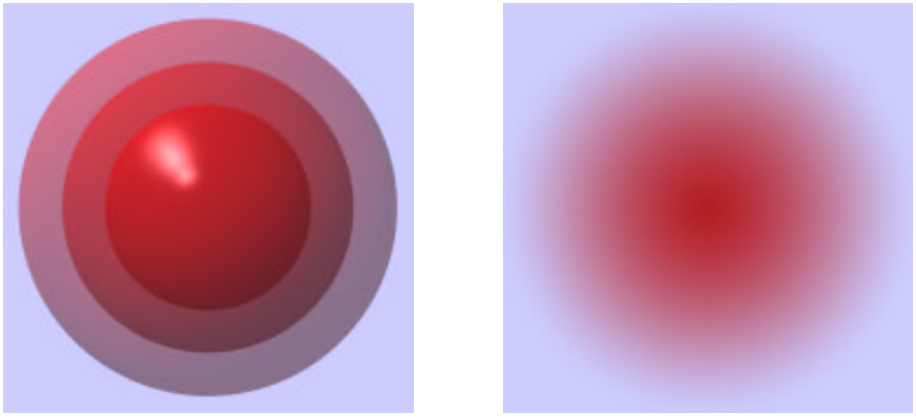
8.10.1 Rendering Algorithms

A number of different volume rendering algorithms are described in the literature, including direct volume rendering [Lev88], direct surface rendering [Jon97] and variants of these such as the maximum intensity projection method [ASK94]. A discussion of these is given in Chapter 2. The DVR, DSR and MIP algorithms are all supported by the API to enable users to render amorphous matter, iso-surfaces or classical X-ray visualisations according to the requirements of their scenes.

vlib applies the DVR algorithm to all objects until the user explicitly selects another algorithm with the Scope 2 command:

`void` **vlRenderType**(`vlenum` *type*);

were *type* is either `VL_DVR`, `VL_DSR` or `VL_MIP`. If a different algorithm is selected then that algorithm will be used to render all objects defined subsequently until the rendering algorithm is changed again, and so on. vlRenderType cannot be called from inside an object block to avoid ambiguities arising when multiple rendering algorithms are selected for a single volume object. The MIP algorithm will render the voxel with the greatest opacity along each ray, for each object. It will only reflect ambient light and will not compute inter-object reflections or refraction.



(a) DSR (b) DVR

Figure 8.15: Effect of rendering a given object with different rendering algorithms.

If vlRenderType is called to request that an object should be rendered with the DSR algorithm then one or more iso-surfaces for that object should be defined. Figure 8.15 shows two images of the same object. The left object is rendered using DSR with three iso-surfaces and the right object is rendered using DVR. Intuitively, if no iso-surfaces are defined then DSR will not render anything. Although it is perfectly legal to apply DSR to an object without defining any iso-surfaces, we have not identified any modelling or rendering benefits of doing this. At any point within an object block, any number of iso-surfaces can be defined by issuing calls to the Scope 3 command:

`void` **vlSurface**(`vlint` *fields*, `vlfloat` τ);

For each iso-surface defined by such a call, the system will render $f(x, y, z) = \tau$, for all fields f referenced by the first argument. It is not legal to call vlSurface from nested (i.e., child) object blocks. If iso-surfaces need to be defined for a compound object then they must be done so from immediately within the top-level parent object. The reason for this is that child objects exist only to partially define a single complex object, so either an iso-surface is defined for that single object or it is not. The API does not allow iso-surfaces to be defined over part of an object either. If one does not choose to render an object using DSR, all defined iso-surfaces will still be used in conjunction with the vlProject command.

Table 8.7: Boolean rendering parameters for the vlToggle command.

Feature	Default	Description
VL_CASTSHADOWS	√	Attenuate light according to the opacity and colour fields, potentially causing shadows to fall on any object with the VL_RECEIVESHADOWS flag set.
VL_DATASETNORMALS	√	If the geometry field has been set with vlDataset then estimate normal vectors by filtering dataset elements, otherwise use object-space central differencing.
VL_FLIPNORMALS		Allow normal vectors to be reversed if they point away from a light source.
VL_RECEIVELIGHT	√	Receive light from light sources, including the ambient light source.
VL_RECEIVESHADOWS		Receive shadows cast by any object with the VL_CASTSHADOWS flag set.
VL_REFLECTBACKGROUND		Integrate the background colour into reflective rays, even if the VL_REFLECTOBJECTS flag is not set.
VL_REFLECTINOBJECTS	√	Appear in the reflections of objects with the VL_REFLECTOBJECTS flag set.
VL_REFLECTOBJECTS		Allow reflective rays to intersect any object with the VL_REFLECTINOBJECTS flag set. The background colour will not be reflected unless the VL_REFLECTBACKGROUND flag is set.
VL_REFRACT		Refract rays according to the index of refraction (IOR) field. The IOR field will have no affect if this flag is not set.
VL_VISIBLE	√	Appear visible to primary rays (rays originating at the camera). An object without this flag set may still appear in reflections if the VL_REFLECTINOBJECTS flag is set.

8.10.2 Toggle Options

The rendering features that we spoke of earlier are generally specified on a per-object basis, that is, they are constant for all points in an object unlike the pointwise information defined by the scalar fields in the field-based object model. An advantage of the way in which objects are represented by the field-based object model and are defined by the API is precisely that rendering features *can* be assigned on an object-wise basis, unlike traditional representations of volumetric scenes in which it is usually impossible to distinguish between different objects, whereby rendering parameters are specified on a far less flexible scene-wise basis.

A number of object-wise rendering parameters that we perceive to be useful are Boolean in the sense that an attribute is either set or unset, or a feature is either on or off. In the most part, these Boolean parameters, or toggle options, exist to allow varying degrees of rendering quality and consequently a change in their configuration could cause the rendering time for a given scene to increase from a few seconds to many minutes, depending on the specification of that scene's constituent objects.

The API maintains a state of Boolean rendering parameters where each parameter is initially set to a carefully-chosen pre-defined value. As each volume object is defined, it inherits a copy of the state, which then becomes fixed for that object. The state can only be changed in Scope 2 by toggling rendering parameters with:

`void` **vlToggle**(`vlint` *params*, `vlbool` *active*);

where *params* is a reference to one or more of the features cited in Table 8.7 and active is `VL_TRUE` to indicate that the referenced features should be turned on or `VL_FALSE` to indicate that they should be turned off. A tick is placed in the centre column in Table 8.7 to indicate that the corresponding feature is active by default. A feature is active by default if it is commonly used in volume rendering but does not slow down the rendering process. If the `VL_CASTSHADOWS` feature is set for an object, for example, then that object will attenuate light from the appropriate light sources. However, shadows will not appear, and hence rendering will not become slow, until the `VL_RECEIVESHADOWS` flag is set for one or more objects, which is off by default.

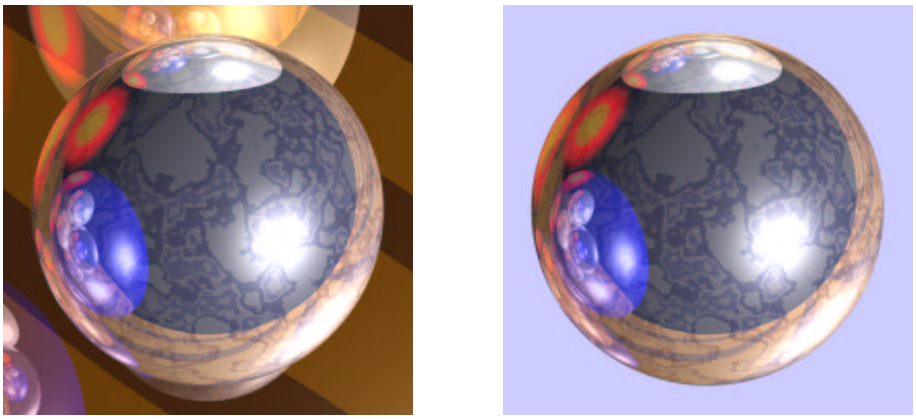
(a) (b)

Figure 8.16: Effect of toggling `VL_DATASETNORMALS` (a) on and (b) off.

A few of these effects are illustrated on these pages. A 3D text object is shown in Figure 8.16, and was created by first converting a greyscale image of the word "*vlib*" to a $486 \times 160 \times 2$ dataset and then by using the vlDataset command to associate this dataset with the geometry field of the volume

object. The normals in the left image were estimated by the system using the central differencing method in discrete dataset space; the result of which is that the object appears to have a smooth, flat top. In contrast, the object in the right image was rendered with the `VL_DATASETNORMALS` toggle option turned off, so the system estimated the normals directly in world space; the result is that the object appears to have a rounded top. It is important to note that the specification of the object in terms of the field-based object model remained the same for both images.

A second example of how toggle options affect rendering is shown in Figure 8.17. A simple scene comprising several objects, all with high specular reflection values and inter-object reflections, was modelled. The `VL_REFLECTOBJECTS` toggle option was turned on before any of these objects were specified in order to make them all reflective. The left image clearly shows a sphere resting on a surface, where both the surface and the sphere are reflecting the surrounding objects which are just out of view of the camera. In contrast, only the sphere is visible in the right image although the surface and all surrounding objects are clearly present since they appear in the sphere's reflections. This interesting rendering effect was achieved by turning the `VL_VISIBLE` toggle option off just after specifying the visible sphere but before specifying the other objects in the scene. This feature is particularly useful for drawing the viewer's attention to one specific object.



(a) (b)

Figure 8.17: Effect of toggling `VL_VISIBLE` (a) on and (b) off.

8.10.3 Discrete Ray-Tracing

The size of the interval between sampling points along a ray is the main factor influencing the quality and speed of rendering. *vlib* uses a canonical unit sampling interval by default although this should be adjusted by the user according to the desired image quality and the amount by which volume objects are scaled. In our scenes, we tend to scale objects by approximately 200 units in each dimension where a unit sampling interval is adequate. However, for very high quality rendering, we usually reduce the sampling interval to a tenth of a unit. Often, there is no discernible difference in image quality between this and smaller sampling intervals although the rendering time can change considerably due to the increased number of samples taken.

A consequence of modifying the sampling interval is a change in the opacity accumulation rate. A smaller sampling interval, for instance, causes opacity to accumulate more quickly creating the illusion of opaquer objects. A more detailed discussion of this can be found in Section 4.5.2. The API accommodates this by allowing the user to specify not only the actual sampling interval, δ, but also the sampling interval, Δ, at which opacity accumulation would be correct. The default value for Δ is also unity. The sampling interval can therefore be reduced to achieve high quality images without affecting the opacity accumulation and the "correctness" of the images. Both sampling intervals, δ and Δ, are specified using the Scope 2 command:

`void` **vlStepSize**(`vlfloat` δ, `vlfloat` Δ);

The sampling intervals are specified on a per-object basis.

vlib supports recursive ray-tracing with shadows, reflection and refraction. A ray will reflect upon encountering a voxel with a non-zero specular reflection field value and a non-zero normal vector (providing that the `VL_REFLECTOBJECTS` toggle option is set), and each reflective ray will recursively spawn new reflective rays until either a non-reflective voxel is hit, a ray passes out of the scene, or the permitted depth of recursion is exceeded. A default depth of recursion of 2 is used although this can be changed with the Scope 2 command:

`void` **vlRayDepth**(`vlint` *depth*);

If repeated calls to this command are made when defining any given scene then only the depth specified by the final call will be used by the system — all other calls will be ignored. The `ERR_BADPARAMS` error will be reported if a negative depth is specified.

If a ray terminates without accumulating full opacity then a background colour will be added into the ray's accumulated colour store. (If the `VL_REFLECTBACKGROUND` toggle option is not set for an object then reflective rays spawned inside that object will not reflect the background colour.) A scene's background colour is set with the Scope 2 command:

`void` **vlBackground**(`vlfloat` *r*, `vlfloat` *g*, `vlfloat` *b*);

where *r*, *g* and *b* are the intensities of the red, green and blue colour components respectively and will be clipped by the system if they fall outside the range $[0, 1]$. The default background colour is black. As before, if repeated calls to this command are made then only the colour specified by the final call will be used by the system.

8.10.4 Parallel Rendering

A *vlib* implementation should support parallel processing whereever possible since high quality volume ray-tracing, especially in the case of large images or animation sequences, is slow. A number of programming environments and libraries exist which remove much of the tedium from parallel programming by launching *slave* programs (often copies of a master or parent process) on separate processors and pass user-defined messages between them. Among such environments are Express, PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

The API allows the user to specify any number of processors over which an image should be rendered. This is done with the Scope 1 command:

```
void vlParallel( vlstring machine, vlstring filename,
   vlint argc, vlstring * argv, vlint nice, vlint tasks );
```

where *machine* is the name of the slave processor to launch the slave program on, *filename* is the full path and name of the slave program to be executed, *argc* and *argv* are the number and contents of command line arguments respectively, *nice* is the UNIX "nice" value to be used for the slave process and *tasks* is the number of slave processes to run concurrently on the specified slave processor.

If the slave programs are exact copies of the parent program then it may happen that some user-controlled tasks will be executed repeatedly, albeit unnecessarily. For example, the user might wish his program to create a volume dataset and store it on disk for the slave processes to share and render. However, rather than instruct each slave process to create the same dataset and therefore potentially waste processing time, it should be created once only by the parent process. A program can determine whether it is a parent or slave using the Scope 7 command:

```
vlbool vlIsParent( void );
```

The details concerning parallel processing, data distribution and task allocation should be transparent to the user as far as possible. In some instances though, it might be necessary to make the user aware that file permissions and environment variables need to be set in order for the parallel programming environment to operate correctly.

8.11 Lighting

A few common light sources are currently supported by the API, namely ambient, point and directional types. They are defined respectively by the following three Scope 2 commands:

- ambient:

  ```
  void vlAmbient( vlfloat r, vlfloat g, vlfloat b, vlfloat i );
  ```

- point:

  ```
  void vlPointLight( vlfloat x, vlfloat y, vlfloat z,
     vlfloat c1, vlfloat c2, vlfloat c3,
     vlfloat r, vlfloat g, vlfloat b, vlfloat i );
  ```

- directional:

  ```
  void vlDirectionalLight( vlfloat nx, vlfloat ny, vlfloat nz,
     vlfloat r, vlfloat g, vlfloat b, vlfloat i );
  ```

where in each case *r*, *g* and *b* are the red, green and blue colour components and *i* is the brightness, which simply scales them. If any of these parameters lie outside the range $[0, 1]$ then they will be clipped but no error will be reported.

A scene can contain a maximum of one ambient light source. If multiple calls to vlAmbient are made then only the final call will be considered by the system — all preceeding calls will effectively be ignored. However, nny number of point and directional lights can be included. The point light source is the only one that attenuates over distance, since the directional light source is commonly

Table 8.8: The `vlcamera` data type.

Name	Type	Description
vrp	`vlvector`	View referenced point.
lookat	`vlvector`	A point at the centre of projection. The view plane normal is formed by *lookat* − *vrp*.
vuv	`vlvector`	View up vector.
distance	`vlfloat`	Distance from the camera to the viewpoint. Affects the field of view.
viewport	`vlfloat [2]`	Dimensions of the view port.
type	`vlenum`	The projection type. Either `VL_PERSPECTIVE` or `VL_ORTHOGRAPHIC`.

considered to be a non-attenuating light situated an infinite distance away. A point light, positioned at (x, y, z), attenuates light according to f_{att}, where for example:

$$f_{\text{att}} = \min(1, \frac{1}{c_1 + c_2 d + c_3 d^2}).$$

A directional light source emits light in a constant direction along the vector $[n_x, n_y, n_z]$. If the arguments passed to vlDirectionalLight are such that this vector has a zero length then the `ERR_BADPARAMS` error will be reported.

8.12 System Output

A rendering engine meeting the full API specification will be able to generate colour images or discrete volume datasets to represent each of the scenes modelled by the user. A discussion of these two output types now follows.

8.12.1 Image Output

A rendering engine can only render a scene if both a virtual camera and appropriate information about the output image are properly defined. A rendering engine supporting this API will be triggered implicitly when a scene block is terminated with vlEnd, but only if both a virtual camera and the dimensions, aspect ratio and file name of the output image are specified. If a camera is defined but not an image (or vice versa) then the scene block's vlEnd command will fail as the scene specification is considered to be incomplete. In this case, the `ERR_INCOMPLETE` error will be reported.

To place a virtual camera into a scene, a variable of type `vlcamera` (Table 8.8) must be declared and initialised appropriately. The camera model supported by the current version of this API is relatively simple but has proved sufficient for producing a varied range of images. It is possible for a rendering engine to support a variety of advanced effects in a camera model, such as a depth of field, motion blur and other types of projection. However, such effects are not related directly to volume graphics and therefore have not been supported by us so far. A camera is defined by passing a reference to the camera variable to the following Scope 2 command:

`void` **vlCamera**(`vlcamera` **camera*);

If multiple cameras are defined in any given scene then only the last camera will be considered by the system — all others will be ignored. This is due to the fact that this API supports rendering engines that generate images from one viewpoint only. If either the specified *vuv* vector or the view plane normal vector, computed with *lookat* − *vrp*, has a zero length, or if the two vectors are collinear, then the call to vlCamera will fail and the ERR_BADPARAMS error will be reported. The same error will also be reported if a negative distance or a negative viewport dimension is specified.

The output image is defined with the Scope 2 command:

`void` **vlImage**(`vlstring` *filename*, `vlfloat` *aspect*, `vlint` *w*, `vlint` *h*);

In the event that multiple calls to this command are made, only the last call will be considered by the system. The underlying rendering engine will render the scene into a 24-bit colour frame buffer with dimensions $w \times h$ and an aspect ratio *aspect*. Typically, the aspect ratio will be w/h, however other values can be used to achieve a squashed or stretched image. The contents of the frame buffer will be written to a file with the path, name and extension specified in *filename*. The frame buffer will not be retained in memory and therefore the image must be loaded from disk by the user if it is to be displayed or subsequently processed by the user's application. The file suffix specified in *filename* is used to indicate the file format required in the event that multiple formats are supported by the graphics system. If, for instance, the suffix "*.ppm*" is given then the system will write the frame buffer to disk in the *Portable Pixmap* format. If only a single format is supported (e.g., BMP on a Windows implementation) then the image will be saved on disk in the supported format regardless of any file suffix given. If any of the numeral arguments to vlImage are zero or negative then the ERR_BADPARAMS error will be reported.

If an implementation of the *vlib* API supports discrete ray-tracing then, by default, it should render images by casting a single ray through the centre of each pixel. Although rendering using only one ray per pixel is reasonably fast, an aliasing phenomenon will most likely occur in regions of the image where the colour between adjacent pixels is significantly different. A more accurate image could be attained through super-sampling with the Scope 2 command:

`void` **vlSuperSample**(`vlint` *s*);

where s^2 is the actual number of rays to be cast per pixel. A pixel intensity is derived by averaging the colours determined by all rays cast through that pixel. If $s = 1$ then no super-sampling will be performed since only one ray will be cast. If $s \leq 0$ then the call will fail and the ERR_BADPARAMS error will be reported.

It is possible to render only a part of an image. This is useful since it saves the user time while modelling one object by not rendering surrounding objects which may happen to be visible to the camera. Also, it offers an alternative mechanism for parallel rendering if the user does not have a network, or does not have appropriate message passing software required by the implementation for parallel processing. A rectangular region of the output image is specified with the Scope 2 command:

`void` **vlWindow**(`vlint` x_1, `vlint` y_1, `vlint` x_2, `vlint` y_2);

If multiple calls to this command are made then only the final call will be considered by the system — all preceeding commands will effectively be ignored. The rectangular region will be clipped if it extends beyond the dimensions of the image and all image pixels that are not inside the rectangular

region specified will be set to black. A call to this command will be ignored if an output image is not specified using vlImage.

A second type of output image is supported to enable the user to render templates to help create texture maps for the sake of the projective texture mapping technique (Chapter 6). A volume object will be rendered from the perspective of an intermediate surface bounding the object's original bounding box if the Scope 3 command:

```
void vlRenderTexture( vlenum type, string filename, vlenum bpp,
  vlint w, vlint h );
```

is issued. The arguments to this command differ slightly from those passed to vlImage. *filename*, *w* and *h* are as before. *type* is type of intermediate surface (e.g., sphere, cylinder, cube) that rays will be projected from and must be one of the types listed in Table 8.6 on page 184. In Chapter 6, we mentioned briefly that 8-bit distance maps sometimes lead to sub-standard displacement mapping. Therefore, the user is given the option of rendering either 8- or 16-bit textures by setting *bpp* to either VL_8BIT or VL_16BIT respectively. Any number of calls to vlRenderTexture can be issued from inside an object block. The images rendered will be super-sampled according to vlSuperSample and will be rendered in parallel if calls to vlParallel are made. The rules governing the image file format are the same as those for vlImage. If any of the numeral arguments to vlRenderTexture are zero or negative then the ERR_BADPARAMS error will be reported.

8.12.2 Volume Output

As shown in Chapter 4, our new *direct evaluation* rendering method poses several advantages over that traditionally taken in volume graphics in which object models are voxelised prior to rendering and are then reconstructed during the sampling process. To recap briefly, the proposed method reduces memory overhead and results in very high quality images by rendering directly from continuous functions rather than discrete approximations to continuous functions. It also allows rendering parameters to be assigned to objects in a multi-object environment on a per-object basis. A drawback of our approach is that scalar fields with complex specifications could take an undesirably long time to evaluate. This problem is accentuated particularly when rendering animation. To assist in overcoming this problem, a mechanism has been incorporated into the API which provides users with the option to voxelise a region of the environment into one or more regular volume datasets—one for each scalar field. A complex field specification can then be replaced with a simple call to vlDataset.

A call to the Scope 2 command:

```
void vlVoxelize( vlvoxelize * voxelize );
```

will instruct the system to voxelise all fields referenced by *fields* in the *voxelize* variable over a bounded region of the environment. vlVoxelize can be called instead of, or in addition to, vlImage and vlCamera. Voxelisation is performed by sampling the appropriate fields at regular intervals in world space:

$$\left\{(x_1 + i\frac{x_2 - x_1}{D_x - 1}, y_1 + j\frac{y_2 - y_1}{D_y - 1}, z_1 + k\frac{z_2 - z_1}{D_z - 1})\right\}$$

where

$$(i, j, k) \in \{0, \ldots, D_x - 1\} \times \{0, \ldots, D_y - 1\} \times \in \{0, \ldots, D_z - 1\}.$$

Table 8.9: The `vlvoxelize` data type.

Name	Type	Description
fields	`vlint`	A reference to the fields to be voxelised.
v1	`vlvector`	Corner 1 of the axis-aligned box bounding the region to be voxelised.
v2	`vlvector`	Corner 2 of the axis-aligned box bounding the region to be voxelised.
min	`vlfloat`	The minimum voxel value to be stored.
max	`vlfloat`	The maximum voxel value to be stored.
dim	`vlint [3]`	The resolution of the resulting volume dataset(s).
type	`vlenum`	The data type used to store each voxel value.
endian	`vlenum`	The endianness of the data. Either `VL_BIG` or `VL_LITTLE`.
filename	`vlstring`	The name of the file storing the resulting volume dataset(s).

A field will have a zero value at any point where no volume objects exist.

The resulting regular rectilinear volume dataset will be stored on disk in a file named *filename* (again, in the *voxelize* variable). The file naming convention is similar to that used by vlImage whereby the suffix of the specified file name establishes the file format used to store the dataset, in the event that multiple formats are supported. If multiple fields are voxelised by a single call to vlVoxelize and the requested file format does not support vector (multi-channel) volume datasets then the `ERR_BADPARAMS` error will be reported. To alleviate this problem, the user should issue multiple calls to vlVoxelize referencing a single field each time and specify different file names for each call. (It must be noted that the system duplicates and stores the structure passed into the vlVoxelize command thereby enabling the user to pass the same *voxelize* structure, with changes made, to multiple vlVoxelize commands.) The numeric data type used for storing voxels on disk is specified in *type* and must be one of the types listed in the *Constant* column in Table 8.5 on page 178. If the requested file format does not support the requested data type then the `ERR_BADPARAMS` error will be reported.

8.13 Summary and Further Work

An API has been presented which supports the field-based modelling framework of Chapter 4, as well as many other significant developments in volume graphics and visualisation. In summary, some of the main features of the interface include:

- a coherent volume graphics pipeline capable of controlling both modelling and rendering effects;
- a flexible specification of complex field-based volume objects;
- the ability to specify independently the ten photometric properties of a volume object;
- a multi-object rendering scheme with support for constructive object representations;
- a number of lighting effects including shadows, reflection and refraction;
- the direct volume, direct surface and maximum intensity projection rendering algorithms;
- built-in linear ramp and lookup table transfer functions;

- a user-definable normal estimation scheme;
- voxelisation of one or more fields of a volume object;
- support for up to 64-bit integer and floating-point volume datasets;
- parallel rendering.

As part of this work, a complete volume rendering library, compliant with the API specification, has been produced using approximately 10,000 lines of C code and released freely to the volume graphics community. In addition, a secondary library containing a number of useful plug-in functions, such as solid textures, distance fields, spatial transfer functions and camera routines, has also been developed. All rendered images included in this thesis have been created using these libraries. A detailed description of some of the more complex implementation issues is given in the following chapter.

The speed of rendering using our implementation is governed by many factors including the amount of available processing power, image resolution, the super-sampling rate, recursion depth, sampling interval, object complexity, the rate of opacity accumulation and the normal estimation method. A scene can take anywhere between a fraction of a second, or if the rendering parameters are altered substantially, several hours to render, however most images shown in this thesis have taken several minutes to render on seven 1.4GHz Athlon PCs running in parallel. A hardware implementation of the interface could achieve interactive frame-rates but might not support all available features, such as super-sampling or direct surface rendering.

The future of this work lies primarily in maintaining and expanding the interface so that it reflects the developing needs of the volume graphics community. Undoubtedly, this is a continual process and already several third parties have started work on porting the code to untested platforms and adding new rendering algorithms, such as a new pre-integrated volume rendering method. A conceivable expansion would be to redesign elements of the API so that the number of fields in an object, and the role of those fields, is constrained by the rendering engine and not by the interface itself. So, a new rendering engine supporting an unusual lighting model, or non-photo-realistic rendering, will immediately be supported by the API with no change to the interface design.

The author has presented this work at the 18th Eurographics UK Conference (Swansea, UK), the 2nd International Workshop on Volume Graphics (New York, USA) and by invitation at the Libre Software Meeting (Bordeaux, France), where the source code for the *vlib* implementation appeared on the electronic proceedings. The work has also been included in *Volume Graphics 2001* published by Springer.