# vlib: A Volume Graphics API

Andrew S. Winter and Min Chen

University of Wales, Swansea

**Abstract.** This paper describes *vlib*, a generic application programming interface for volume graphics which supports many of the significant developments in the field to date. We present an overview of the interface and describe how its novel object modeling framework is able to facilitate a variety of modeling and rendering features, including scene graphs allowing constructive object representations, normal perturbation, spatial deformations, hypertexturing and more. We also discuss a volumetric ray-tracing algorithm for producing high quality images with a minimal memory overhead. The paper closes with some comments on our Open Source implementation of the interface and its underlying graphics system.

## 1 Introduction

Since it was first proposed in 1993 [1], volume graphics has established itself as an independent graphics technology. Its success can be attributed to a number of exclusive properties which the traditional surface-based methods of image synthesis lack, such as the facilitation of a true 3D description of space, and the ability to render solid and amorphous objects in a consistent manner. But despite the advancements made in the field, volume graphics today still stands someway short of its potential, especially in terms of the generality and availability of modeling and rendering software.

This article presents an application programming interface (API) which exploits many of the significant developments in volume graphics to date. The nature of the API allows it to meet traditional visualization requirements as well as those currently emerging in the art and entertainment industries. In fact, we believe it to be generic enough to form the backbone of both GUI-based graphics and visualization systems, and also volume modeling languages. Invariably, such systems and languages will arise in time as the field of volume graphics develops.

The discussions in this article are organized as follows. In section 2, we present the unique object modeling framework upon which the specification of the API is based. Then, an overview of our new system is given in Section 3, followed in Section 4 by an in-depth description of its modeling capabilities. A discussion of the rendering features is presented in Section 5, and finally, a few concluding remarks on the potential of our API are made in Section 6.

## 2 A Volume Modeling Framework

Every graphical system has some underlying mathematical framework from which visual properties are derived for image synthesis. This section presents a field-based framework which is fundamental to the material contained in the remainder of this article.

### 2.1 Volume objects (general)

We begin by introducing some standard terminology. Let $\mathbb{E}^3$ be 3D Euclidean space and $\mathbb{R}$ be the set of all reals.

**Definition.** A *scalar field* is a total function $f : \mathbb{E}^3 \longrightarrow \mathbb{R}$.

A scalar field represents a single property of an object, such as temperature, opacity, speed or color. In the context of a graphics system, a scalar field is only really useful if it represents a photometric property.

**Definition.** A *spatial object* is a tuple of $k > 0$ scalar fields $\mathcal{O} = (f_1, f_2, \ldots, f_k)$.

A typical visualization system might use only four scalar fields to define pointwise opacity and color values over $\mathbb{E}^3$. The assumption is ordinarily made that every spatial object has at least an opacity field which defines its visible parts.

**Definition.** A scalar field $f$ is *bounded* if $f(\mathbf{x}) = 0$ for all $\mathbf{x}$ not in a set $\mathcal{X}$.

Although spatial objects are infinite, a boundedness restriction is desirable for a software implementation given the limitations of current computer hardware. For our graphics system, we say that $\mathcal{X}$ is $[0,1]^3$. The implication of this restriction is shown later on.

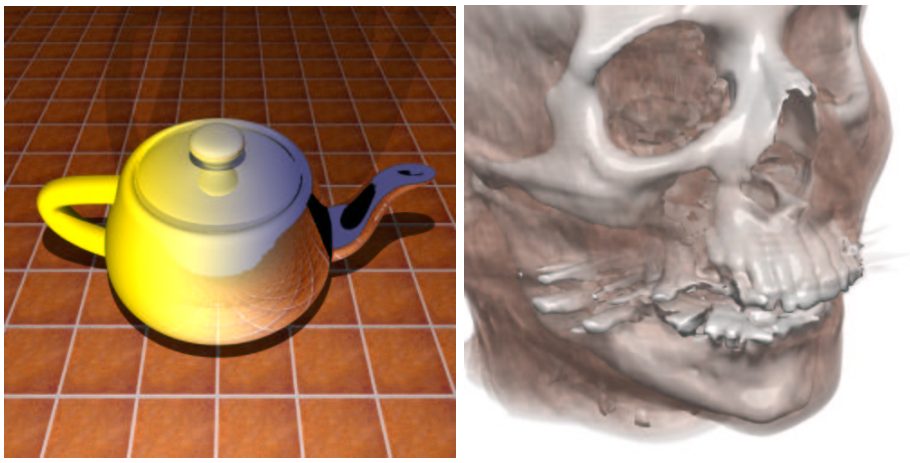
**Definition.** A *volume object* is a spatial object whose $k$ scalar fields are all bounded.

A volume object is therefore considered to be a spatial object occupying only a finite region of Euclidean space. It is clear from our definitions that the terms "volume dataset" and "volume object" are not used synonymously. A dataset does not define an object, but merely one property of an object. Moreover, a dataset may not always offer a total mapping from $[0,1]^3$ to $\mathbb{R}$. We assume that elements of a dataset are nodal and not cellular, and therefore totalness is achieved by accompanying a dataset with an interpolation function. This will be described in greater detail later on.

### 2.2 Volume objects (API)

The object model for the API has been designed so that every major visual property of a volume object can be assigned heterogeneously. Using the same notation as above, the model is written

$$\mathcal{O} = (\mathbf{o}, \mathbf{r}, \mathbf{g}, \mathbf{b}, \mathbf{k_a}, \mathbf{k_d}, \mathbf{k_s}, \mathbf{n}, \mathbf{r_f}, \mathbf{f})$$



(a) (b)

**Fig. 1.** Heterogeneous reflection coefficients. (a) A smooth transition from a non-reflective ceramic-like material (left) to a highly-polished metal (right); (b) Shiny bone and dull skin.

where

| | | |
|---|---|---|
| $\mathbf{o}$ | = | opacity; |
| $\mathbf{r}, \mathbf{g}, \mathbf{b}$ | = | red, green and blue color components; |
| $\mathbf{k_a}, \mathbf{k_d}, \mathbf{k_s}$ | = | ambient, diffuse and specular reflection coefficients; |
| $\mathbf{n}$ | = | specular exponent; |
| $\mathbf{r_f}$ | = | index of refraction; |
| $\mathbf{f}$ | = | geometry. |

The geometry field is non-visual but provides a useful means of defining object shape without placing any dependency on the opacity field. We restrict the co-domain of the *vlib* scalar fields to $[0, 1]$. Aside from $\mathbf{n}$, $\mathbf{r_f}$ and $\mathbf{f}$, all fields have co-domains which fall naturally into this range and may therefore be modeled consistently. In order to permit a simple and efficient implementation of the API, total consistency is achieved by normalizing $\mathbf{n}$, $\mathbf{r_f}$ and $\mathbf{f}$. A thorough discussion of this process is reserved for *vlib* specification document [2]. The specification of heterogeneous reflection coefficients is a native feature of *vlib*, but is often quite difficult to achieve in conventional graphics systems. Some benefits of this heterogeneity are shown in Figure 1.

## 3 An Overview of the API

*vlib* (short for "volume library") is a programmatical interface between a high-level volume modeling program and a lower-level volume-based rendering and

voxelization system. It provides a coherent and consistent approach to modeling using several dozen commands which support the majority of significant trends in volume graphics. This section presents an overview of the API. It gives a brief discussion of the interface style and command groups before describing the types of output offered by a system implementation.

### 3.1 The interface and command groups

The interface style is similar to that of a few important surface modeling systems such as OpenGL and the RenderMan Interface. The design principles of these have proven successful and are familiar to many graphics programmers. As a result, we have chosen to adopt a similar style. The current specification of the API contains just forty-four commands[1] that provide a great deal of control over the system operations, camera, lighting, rendering properties, volume objects, scene graph construction and output. Figure 2 illustrates some of these commands and shows how a modeling program may communicate with a renderer by way of the *vlib* interface.

Our API is initialized and terminated by making calls to vlInit and vlEnd, shown on lines 6 and 27 respectively. In turn, any number of scenes may be defined between these commands. A scene specification consists of a number of light source (lines 13–15) and object (16–25) definitions, as well as some rendering instructions cited between calls to vlScene and vlEnd. The API maintains a state of instructions so that any rendering parameters set will be applied to all light sources and objects defined subsequently. The vlEnd command used to close a scene specification (line 26) will request either a graphical or a volumetric representation (or both) of the scene from the underlying rendering system.

The API is generally insensitive to the order in which commands are called, however there are a few considerations to note:

- The sub-objects in a volume hierarchy are indexed by the order in which they are declared.
- The transformation operations do not commute and so they must be issued in a planned order.
- The output of one field specification function may be used as input to another. An incorrect ordering of the specification instructions may therefore produce undesired results.

Additionally, some commands are illegal within certain scopes. The state of rendering parameters, for instance, may not be changed during the specification of an object. Commands which are called illegally will set an error flag which should be checked regularly by the user with the vlError function.

[1] The term "command" refers to pre-defined system subroutines and is not used synonymously with the word "function".

```
01 #include <vlib.h>
02 #include <vlaux.h>
03
04 vlfloat myMap[] = {0.001, 1.0, 1.0, 0.3};
05 main() {
06  vlInit();
07    vlScene();
08       vluCamera(175.0, 250.0, -350.0,
09           175.0, 140.0, 70.0);
10       vlImage("stone.ppm", 1.0, 100, 100);
11       vlBackground(WHITE);
12       vlMapping(1, VL_LOOKUP, 2, 1, myMap);
13       vlPointLight(-200.0, 380.0, -200.0,
14           VL_NOATTENUATION, WHITE, 1.0);
15       vlAmbient(WHITE, 1.0);
16       vlObject();
17          vlScale(350.0, 300.0, 300.0);
18          vlFunction(VL_F, &VL_PARAB1, 0);
19          vlMap(VL_O, VL_F, 1, 1);
20          vlMap(VL_KS, VL_F, 1, 2);
21          vlFunction(VL_R|VL_G|VL_B, &stone, 0);
22          vlSet(VL_KA, 0.3);
23          vlSet(VL_KD, 0.6);
24          vlSet(VL_N, VLSPECULAR(20));
25       vlEnd();
26    vlEnd();
27  vlEnd();
28 }
```

**Fig. 2.** A basic *vlib* program and its output.

### 3.2 System output

Ordinarily, a volume graphics pipeline will make use of entirely disjoint modeling and rendering stages. Modeling involves the voxelization (or discretization) of some arbitrarily defined scene in order to produce a volumetric dataset. Rendering entails computing over the discrete volume elements and calculating shading parameters in order to eventually determine pixel intensities.

A discrete scalar volume dataset is transformed into a continuous scalar field $f$ by way of an interpolation function. Although $f$ may be relatively fast to evaluate, it does not necessarily permit high-quality rendering. Clearly, the fidelity of a reconstructed field to that of the original is dependent on both the sampling frequency during discretization (which determines the resolution of the output dataset), and the quality of the reconstruction filter.

It was mentioned in Section 2 that each photometric property of a volume object can be represented by a function $f(x, y, z)$. Instead of enforcing the discretization and reconstruction of such a function, thereby sacrificing data and incurring a memory overhead, we retain the original function description and pass it directly to the renderer for evaluation during sampling. Using this so-called *direct evaluation* approach, we believe that in many cases we are able to produce images of a higher quality and with less memory overhead than is ordinarily possible. Several images presented in this article have been rendered using this approach.

The *vlib* interface also includes an option to allow the the user to discretize a subspace of the scene into one or more regular scalar volume datasets of any reso-

lution. This functionality is useful for two reasons. Foremost, our system should cater for users of dedicated volume rendering hardware (e.g., VolumePro [3]), which generally accept only discrete representations of objects as input. Secondly, it may not always be preferential to render environments using the direct evaluation approach described above. This may be due to the complexity of some of the chosen functions attributing to an unreasonable increase in rendering time, or alternatively to the user wishing to trade image quality for rendering speed. It may even be advantageous to specify and voxelize an object using the API, and to then feed the resulting dataset back into the API for rendering. Each field voxelized by the system is written to a separate output file as *vlib* does not support tensor datasets.

## 4 Modeling Objects and Scenes

Earlier, we described a volume object as a collection of ten co-spatial bounded scalar fields. This section describes how these objects are modeled; in other words, how their constituent fields are specified using the *vlib* interface. It is helpful to realize that every object field is a function $f : \mathbb{E}^3 \longrightarrow [0, 1]$, and so the collection of commands used to specify one field equally apply to the rest. The process of modeling a scene amounts to modeling a collection of objects, therefore we do not give any mention to "scene modeling" explicitly. In addition to the field specification commands, discussions in this section also include object transformation and arbitrary spatial deformation.

The discrete ray-tracing algorithm will sample a finite number of points through an object during the rendering process. For a single point $\mathbf{x}$, we concentrate on modeling $f(\mathbf{x})$, for all object fields $f$, in the object specification. For instance, the command on line 22 of Figure 2 assigns 0.3 to the ambient reflection coefficient for all $\mathbf{x} \in [0, 1]^3$. The location of $\mathbf{x}$ in 3D space is largely abstracted away from the user, but as we show in this section, it is possible to determine and compute over the location of $\mathbf{x}$ when necessary.

### 4.1 Declaring a volume object

An object specification comprises of a series of ordered instructions between calls to vlObject and vlEnd. This is illustrated in lines 16 to 25 of Figure 2. It is possible to nest objects to form a scene graph, which consequently facilitates constructive object representations such as constructive volume geometry [4]. Any number of hierarchies may be incorporated into a scene but their object boundaries should not intersect. This is because we define no default mechanism for combining their overlapping fields.

### 4.2 Simple assignments

The simplest means to define $f(\mathbf{x})$ is by way of assigning an explicit constant value. The vlSet command performs this assignment and is undoubtedly one of


(a) (b)

**Fig. 3.** The opacity and geometry fields (**o** and **f**) of this object are derived from a dataset, which we created by converting a two-dimensional PPM image.

the most primitive commands in the interface. In fact, it may be used internally to implement many of the other field specification commands. vlSet must be used in conjunction with a field function if $f(\mathbf{x})$ is to be set for only some $\mathbf{x} \in [0,1]^3$. This is described later on.

### 4.3 Volume datasets

Due to our view of discrete datasets as a collection of nodes rather than a collection of cells, it is necessary to enforce continuity if a dataset is to be used as a scalar field. And as a scalar field, its gray-level values must be in $[0,1]$ to adhere to the object model given in Section 2. This restriction presents two problems. Firstly, the majority of existing third-party data files (including emerging standards among medical formats) have either 8 or 16 bit gray-level values, and therefore do not fit into the proposed model. Secondly, the amount of memory required to store volume data could increase dramatically if dataset normalization is made a requirement.

A simple file format has been designed to overcome these problems. The volume data is preceded by a magic number for identification; volume dimensions $(\mathsf{x}, \mathsf{y}, \mathsf{z})$; a flag indicating the number of bytes per voxel; and values $\mathsf{a}$ and $\mathsf{b}$ for voxel normalization.

It is not necessary for $\mathsf{a}$ and $\mathsf{b}$ to represent the minimum and maximum data values present in the file. For registration and segmentation applications this is important to note. A voxel value $v$ extracted from the file is normalized at

render-time with

$$v_n = \min\left(1, \max(0, \frac{v - \mathsf{a}}{\mathsf{b} - \mathsf{a}})\right)$$

vlLoad retrieves a dataset from disk and may be called from anywhere outside an object specification. A consequence of restricting the scope of this command is that multiple objects (and fields) may share volume data to reduce memory consumption. Figure 3(a) shows a chess board scene comprising thirty-two playing pieces and a board. There are sixteen pawns, four rooks, four knights, four bishops, two queens and two kings. However, only six volume datasets were required by the program to produce the image.

As well as the file name, vlLoad is passed a unique identifier to which the volume data is subsequently referred. Then, a dataset may be cast to a scalar field using the vlDataset command. The interpolation function associated with the cast is that stored in the system's state at the time.

Figure 3(b) shows a "text" object whose opacity and geometry fields were set using a volume dataset. The dataset itself was made by converting a simple 2D image using a tool.

### 4.4 Transfer functions

Many visualization systems depend on carefully designed transfer functions for mapping from gray-level volume data to photometric attributes for image synthesis. We support two kinds of transfer function commonly found in the literature.

For the descriptions that follow, assume a series of tuples

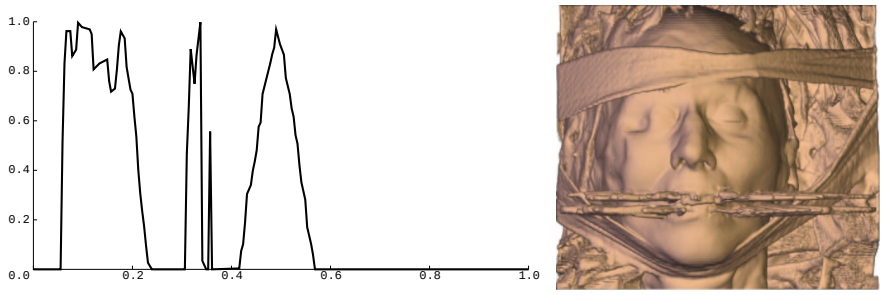
$$(a_1, b_{1,1}, \ldots, b_{1,k}), (a_2, b_{2,1}, \ldots, b_{2,k}), \ldots (a_n, b_{n,1}, \ldots, b_{n,k}) \in [0,1]^{k+1}$$

ordered such that $a_1 < a_2 < \ldots < a_n$. These form the input points of a transfer function (the $a$ values) and a series of $k$ corresponding output points (the $b$ values). From this, we can define two transfer functions of the form $\Phi : [0,1]^3 \times \mathbb{N} \longrightarrow [0,1]$, each taking the value on a field $f(\mathbf{x}) = v$ as well as an output index $j$ as input, and returning a scalar value which may be assigned to a second field $g(\mathbf{x})$. The supported lookup table $\Phi_t$ and linear ramp $\Phi_r$ transfer functions are as defined below:

$$\Phi_t(v,j) = \begin{cases} 0 & \text{if } v \leq a_1 \text{ or } v > a_n \\ b_{i+1,j} & \text{if } \exists i \in \{1..n-1\} \text{ such that } a_i < v \leq a_{i+1} \end{cases}$$

$$\Phi_r(v,j) = \begin{cases} \frac{v\, b_{1,j}}{a_1} & \text{if } 0 < v \leq a_1 \\ b_{i,j} + \frac{(v-a_i)(b_{i+1,j}-b_{i,j})}{a_{i+1}-a_i} & \text{if } \exists i \in \{1..n-1\} \text{ s.t. } a_i < v \leq a_{i+1} \\ b_{n,j} + \frac{(v-a_n)(1-b_{n,j})}{1-a_n} & \text{if } 1 > v > a_n \end{cases}$$

Figure 4 shows a graphical representation of a linear ramp transfer function. We used this function in the specification of the CT head object shown alongside




(a) (b)

**Fig. 4.** The graph in (a) represents the linear ramp transfer function that was used to define the opacity field in (b).

```
01 sphere(vlvector p, void *misc) {
02    float d =
03       (p[0] - .5) * (p[0] - .5)
04     + (p[1] - .5) * (p[1] - .5)
05     + (p[2] - .5) * (p[2] - .5)
06    vlSet(VL_ALLFIELDS, 1.- 2.* sqrt(d));
07 }
```

```
01 marble(vlvector p, void *misc) {
02    float v =
03       vluTurbulence(p, 2.17, 7, 0) + 0.8;
04    vlSet(VL_R, vluSpline(v, 10, rSpline));
05    vlSet(VL_G, vluSpline(v, 10, gSpline));
06    vlSet(VL_B, vluSpline(v, 10, bSpline));
07 }
```

**Fig. 5.** Field functions implementing a spherical distance field (left) and a solid marble texture (right).

and also in the color plate. The gray-level density values from the dataset are shown along the horizontal axis and the output opacity values from the transfer function are shown vertically.

The API requires that transfer functions are declared using the vlMapping command. This allows the system to sort, validate and optimize the data for fast retrieval during rendering. During the specification of an object, the vlMap command passes the value on a user-defined field through the transfer function and assigns the resulting scalar value to one or more of the specified fields.

### 4.5 Field functions

A *field function* is a user-defined procedure **f**(`vlvector x, void *misc`). The primary role of a field function is to allow the user to compute over the location of a point **x** within an object. With this ability, these functions can serve many uses within the API. By allowing the user greater flexibility over an object specification, they facilitate the creation of solid textures, hypertextures, distance fields, constructive modeling operators, texture blending functions and more. Some of these uses are illustrated in the figures cited throughout this article.

Figure 5 lists the code of two field functions implementing a spherical distance field and a solid marble texture. Although the two functions are similar, they serve potentially diverse purposes within the specification of an object. Given that the position p (as it appears in the sample code) is not known to the user, it is not feasible to execute these field functions manually. Therefore, to enable incorporation into an object specification, they must be passed to the vlFunction command, which in turn should be called from within an object declaration.

The ability to build scene graphs for constructive representations of objects in 3D modeling programs is crucial. If an object has sub-objects declared within it, information about those sub-objects may be accessed through a series of commands that are callable only from within a field function. The number of sub-objects may be established with vlChildCount. vlChildHit is a function which the user may employ to determine whether or not the current point lies within a child object. Lastly, vlChildField reads the current value from a field of one of the sub-objects. It is also possible to determine the normal of a child object using vlChildNormal, but this command may not be called from within a field function.

### 4.6 Transforming objects

The concept of object space is valid only due to the boundedness restriction imposed by our object model. Using this restriction, it is possible to define a transformation from object to world space by way of a transformation matrix. Our API offers several commands for performing transformation operations, including vlScale, vlTranslate, vlRotate and vlFlip. Any transformations applied to a hierarchical object are subsequently applied to its children.

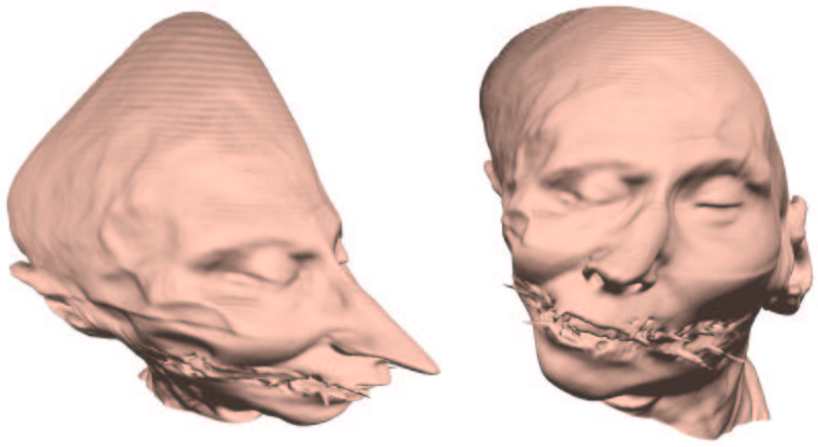
### 4.7 Spatial transfer functions

We have described how fields may be modeled using a series of API commands. The focus of modeling is specifying a value on every field $f$ such that $f(\mathbf{x})$ is defined. The location $\mathbf{x}$ is computed by the system and may be read, but not modified, from within a field function.

A spatial transfer function is similar to a field function but rather than compute over $\mathbf{x}$ to set field values, $\mathbf{x}$ may be modified permanently. The result of a permanent change to the location of points in an object is a spatial deformation. The ability to provide procedural deformations through the API gives rise to a number of interesting effects such as tapering, bending, cracking and exploding.

Figure 6(b) shows the result of a spatial transfer function as applied to a head object. This "twist" effect was achieved by rotating each point around the vertical axis by an amount which is relative to the vertical position of the point within the object. Assume the manipulation of a point is given by $\Psi : \mathbb{E}^3 \longrightarrow \mathbb{E}^3$ where the new point $\mathbf{x}'$ is given by $\Psi(\mathbf{x})$. The twist operation is given by

$$\Psi(\mathbf{x}) = \Psi(\mathbf{x}_x, \mathbf{x}_y, \mathbf{x}_z) = (r \cdot \cos\,\theta + 0.5,\ \mathbf{x}_y,\ r \cdot \sin\,\theta + 0.5)$$



(a) (b)

**Fig. 6.** Deformations rendered by direct evaluation using (a) the Bézier volume equation; and (b) a "twist" spatial transfer function.

where $r$ is the distance from $\mathbf{x}$ to $(0.5, \mathbf{x}_y, 0.5)$, and $\theta$, calculated by

$$\theta = \frac{\pi}{2}\mathbf{x}_y + \arctan\frac{\mathbf{x}_z - 0.5}{\mathbf{x}_x - 0.5}$$

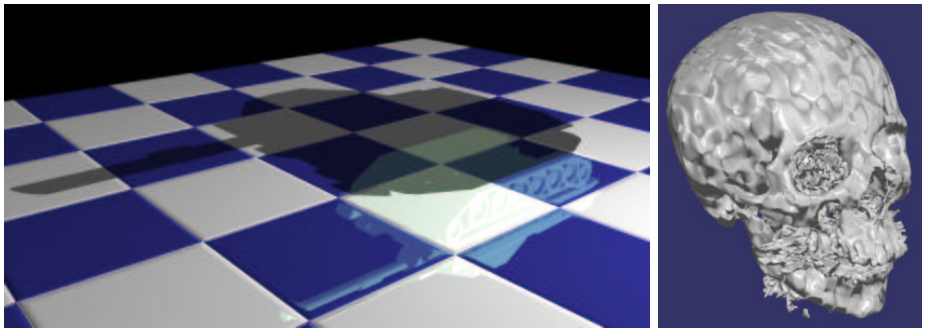
is the angle of rotation about the center vertical axis. Modifying the coefficient of $\mathbf{x}_y$ will change the severity of the twist.

A procedure implementing a spatial transfer function has exactly the same form as the field function, described earlier. It is employed within an object specification by citing the vlSpatial command from within the object declaration.

Depending on the nature of the function, it may be necessary to widen the boundary of a volume object in order to render the corresponding deformation. Consider the inverse function $\Psi^{-1}$ of the twist function $\Psi$ given above. For a point $(1, 0.5, 1)$ on an edge of the object, we have

$$\Psi^{-1}(1, 0.5, 1) = (1/\sqrt{2} + 0.5, 0.5, 0.5) \approx (1.207, 0.5, 0.5)$$

and so parts of the object will extend outside of the permitted boundary. To combat this problem, we have included a vlBoundary command in the interface, which allows the boundary of an object to be scaled (enlarged or even reduced for clipping) without scaling the object within.



(a) (b)

**Fig. 7.** (a) An "invisible" tank. The object is invisible to primary rays yet still casts shadows and reflections; (b) A bump-map effect achieved through perturbing normal vectors with a noise function.

## 5 Scene Rendering

Section 4 described the modeling capabilities of the API. Here, a number of features are presented that control the way in which objects are rendered.

### 5.1 Toggle settings

A number of rendering properties that an object may adopt have a Boolean state: either on or off. The current version of our API allows the user to control whether or not an object should

- cast and / or receive shadows;
- cast and / or receive inter-object reflections;
- refract light [5];
- appear visible to the primary ray;
- flip normals if they face away from the ray origin; and
- reflect the background color.

The system assigns a default state to each feature but this may be overridden with the vlToggle command. To avoid any confusion, we say that the rendering state may not be changed from within an object specification. Figure 7(a) shows a tank object that casts shadows and reflections, although it is invisible to the primary rays.

### 5.2 Rendering algorithms

The *vlib* API supports the direct volume (DVR), direct surface (DSR), and the maximum intensity projection (MIP) rendering algorithms. A rendering type is selected using the vlRenderType command.

As we are working with discrete ray-tracing algorithms, we have made it possible to alter the size of the interval between adjacent sampling points in order to offer a balance between rendering time and image quality. Although the rate of opacity accumulation for DVR is often altered along with the sampling interval as an unwanted side-effect, it can be regulated via the interface with the vlStepSize command. As arguments, vlStepSize is passed the actual size of the sampling interval and also a second value which defines the interval over which the rate of opacity accumulation should be correct.

The vlSurface command can be used to define any number of iso-surfaces over any field of a volume object and with any threshold. These surfaces are located and rendered using the DSR algorithm. Figure 8(b) in the color plate shows a metaball object with three such iso-surfaces, $\mathbf{f}(\mathbf{x}) = \{0.2, 0.35, 0.5\}$.

### 5.3 Normal estimation

It is essential to generate consistently accurate normals in order to produce a pleasing image. The geometry field $\mathbf{f}$ of a volume object is used for estimating normal values within the context of the API. In the ideal case, a gradient vector would be calculated analytically using a derivative function $\bigtriangledown\mathbf{f}$. Although it may be possible to compute $\bigtriangledown\mathbf{f}$ in some situations, the arbitrariness of the field specification functions makes the exact calculation of this function very difficult.

As an alternative to computing the derivative function, we provide the user with a number of commands that aid the approximation of a suitable normal vector. If the $\mathbf{f}$ field of a volume object has been set using the vlFunction command then a world-space central difference method is used to estimate the normal. This is given as

$$\bigtriangledown f(x,y,z) \approx \frac{1}{2}\begin{bmatrix} f(x+\delta_x,y,z) - f(x-\delta_x,y,z) \\ f(x,y+\delta_y,z) - f(x,y-\delta_y,z) \\ f(x,y,z+\delta_z) - f(x,y,z-\delta_z) \end{bmatrix}$$

where $(\delta_x, \delta_y, \delta_z)$ is the neighborhood size and whose dimensions largely depend on the nature of the field function passed to the vlFunction command. The vlNeighborhood command is supplied in order for the user to change the default neighborhood size which is $\epsilon^3$.

If the vlDataset command was used to set the $\mathbf{f}$ field then the normal at the point is estimated by interpolating the normals at surrounding grid points at the appropriate location in the corresponding dataset. The interface supports nearest-neighbor, tri-linear and the family of cardinal tri-cubic spline filters by way of the vlInterpolationType command. An in-depth discussion of these filters may be found in the work of Möller and colleagues [6].

Prior to the illumination of a point $\mathbf{x}$ in a volume object, the normal is estimated by the system as above. This normal vector may then be modified by supplying the object specification with one or more normal functions of the form `f(vlvector N, vlvector x, void *misc)` by way of the vlNormal command. The system-estimated normal is passed to the first user-supplied normal function as parameter $N$. It may then be changed, and the result is passed to the second

normal function as parameter $N$, and so on. The normal set by the final normal function is passed to the illumination model in order to shade the point. Figure 7(b) shows a bump mapping effect and was achieved by perturbing the system-estimated normals by way of a noise function.

For hierarchical objects, it is possible to retrieve and compute over the normal vectors of the child objects by calling the vlChildNormal command from within a normal function.

## 6 Conclusion

An API has been presented which supports many of the significant developments in volume graphics. In summary, some of its main features include:

- a coherent volume graphics pipeline;
- a flexible specification of complex field-based volume objects;
- support for independently controlling an object's ten photometric properties;
- multi-object rendering with support for constructive object representations;
- shadows, reflection and refraction;
- direct volume, direct surface and the maximum intensity projection rendering algorithms;
- linear ramp and lookup table transfer functions;
- a user-definable normal estimation scheme;
- voxelization;
- support for 8- and 16-bit volume data.

As part of our work, we have produced a complete implementation of the *vlib* interface and an underlying graphics system using fewer than 8,000 lines of C code. We have also built an auxiliary library offering several supplementary functions and operations to support *vlib*'s use.

Our code has been tested on a variety of UNIX platforms and we intend to port it to Windows in the near future. The implementation is independent from non-standard libraries and so this process should be relatively straightforward.

The speed of rendering using our implementation is governed by many factors including image resolution, the super-sampling rate, recursion depth, sampling interval, object complexity, rate of opacity accumulation, and the normal estimation method. The amount of available processing power and the efficiency of an implementation are also important factors. The Parallel Virtual Machine (PVM) has been employed for parallel rendering.

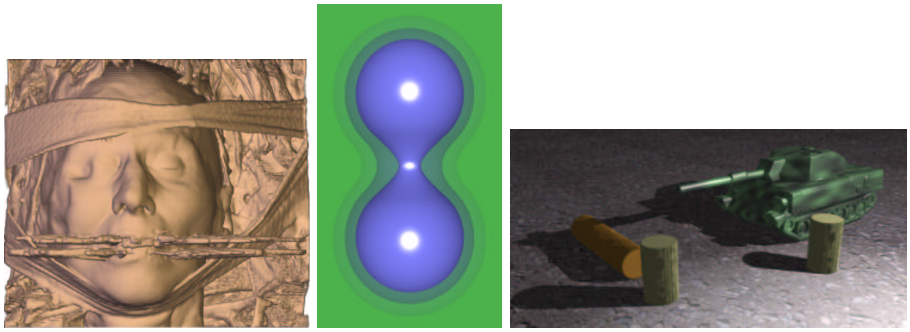
Our future work is concerned primarily with the maintenence and expansion of the interface in order for it to reflect the developing needs of the volume community. The current and future versions of our Open Source implementation may be downloaded from the Web site [2], where several supporting documents and demonstration programs are also available.

## Acknowledgements

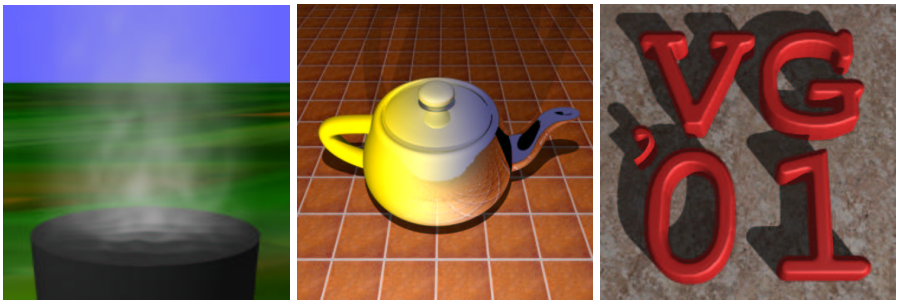
The authors would like to thank Mark Jones for the Chess Pieces, Miloš Šrámek for the Teapot, Arie Kaufman for the Tank and UNC Chapel Hill for the CThead. An acknowledgement is made to the reviewers of this paper for their useful suggestions and also to the many people who have contributed wonderful ideas to this project. This work is sponsored in part by an EPSRC research fund.

## References

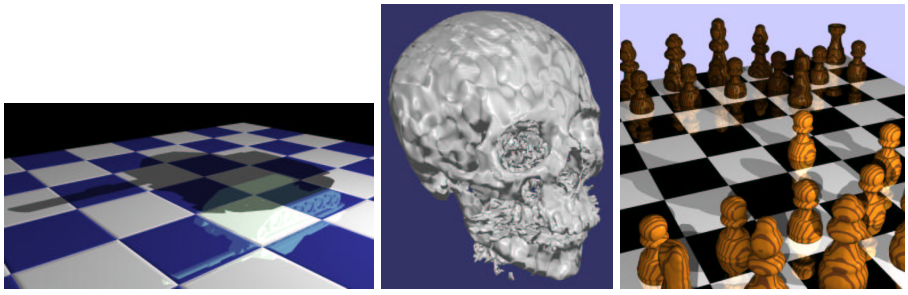
1. A.E. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, July 1993.
2. A.S. Winter. The *vlib* Web Site. *http://vg.swan.ac.uk/vlib/*.
3. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. *Computer Graphics (Proceedings of SIGGRAPH 99)*, pages 251–260, August 1999.
4. M. Chen and J.V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, 2000.
5. D. Rodgman and M. Chen. Refraction in discrete ray tracing. In *Second International Workshop on Volume Graphics*, June 2001.
6. T. Möller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. *1998 Volume Visualization Symposium*, pages 143–151, October 1998.



(a) (b) (c)




(d) (e) (f)



(g) (h) (i)

**Color Plate ("vlib: A Volume Graphics API").** A variety of images produced using the *vlib* volume graphics API: (a) a dataset visualization; (b) two spheres combined with a CVG operator; (c) a collection of solid textured objects; (d) a smoke hyper-texture; (e) a teapot with heterogeneous reflections; (f) a 2D image converted into a 3D object; (g) an "invisible" rendering effect; (h) a bump mapped skull; and (i) a multi-object environment.