VolPack User's Guide

Version 2.0beta1

Table of Contents

Section 1: Overview

Introduction to VolPack

VolPack is a portable software library for volume rendering. It is based on a new family of fast volume rendering algorithms (see Philippe Lacroute and Marc Levoy , *Fast Volume Rendering Using a Shear−Warp Factorization of the Viewing Transformation*, Proc. SIGGRAPH '94 (Orlando, Florida, July 24−29, 1994). In Computer Graphics Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 451−458). The library has the following features:

- Renders data sampled on a regular, three−dimensional grid.
- Supports user−specified transfer functions for both opacity and color.
- Provides a shading model with directional light sources, multiple material types with different reflective properties, depth cueing, and shadows.
- Produces color (24 bits/pixel) or grayscale (8 bits/pixel) renderings, with or without an alpha channel.
- Supports arbitrary affine view transformations.
- Supports a flexible data format that allows an arbitrary C structure to be associated with each grid point.
- Achieves very fast rendering times without specialized hardware.

The library is intended for use in C or C++ programs but may be useful with other programming languages. The current implementation does not support perspective projections or clipping planes. These features will be added in a future release.

The remainder of this section contains a brief introduction to the conceptual volume rendering pipeline used by VolPack, followed by a high−level description of the data structures and algorithms used by the library. This background material lays the foundation for Section 2 which describes each of the routines provided by VolPack. The routines are grouped by function and are presented roughly in the order that they would be called in a typical application. More detailed descriptions of each command can be found by consulting the man pages for VolPack. Finally, Section 3 covers some tips for maximizing rendering performance and image quality, and describes how to obtain the VolPack software.

The Volume Rendering Pipeline

The input to the volume renderer is a three−dimensional array of data. Each element of the array is a C structure containing any number of fields of data, such as tissue density or temperature. Each element is called a "voxel." The first stage in the volume rendering pipeline is to *classify* the volume data, which means to assign an opacity to each voxel. Opacity is the inverse of transparency: an opacity of 0.0 indicates a fully−transparent voxel, while an opacity of 1.0 indicates a voxel which completely occludes anything behind it. Intermediate values between 0.0 and 1.0 indicate semi−transparent voxels. The purpose of classification is to assign low opacities to regions of the data set which are uninteresting or distracting and high opacities to regions of the data set which should be visible in the rendering. Intermediate opacity values are used for smooth transitions from transparent to opaque regions, and for effects such as semi−transparent voxels which should not completely occlude objects behind them.

VolPack provides a classification method based on lookup tables. To use this method you specify a transfer function which maps the scalar data in a particular array element into the opacity for that element. Alternatively you can implement other classification techniques such as context−sensitive segmentation and then provide VolPack with a pre−classified volume.

The second rendering stage is to assign a color to each voxel, an operation which is called *shading* (or more precisely, *lighting*). VolPack includes support for the standard Phong shading equation. To use this shading technique, the volume data is preprocessed before rendering in order to compute a gradient vector for each voxel. The gradient vector can then be used as a pseudo surface normal to compute how light reflects off of each voxel. The user specifies the position and color of one or more light sources, and the reflective properties of the volume data. See *Computer Graphics: Principles and Practice* (Chapter 16, 2nd ed.), by Foley, van Dam, Feiner and Hughes, for a detailed discussion of the Phong shading equation. Alternative shading models can be implemented through a callback function.

The third rendering stage is to specify a view transformation and to transform the volume accordingly. This step can be as simple as choosing the position from which to look at the volume, or it can include an arbitrary affine transformation of the volume including non−uniform scaling and shearing. The view transformation also specifies how the volume is projected onto a 2D image plane.

The fourth and final rendering stage is to composite the voxels into an image. Digital compositing is analogous to the compositing process used in the film industry: several layers of semi−transparent film are merged together into a final image. VolPack provides several rendering algorithms that use different techniques to accelerate the compositing stage. The next subsection briefly describes the available algorithms.

Data Structures and Rendering Algorithms

VolPack includes three rendering algorithms which are useful in different situations. The algorithms differ in the degree to which they trade flexibility for speed and in the type of preprocessing required before rendering.

The fastest algorithm allows the user to rapidly render a volume with any view transformation and with any shading parameters while keeping the classification fixed. This algorithm relies on a special data structure which contains run−length encoded, classified volume data. Depending on the volume size it can take several minutes to precompute the run−length encoded volume, so this algorithm is most suitable when many renderings will be made from the same volume without changing the classification.

The steps when using this algorithm to render a classified volume are:

- load the volume data
- choose the classification function
- precompute the classified volume
- repeat:
  - set the view and shading parameters
  - render with `vpRenderClassifiedVolume()`

The second algorithm is useful in situations where the classification will be adjusted frequently. It also relies on a special data structure: a min−max octree which contains the minimum and maximum values of each voxel field. This data structure must be computed once when a new volume is acquired. The volume can then be rendered multiple times with any opacity transfer function, any view transformation and any shading parameters.

The steps when using this algorithm to render an unclassified volume are:

- load the volume data
- precompute the min−max octree with `vpCreateMinMaxOctree()`
- repeat:
  - choose the classification function
  - set the view and shading parameters
  - render with `vpRenderRawVolume()`

Finally, the third algorithm does not use any precomputed data structures. In most cases it is significantly slower than the other two algorithms and is useful only if you wish to make a single rendering from a volume. The steps for using this algorithm are identical to the previous algorithm except that there is no need to compute the min−max octree.

Section 2: Using VolPack

This section describes how to use the routines provided by VolPack. For more specific information about a particular routine, consult the man pages provided with the library.

Include Files and Libraries

All of the definitions needed by a program which uses VolPack are included in the header file `volpack.h`. The program must be compiled with the VolPack library by including the switch `-lvolpack` on the compilation command line. Other useful free libraries you may wish to use are John Ousterhout's Tcl/Tk libraries to build a graphical user interface, and Jef Poskanzer's pbmplus library or Sam Leffler's TIFF library to store images.

The header file defines the following data types:

- `vpContext`: a rendering context.
- `vpResult`: a result code.
- `vpVector3`: a three−element double−precision vector.
- `vpVector4`: a four−element double−precision vector.
- `vpMatrix3`: a three−by−three double−precision matrix.
- `vpMatrix4`: a four−by−four double−precision matrix.

Rendering Contexts

The first argument of most of the routines in the VolPack library is a *rendering context*, declared as a variable of type `vpContext`. A rendering context contains all of the information required to render a volume, including the classification and shading parameters, the view transformation, a description of the format of the volume, and private data structures used by the rendering routines. The contents of a rendering context are not directly accessible to the application programmer; instead, you use the routines provided by the library to set, modify and query the state in a context. A program can have multiple active contexts, for instance to render different volumes or to render the same volume with different parameters simultaneously.

To create a new context, use `vpCreateContext()`:

```
vpContext *vpCreateContext();
```

The return value is a pointer to the new context. It contains default values for most of the rendering parameters, but you can change all of them with the routines described later in this section.

To destroy a context and free the memory associated with it, use `vpDestroyContext()`:

```
void vpDestroyContext(vpContext *vpc);
```

Volumes

A volume is simply a 3D array of data. The type of data can be almost anything, but if you choose to use the classification and shading routines provided by VolPack then you must supply the fields these routines require. You may also wish to precompute information required by your shader or classifier and store it in the voxel. Here is an example layout for a voxel:

```
typedef unsigned char Scalar;
typedef unsigned short Normal;
typedef unsigned char Gradient;
typedef struct {
    Normal normal;
    Scalar scalar;
    Gradient gradient;
} Voxel;
```

In this example the data stored in a voxel includes an 8−bit scalar value and two precomputed fields. The first precomputed field is a surface normal vector encoded in a 16−bit field; this field is used by VolPack's shading routines. The second precomputed field is the gradient−magnitude of the scalar value; this field can be used for detecting surface boundaries during classification, for instance.

Note that the structure fields have been specified in the voxel structure in a very particular order. Many machines have alignment restrictions which require two−byte quantities to be aligned to two−byte boundaries, four−byte quantities to be aligned to four−byte boundaries, and so on. The compiler may have to insert wasted space in between fields to satisfy these requirements if you are not careful. Use the `sizeof()` operator to make sure the size of the voxel matches your expectations.

You should also place the fields which are required for shading first, followed by any other fields used only for classification. Ordering the fields this way makes it possible to store just the fields for shading when a classified volume is created for the fast rendering algorithm. This saves memory and improves cache performance.

Once you have decided on the format of your volume you must describe it to VolPack. To set the dimensions of the volume use `vpSetVolumeSize`:

```
vpResult
vpSetVolumeSize(vpContext *vpc, int xlen, int ylen, int zlen);
```

The first argument is the context whose state you wish to modify, and the remaining arguments are the number of elements in each dimension of the 3D volume array. The return value is a result code (type `vpResult`, which is an integer). The value VP_OK means the arguments are valid and the routine completed successfully. Other values indicate the type of error which occurred. See the man pages for the specific types of errors which can occur for each routine, or see the list of error codes in the Result Codes and Error Handling section.

Use `vpSetVoxelSize()` to declare the size of the voxel and the number of fields it contains:

```
vpResult
vpSetVoxelSize(vpContext *vpc, int bytes_per_voxel,
               int num_voxel_fields, int num_shade_fields,
               int num_classify_fields);
```

`Bytes_per_voxel` is the total size of a voxel in bytes. `Num_voxel_fields` is the number of fields in the voxel. `Num_shade_fields` is the number of fields required for shading. `Num_classify_fields` is the number of fields required for classification. The return value is a result code.

Continuing the earlier example, use the following call:

```
#define NUM_FIELDS          3
#define NUM_SHADE_FIELDS    2
#define NUM_CLASSIFY_FIELDS 2
vpSetVoxelSize(vpc, sizeof(Voxel), NUM_FIELDS, NUM_SHADE_FIELDS,
               NUM_CLASSIFY_FIELDS);
```

Now call `vpSetVoxelField()` and the `vpFieldOffset()` macro once for each field to declare its size and position in the voxel:

```
int
vpFieldOffset(void *voxel_ptr, field_name);

vpResult
vpSetVoxelField(vpContext *vpc, int field_num, int field_size,
                int field_offset, int field_max);
```

`Voxel_ptr` is a pointer to a dummy variable of the same type as your voxel, and `field_name` is the name of the voxel field. The return value of the macro is the byte offset of the field from the beginning of the voxel. `Field_num` is the ordinal index of the voxel field you are declaring, starting with 0 for the first field. `Field_size` is the size of the field in bytes. Use the `sizeof()` operator (e.g. `sizeof(voxel_ptr->field_name)`). `Field_offset` is the byte offset returned by `vpFieldOffset()`. `Field_max` is the maximum value of the quantity stored in the field. The return value is a result code.

Strictly speaking, the `vpSetVoxelField()` procedure must be called only for voxel fields which will be used by the VolPack classifier and shader. However, if you declare the other fields too then VolPack can automatically convert volumes that were created on machines with a different byte ordering. Only fields with size 1, 2 or 4 bytes can be declared with `vpSetVoxelField()`.

For the example voxel layout, make the following calls:

```
#define NORM_FIELD  0
#define NORM_MAX    VP_NORM_MAX
#define SCALAR_FIELD  1
```

```
#define SCALAR_MAX    255
#define GRAD_FIELD    2
#define GRAD_MAX      VP_GRAD_MAX
Voxel *dummy_voxel;
vpSetVoxelField(vpc, NORM_FIELD, sizeof(dummy_voxel->normal),
                vpFieldOffset(dummy_voxel, normal), NORM_MAX);
vpSetVoxelField(vpc, SCALAR_FIELD, sizeof(dummy_voxel->scalar),
                vpFieldOffset(dummy_voxel, scalar), SCALAR_MAX);
vpSetVoxelField(vpc, GRAD_FIELD, sizeof(dummy_voxel->gradient),
                vpFieldOffset(dummy_voxel, gradient), GRAD_MAX);
```

The constants `VP_NORM_MAX` and `VP_GRAD_MAX` are predefined by VolPack. In this example these fields will be computed using standard routines provided by the library.

To specify the volume data itself, use `SetRawVoxels()`:

```
vpResult
vpSetRawVoxels(vpContext *vpc, void *voxels, int size,
               int xstride, int ystride, int zstride);
```

`Voxels` is a pointer to the voxel data. `Size` is the number of bytes of voxel data. The remaining arguments are the strides in bytes for each of the three dimensions of the volume. For instance, `xstride` is the byte offset from the beginning of one voxel to the beginning of the next voxel along the x axis. Some of the VolPack routines operate faster if the volume is stored in z−major order (xstride < ystride < zstride), but it is not strictly necessary. If `voxels` is a pointer to dynamically allocated storage then the caller is responsible for freeing the memory at the appropriate time. VolPack does not free the voxel array when a context is destroyed. The data in the voxel array may be initialized or modified at any time, before or after calling `vpSetRawVoxels`.

Our running example continues as follows:

```
Voxel *volume;
unsigned size;
#define VOLUME_XLEN    256
#define VOLUME_YLEN    256
#define VOLUME_ZLEN    256
size = VOLUME_XLEN * VOLUME_YLEN * VOLUME_ZLEN * sizeof(Voxel);
volume = malloc(size);
vpSetRawVoxels(vpc, volume, size, sizeof(Voxel),
               VOLUME_XLEN * sizeof(Voxel),
               VOLUME_YLEN * VOLUME_XLEN * sizeof(Voxel));
```

VolPack provides a number of routines to help initialize some of the fields of the volume. If your input data consists of a three−dimensional array of 8−bit values and you wish to compute gradient−magnitude data or encoded normal vectors, then you can use `vpVolumeNormals()`:

```
vpResult
vpVolumeNormals(vpContext *vpc, unsigned char *scalars,
                int size, int scalar_field,
                int gradient_field, int normal_field);
```

`Scalars` is a pointer to the array of 8−bit values. `Size` is the size of the array in bytes. It must equal the number of voxels in the volume as previously specified with `vpSetVolumeSize()`. `Scalar_field`, `gradient_field` and `normal_field` are the voxel field numbers in which to store the scalar values from the array, the gradient−magnitudes of the scalar values, and the encoded surface normals respectively. Any of these field numbers may be equal to the constant `VP_SKIP_FIELD` if that item should not be stored in the volume. This function computes the specified fields and loads them into the volume array last specified with `vpSetRawVolume()`.

In our example, we can initialize the volume array as follows:

```
unsigned char *scalars;
scalars = LoadScalarData();
vpVolumeNormals(vpc, scalars, VOLUME_XLEN*VOLUME_YLEN*VOLUME_ZLEN,
                SCALAR_FIELD, GRAD_FIELD, NORM_FIELD);
```

`LoadScalarData()` might be a routine to load volume data from a file.

If your volume is large it may be inefficient to load all of the scalar data into one array and then copy it to the volume array. If this is the case then you can use `vpScanlineNormals()` to compute one scanline of the volume at a time:

```
vpResult
vpScanlineNormals(vpContext *vpc, int size,
```

```
                        unsigned char *scalars,
                        unsigned char *scalars_minus_y,
                        unsigned char *scalars_plus_y,
                        unsigned char *scalars_minus_z,
                        unsigned char *scalars_plus_z,
                        void *voxel_scan, int scalar_field,
                        int gradient_field, int normal_field);
```

`Size` is the length in bytes of one scanline of scalar data (which should equal the x dimension of the volume). `Scalars` points to the beginning of one scanline of scalars. `Scalars_minus_y` and `scalars_plus_y` point to the beginning of the previous and next scanlines in the y dimension, respectively. Similarly, `scalars_minus_z` and `scalars_plus_z` point to the beginning of the previous and next scanlines in the z dimension. These last four scanlines are the immediately−adjacent neighbors of the first scanline and are used to compute the gradient and surface normal vector. The next argument, `voxel_scan`, points to the scanline of the voxel array to write the result data into. The last three arguments are the voxel fields to write each type of data into and are identical to the corresponding arguments to `vpVolumeNormals()`. You can use `vpScanlineNormals()` in a loop which reads in the scalar data slice−by−slice, keeping at most three slices of data in memory at a time (in addition to the entire volume).

If you wish to compute normal vectors yourself but you still want to use the shading routines provided by VolPack, you can use `vpNormalIndex()` to encode a vector into the form expected by the shaders:

```
int vpNormalIndex(double nx, double ny, double nz);
```

The arguments are the components of the normal vector, which must be normalized (nx*nx + ny*ny + nz*nz == 1), and the return value is the 16−bit encoded normal. A routine is also provided to decode normals:

```
vpResult
vpNormal(int n, double *nx, double *ny, double *nz);
```

The encoded normal given by `n` is decoded, and the normal vector components are stored in the locations specified by the remaining arguments.

Classification

The classification routines provided by VolPack allow you to customize the opacity transfer function by specifying a collection of lookup tables. Each lookup table is associated with one voxel field. To classify a voxel, VolPack uses the value in each of the specified fields of the voxel to index the corresponding tables. The table values are then multiplied together to get the opacity of the voxel. The tables should contain numbers in the range 0.0−1.0 so that the final opacity is also in that range.

A lookup table is specified with `vpSetClassifierTable()`:

```
vpResult
vpSetClassifierTable(vpContext *vpc, int param_num, int param_field,
                     float *table, int table_size);
```

`Param_num` is the parameter number associated with the table you are declaring. The total number of tables must equal the `num_classify_fields` argument to `vpSetVoxelSize()`. The first table is numbered 0. `Param_field` is the number of the voxel field which should be used to index the table. `Table` is a pointer to the lookup table itself, and `table_size` is the size of the table in bytes (not the number of entries!) Note that even if `table` is dynamically allocated it is never deallocated by VolPack, even if the rendering context is destroyed. The data in the table may be initialized or modified at any time, before or after calling `vpSetClassifierTable`.

We could declare a two−parameter classifier for our example using the following calls:

```
float scalar_table[SCALAR_MAX+1];
float gradient_table[GRAD_MAX+1];
vpSetClassifierTable(vpc, 0, SCALAR_FIELD, scalar_table,
                     sizeof(scalar_table));
vpSetClassifierTable(vpc, 0, GRAD_FIELD, gradient_table,
                     sizeof(gradient_table));
```

VolPack provides a useful utility routine for initializing classification tables with piecewise linear ramps:

```
vpResult
vpRamp(float array[], int stride, int num_points,
       int ramp_x[], float ramp_y[]);
```

`Array` is the table to be initialized. `Stride` is the number of bytes from the start of one array element to the start of the next (useful if there are other fields in the array which you want to skip over). `Num_points` is the number of endpoints of the piecewise linear segments. `Ramp_x` is an array of x coordinates (table indices), and `ramp_y` is an array of y coordinates (values to store in the array). `vpRamp` linearly−interpolates values for the table entries in between the specified x coordinates.

For example, we can initialize our two classification tables as follows:

```
#define SCALAR_RAMP_POINTS 3
int scalar_ramp_x[] =    {  0,  24, 255};
float scalar_ramp_y[] =  {0.0, 1.0, 1.0};
vpRamp(scalar_table, sizeof(float), SCALAR_RAMP_POINTS,
       scalar_ramp_x, scalar_ramp_y);

#define GRAD_RAMP_POINTS 4
int grad_ramp_x[] =   {  0,   5,  20, 221};
float grad_ramp_y[] = {0.0, 0.0, 1.0, 1.0};
vpRamp(gradient_table, sizeof(float), GRAD_RAMP_POINTS,
       grad_ramp_x, grad_ramp_y);
```

If you wish to use an alternative classification algorithm instead of the lookup−table classifier then you should store the voxel opacities you compute in one of the fields of the voxel and define a lookup table which converts the values in that field into floating−point numbers. For instance, define a 1−byte opacity field which contains values in the range 0−255, and declare a lookup table with a linear ramp mapping those numbers to the range 0.0−1.0.

In addition to setting the classification function, you should also set the minimum opacity threshold with `vpSeti`. This threshold is used to discard voxels which are so transparent that they do not contribute significantly to the image. The higher the threshold, the faster the rendering algorithms. For example, to discard voxels which are at most 5% opaque, use the following:

```
vpSeti(vpc, VP_MIN_VOXEL_OPACITY, 0.05);
```

Classified Volumes

The fastest rendering algorithm provided by VolPack uses a run−length encoded volume data structure which must be computed before rendering. Three routines are provided to compute this data structure. Remember to set the opacity transfer function and the minimum voxel opacity before calling the functions in this subsection.

If you have already constructed an unclassified volume and defined the classification function as described in the previous subsections then use `vpClassifyVolume()`:

```
vpResult
vpClassifyVolume(vpContext *vpc);
```

This routine reads data from the currently−defined volume array, classifies it using the current classifier, and then stores it in run−length encoded form in the rendering context. The volume array is not modified or deallocated.

If you wish to load an array of 8−bit scalars and compute a classified volume directly without building an unclassified volume, then use `vpClassifyScalars()`:

```
vpResult
vpClassifyScalars(vpContext *vpc, unsigned char *scalars,
                  int size, int scalar_field,
                  int gradient_field, int normal_field);
```

The arguments to this routine are identical to those for `vpVolumeNormals()` described above. The difference between the two routines is that `vpClassifyScalars()` stores the result as a classified, run−length encoded volume instead of as an unclassified volume. The volume size, voxel size, voxel fields, and classifier must all be declared before calling this routine, but there is no need to call `vpSetRawVoxels()`.

If you wish to classify one scanline of voxel data at a time instead of loading the entire array of scalar data at once then use `vpClassifyScanline()`:

```
vpResult
vpClassifyScanline(vpContext *vpc, void *voxel_scan);
```

`Voxel_scan` is a pointer to one scanline of voxel data, in the same format as the full unclassified volume. You could, for instance, use `vpScanlineNormals()` to compute the fields of the scanline before passing it to `vpClassifyScanline()`. Each call to this routine appends one new scanline to the current classified volume. Out−of−order calls are not possible, and the volume cannot be rendered until all of the scanlines have been loaded.

Only one classified volume may be stored in a rendering context at a time. If you start classifying a new volume, any old classified volume data is deallocated. You can also force the current classified volume to be deallocated with `vpDestroyClassifiedVolume()`:

```
vpResult
vpDestroyClassifiedVolume(vpContext *vpc);
```

Note that if you change the contents of the unclassified volume array and you wish the classified volume to reflect those changes then you must call one of the routines in this section to recompute the classified volume.

Min−Max Octrees

A min−max octree is a hierarchical data structure which contains minimum and maximum values for each field used to index the classification tables. This data structure can be used to accelerate rendering unclassified volumes, and it can also accelerate the computation of a classified volume from an unclassified volume.

To compute a min−max octree, first define an unclassified volume with `vpSetVolumeSize()`, `vpSetVoxelSize()`, `vpSetVoxelField()`, and `vpSetRawVoxels()`. Also be sure to initialize the volume data. Now for each classification table make one call to `vpMinMaxOctreeThreshold()`:

```
vpResult
vpMinMaxOctreeThreshold(vpContext *vpc, int param_num, int range);
```

`Param_num` is the same parameter number you passed to `vpSetClassifierTable()`. `Range` is a range of table indices for this parameter which you consider to be "small". The opacity of a voxel should not vary much if the table index is changed by the amount specified in `range`. Choosing a value which is too small or too large may result in a reduced performance benefit during rendering. You may wish to experiment, but the octree should improve performance even if you don't use the optimum range value. You can use the routine `vpOctreeMask()` to visualize the effectiveness of the octree (see the man pages).

To compute the octree, call `vpCreateMinMaxOctree()`:

```
vpResult
vpCreateMinMaxOctree(vpContext *vpc, int root_node_size,
                     int base_node_size);
```

`Root_node_size` is currently not used but is reserved for future use. `Base_node_size` specifies the size in voxels of one side of the smallest node in the octree. The smaller the value, the better the resolution of the data structure at the expense of an increase in size. A value of 4 is a good starting point. This routine reads the data in the unclassified volume array, computes an octree, and stores it in the rendering context.

Once the octree has been computed it will be used automatically whenever you call `vpClassifyVolume()` or `vpRenderRawVolume()`. If you change the data in the volume array you MUST call `vpCreateMinMaxOctree` to recompute the octree, or else your renderings will be incorrect. You can also destroy the octree by calling `vpDestroyMinMaxOctree()`:

```
vpResult
vpDestroyMinMaxOctree(vpContext *vpc);
```

View Transformations

VolPack maintains four transformation matrices: a modeling transform, a viewing transform, a projection transform, and a viewport transform. The primary use of these matrices is to specify a transformation from the volume data's coordinate system to the image coordinate system. However, they also affect light direction vectors (and in future releases of the library they will affect the positioning of clipping planes and polygon primitives).

There are five coordinate systems implied by the transformation matrices: object coordinates, world coordinates, eye coordinates, clip coordinates, and image coordinates. In the object coordinate system the volume is entirely contained in a unit cube centered at the origin. The modeling transform is an affine transform which converts object coordinates into world coordinates. The modeling transform is also applied to light direction vectors to transform them to world coordinates. The view transform is an affine transform that converts world coordinates into eye coordinates. In eye coordinates the viewer is looking down the −Z axis. The view transform is typically used to specify the position of the viewer in the world coordinate system. The projection transform converts eye coordinates into clip coordinates. This transform may specify a perspective or a parallel projection, although perspective rendering is not yet supported. Finally, the viewport transform converts the clip coordinate system into image coordinates.

VolPack provides a number of routines to change the modeling matrix, viewing matrix and the projection matrix. First, use `vpCurrentMatrix()` to select the matrix you wish to modify:

```
vpResult
vpCurrentMatrix(vpContext *vpc, int option);
```

`Option` is one of the constants `VP_MODEL`, `VP_VIEW` or `VP_PROJECT`. Now use the following functions to modify the matrix contents (see the man pages for specifics):

`vpIdentityMatrix(vpContext *vpc)`
: Load the identity matrix into the current transformation matrix.

`vpTranslate(vpContext *vpc, double tx, double ty, double tz)`
: Multiply the current transformation matrix by a translation matrix.

`vpRotate(vpContext *vpc, int axis, double degrees)`
: Multiply the current transformation matrix by a rotation matrix. `Axis` is one of the constants `VP_X_AXIS`, `VP_Y_AXIS` or `VP_Z_AXIS`.

`vpScale(vpContext *vpc, double sx, double sy, double sz)`
: Multiply the current transformation matrix by a scaling matrix.

`vpMultMatrix(vpContext *vpc, vpMatrix4 m)`
: Multiply the current transformation matrix by the given matrix.

`vpSetMatrix(vpContext *vpc, vpMatrix4 m)`
: Load the given matrix into the current transformation matrix.

By default, all of the routines use post−multiplication. For instance, if the current modeling matrix is M and a rotation matrix R is applied, then the new transformation is M*R. If a light direction vector v is now specified (using commands discussed in the section on shading), it is transformed into M*R*v before it is stored in the current rendering context. If you prefer pre−multiplication of matrices then call `vpSeti` with the `CONCAT_MODE` argument. Note that vectors are always post−multiplied.

Two special routines are provided for creating projection matrices. These routines always store their result in the projection matrix, not the current matrix. The first is `vpWindow()`:

```
vpResult
vpWindow(vpContext *vpc, int type, double left, double right,
         double bottom, double top, double near, double far);
```

`Type` must be the constant `VP_PARALLEL` to specify a parallel projection. In a future release perspective projections will be allowed. The remaining arguments specify the left, right, bottom, top, near, and far coordinates of the planes bounding the view volume in eye coordinates. This routine works just like the `glFrustum()` and `glOrtho()` routines in OpenGL.

The second routine for creating a projection matrix uses the PHIGS viewing model:

```
vpResult
vpWindowPHIGS(vpContext *vpc, vpVector3 vrp, vpVector3 vpn,
         vpVector3 vup, vpVector3 prp, double viewport_umin,
         double viewport_umax, double viewport_vmin,
         double viewport_vmax, double viewport_front,
         double viewport_back, int type);
```

`Vrp` is the view reference point, `vpn` is the view plane normal, `vup` is the view up vector, `prp` is the projection reference point, the next six arguments are the bounds of the viewing volume in view reference coordinates, and `type` is the constant `VP_PARALLEL` to specify a parallel projection. Since these parameters specify a viewpoint as well as a viewing volume, typically the view matrix contains the identity. See *Computer Graphics: Principles and Practice* (Chapter 6, 2nd ed.), by Foley, van Dam, Feiner and Hughes for a complete discussion of the PHIGS viewing model.

The viewport transform is set automatically when you set the size of the image, which is discussed in the next subsection.

Here is an example showing all the steps to set the view transformation:

```
vpCurrentMatrix(vpc, VP_MODEL);
vpIdentityMatrix(vpc);
vpRotate(vpc, VP_X_AXIS, 90.0);
vpRotate(vpc, VP_Y_AXIS, 23.0);
vpCurrentMatrix(vpc, VP_VIEW);
vpIdentityMatrix(vpc);
vpTranslate(vpc, 0.1, 0.0, 0.0);
vpCurrentMatrix(vpc, VP_PROJECT);
vpWindow(vpc, VP_PARALLEL, -0.5, 0.5, -0.5, 0.5, -0.5, 0.5);
```

Note that light direction vectors are transformed according to the modeling matrix in effect at the time of the call to `vpSetLight`, and volumes are transformed according to the modeling matrix in effect at the time of rendering. The same viewing, projection and viewport transforms are applied to everything at the time of rendering.

Shading and Lighting

VolPack supports two shading methods: shading via lookup tables, and shading via callback functions. In addition, routines are provided to initialize shading tables for the Phong illumination model.

The built−in routines are designed to support the multiple−material voxel model described in *Volume Rendering* by Drebin, Carpenter and Hanrahan in Proceedings of SIGGRAPH 88. Each voxel is assumed to contain a mixture of basic material types. Each material type has its own shading parameters, such as color and shinyness. The color of a voxel is found by computing a color for each material type and then combining the colors in proportion to the fraction of each material in the voxel.

This functionality is implemented by storing two table indices in each voxel and using two lookup tables. One voxel field must contain an encoded surface normal vector as computed by `vpNormalIndex()`. This field is used to index a table which contains a color for each of the material types. The actual colors retrieved from the table depend on the surface normal, so directional lights can be implemented by storing appropriate values in the table. The second voxel field contains a value which is used to index the second table. Each row of the second table contains a fractional occupancy for each material type. These fractional occupancies are used as weights to determine the relative strength of each color retrieved from the first table.

To declare a lookup−table shader, use `vpSetLookupShader()`:

```
vpResult
vpSetLookupShader(vpContext *vpc, int color_channels,
                  int num_materials, int color_field,
                  float *color_table, int color_table_size,
                  int weight_field,
                  float *weight_table, int weight_table_size);
```

`Color_channels` is 1 for grayscale renderings or 3 for color (RGB) renderings. `Num_materials` is the number of material types. `Color_field` is the voxel field number for the color lookup table index. `Color_table` is the corresponding lookup table, and `color_table_size` is the size of the table in bytes. `Weight_field`, `weight_table` and `weight_table_size` are the field number, lookup table and table size for the second table which contains weights for each material type. The color table must be an array with the following dimensions:

```
float color_table[n][num_materials][color_channels];
```

where `n` is the number of possible values for the color field. The colors are values in the range 0.0−1.0 (zero intensity to full intensity). The weight table must be an array with the following dimensions:

```
float weight_table[m][num_materials];
```

where `m` is the number of possible values for the weight field. Weights are in the range 0.0−1.0. If there is only one material type then the weight table is not used and the corresponding parameters may be set to 0.

Returning to our example, the following code declares an RGB shader with two material types:

```
#define COLOR_CHANNELS   3
#define MATERIALS        2
float color_table[NORM_MAX+1][MATERIALS][COLOR_CHANNELS];
float weight_table[SCALAR_MAX+1][MATERIALS];
vpSetLookupShader(vpc, COLOR_CHANNELS, MATERIALS,
                  NORM_FIELD, color_table, sizeof(color_table),
                  SCALAR_FIELD, weight_table, sizeof(weight_table));
```

The weight table can be initialized using the `vpRamp()` function previously described, or using a loop which fills in values in whatever way you choose. To initialize the color table, VolPack provides a routine called `vpShadeTable()`. Before calling the routine you must set the lighting and shading parameters as follows.

To set the lighting parameters, use `vpSetLight()`:

```
vpResult
vpSetLight(vpContext *vpc, int light_num, int property,
           double n0, double n1, double n2);
```

`Light_num` is one of the constants `VP_LIGHT0`, `VP_LIGHT1`, ..., `VP_LIGHT5` and indicates which of the six light sources you wish to adjust. `Property` is either `VP_COLOR` or `VP_DIRECTION`. For `VP_COLOR` the remaining three arguments are the RGB components of the light color, in the range 0.0–1.0. For `VP_DIRECTION` the remaining three arguments are the x, y and z components of the direction of the light source. This vector is transformed by the current modeling matrix before it is stored in the rendering context (see View Transformations). You must also call `vpEnable()` to enable the light. By default, light 0 is enabled and all others are disabled.

For example, to create a cyan light coming from above the viewer's right shoulder, use the following:

```
vpSetLight(vpc, VP_LIGHT1, VP_COLOR,      0.0,  1.0,  1.0);
vpSetLight(vpc, VP_LIGHT1, VP_DIRECTION, -0.6,  0.6,  1.0);
vpEnable(vpc, VP_LIGHT1, 1);
```

You can also select "two–sided" lights using `vpEnable()` with the `VP_LIGHT_BOTH_SIDES` option. Under this lighting model each directional light shines in two directions, both in the specified direction and in the opposite direction.

To set the material parameters for a particular material type, call `vpSetMaterial()`:

```
vpResult
vpSetMaterial(vpContext *vpc, int material_num, int property,
              int surface_side, double r, double g, double b);
```

`Material_num` is one of the constants `VP_MATERIAL0`, `VP_MATERIAL1`, ..., `VP_MATERIAL5` and indicates which material you wish to adjust. `Property` is one of the following:

`VP_AMBIENT`
: Set the R, G and B ambient light reflection coefficients.

`VP_DIFFUSE`
: Set the R, G and B diffuse light reflection coefficients.

`VP_SPECULAR`
: Set the R, G and B specular light reflection coefficients.

`VP_SHINYNESS`
: Set the specular exponent. The `g` and `b` arguments are not used.

`Surface_side` is either `VP_EXTERIOR`, `VP_INTERIOR`, or `VP_BOTH_SIDES`. In the first case the parameters will only affect voxels on the "exterior" side of a surface, which by default means that the voxel's gradient points towards the viewer (you can use `vpEnable()` with the `VP_REVERSE_SURFACE_SIDES` option to reverse the meaning of exterior and interior). In the second case the parameters will only affect voxels whose gradient points away from the viewer. In the third case, all voxels are affected.

Here is an example which sets surface 0 to reflect red and green ambient and diffuse light, and to have fairly strong specular highlights which retain the color of the light source:

```
vpSetMaterial(vpc, VP_MATERIAL0, VP_AMBIENT,
              VP_BOTHSIDES, 0.1, 0.1, 0.0);
vpSetMaterial(vpc, VP_MATERIAL0, VP_DIFFUSE,
              VP_BOTHSIDES, 0.4, 0.4, 0.0);
vpSetMaterial(vpc, VP_MATERIAL0, VP_SPECULAR,
              VP_BOTHSIDES, 0.5, 0.5, 0.5);
vpSetMaterial(vpc, VP_MATERIAL0, VP_SHINYNESS,
              VP_BOTHSIDES, 10.0, 0.0, 0.0);
```

Now that all of the lighting and shading parameters have been set, the color lookup table has been declared with `vpSetLookupShader()`, and the viewing parameters have been set, you can call `vpShadeTable()` to recompute the entries of the lookup table:

```
vpResult
vpShadeTable(vpContext *vpc);
```

This routine computes all of the entries in the currently–defined color table using the current lighting and material parameters and the current view transformation. You should call `vpShadeTable()` after any changes to the shading or viewing parameters, but before calling any of the rendering routines.

If you wish to use some other shading model you have two options. One approach is to create your own routine to initialize the shading lookup tables. If you take this approach then you may define tables of any size (there is no need to use VolPack's encoded normal vectors). For example, you could use a color transfer function which assigns a unique color to each possible value of the scalar field in your volume. The second option is to define a callback routine which will be called to shade each voxel during rendering. You do so by calling `vpSetCallback()` instead of `vpSetLookupShader()`. For example, to declare a grayscale shading callback function use the following call:

```
void myshader();
vpSetCallback(vpc, VP_GRAY_SHADE_FUNC, myshader);
```

The function `myshader()` can do whatever you like to compute a color. See the man page for `vpSetCallback()` for more details. Using callback functions can lead to significant performance degradation during rendering.

There is one more shading option which is independent of the shading model you choose: depth cueing. Depth cueing allows you to introduce black "fog" which makes more distant voxels appear darker then voxels which are close to the viewer, thereby making it easier to distinguish foreground objects from background objects. To enable depth cueing call `vpEnable()`:

```
vpEnable(vpc, VP_DEPTH_CUE, 1);
```

You can use `vpSetDepthCueing()` to change the depth cueing parameters:

```
vpResult
vpSetDepthCueing(vpContext *vpc, double front_factor,
                 double density);
```

`Front_factor` is the transparency of the fog at the front plane of the viewing volume. It must be a positive number and it is usually less than 1.0 (although larger numbers can be used to brighten the foreground). `Density` controls the "density" of the fog, or how rapidly objects recede into darkness. The equation for the transparency of the fog is:

T = front_factor * exp(−density * depth)

where "depth" is 0 at the front plane of the viewing volume and 1 at the back plane. Each voxel color component is multiplied by the fog transparency during rendering.

VolPack also supports a fast one−pass shadow algorithm implemented with lookup tables (in a similar fashion to the procedure described above). See the man page for `vpSetShadowLookupShader`.

Images

The last step before rendering is to declare the array that VolPack should store the image into. Use `vpSetImage`:

```
vpResult
vpSetImage(vpContext *vpc, unsigned char *image, int width,
           int height, int bytes_per_scan, int pixel_type);
```

`Image` is a pointer to the array for the image. The next two arguments are the size of the image. These arguments also implicitly determine the viewport transformation: the clip coordinates are scaled to make the left, right, top and bottom planes of the viewing volume align with the sides of the image. The next argument is the number of bytes in one scanline of the image. This argument can be used to add padding to the end of each scanline in case the image display routines on your system impose alignment restrictions on the beginning of each scanline. Finally, the last argument is a code that specifies the format of the pixels in the image. The following formats are allowed:

`VP_ALPHA`
: opacity (1 byte/pixel)

`VP_LUMINANCE`
: grayscale color (1 byte/pixel)

`VP_LUMINANCEA`
: grayscale color plus opacity (2 bytes/pixel)

`VP_RGB`
: RGB color (3 bytes/pixel)

`VP_RGBA`
: RGB color plus opacity (4 bytes/pixel)

`VP_BGR`
: RGB color, byte−swapped (3 bytes/pixel)

`VP_ABGR`
: RGB color plus opacity, bytes−swapped (4 bytes/pixel)

Use the luminance formats only with grayscale shaders, and the RGB formats only with color shaders. The image should have dimensions:

```
unsigned char image[bytes_per_scan][height][bytes_per_pixel];
```

where `bytes_per_pixel` is the size of the pixel as determined by the pixel format.

Rendering

VolPack provides two rendering routines. The first routine is used to render pre−classified volumes which are created with the routines in the Classified Volumes subsection:

```
vpResult
vpRenderClassifiedVolume(vpContext *vpc);
```

This routine uses the current viewing and shading parameters to render the classified volume stored in the rendering context. The result is placed in the image buffer declared with `vpSetImage()`.

The second routine is used to render unclassified volumes which are created with the routines in the Volumes subsection:

```
vpResult
vpRenderRawVolume(vpContext *vpc);
```

This routine is identical to `vpRenderClassifiedVolume()` except that the source of the volume data is the raw volume data stored in the rendering context, and the volume data is classified on−the−fly during rendering. If a min−max octree data structure is present in the rendering context then it is used to accelerate rendering. However, even with the octree this routine is slower than `vpRenderClassifiedVolume()` because of the additional work which must be performed.

There is one important state variable which can be used to improve rendering performance: the maximum ray opacity threshold. During compositing, if the opacity of an image pixel reaches this threshold then no more voxel data is composited into the pixel. The threshold should be a number slightly less than one (0.95 is a good value), so that there is very little image degradation but voxels which do not make a significant contribution to the image can be skipped. You set the threshold with `vpSetd()` and the `VP_MAX_RAY_OPACITY` option. For example:

```
vpSetd(vpc, VP_MAX_RAY_OPACITY, 0.95);
```

State Variables

The previous subsections have described many routines which set state variables in a rendering context. This subsection briefly mentions the routines available to retrieve the values of these variables.

The function `vpGeti()` is used to retrieve integer state variables:

```
vpResult
vpGeti(vpContext *vpc, int option, int *iptr);
```

`Option` is a constant indicating the particular value you wish to get. The man page for `vpGeti` lists all of the options. The value is stored in the integer pointed to by `iptr`. As always, the return value of the routine is a result code (not the state variable value).

To retrieve floating−point state variables used `vpGetd()`:

```
vpResult
vpGetd(vpContext *vpc, int option, double *dptr);
```

This routine stores its result in the double pointed to by `dptr`. To retrieve pointers (e.g. the current raw volume data pointer) use `vpGetp`:

```
vpResult
vpGetp(vpContext *vpc, int option, void **pptr);
```

`Pptr` is a pointer to a pointer, so the value of the state variable is stored in `*ptr`. Transformation matrices can be retrieved with `vpGetMatrix()`:

```
vpResult
vpGetMatrix(vpContext *vpc, int option, vpMatrix4 m);
```

The matrix values are stored in `m`.

Lighting and material parameters can be retrieved with `vpGetLight()` and `vpGetMaterial()` which have arguments which are similar to the corresponding functions to set these parameters.

Utility Functions

VolPack provides a small collection of convenient utility functions. First, there are routines to store volume data structures in files and load them back into a rendering context. They allow you to perform all of the time−consuming preprocessing steps once and save the results in a file. See the man pages for the following routines:

- `vpStoreRawVolume()`
- `vpLoadRawVolume()`
- `vpStoreClassifiedVolume()`
- `vpLoadClassifiedVolume()`
- `vpStoreMinMaxOctree()`
- `vpLoadMinMaxOctree()`

The routine `vpExtract()` allows you to extract a rectangular solid region from either the raw volume data or the classified volume data. You can extract individual fields of the volume (e.g. just the scalar data), or computed values (e.g. opacity computed with the current classification function).

The routine `vpTranspose()` allows you to transpose the raw volume data. This can be useful to improve rendering performance for very large volumes. You can use `vpGeti` with the `VP_VIEW_AXIS` option to determine how the volume should be transposed for optimum performance given the current viewing parameters.

The routine `vpResample()` allows you to scale a volume to a different resolution using a variety of resampling filters. It is useful for scaling very large volumes down to a smaller size for fast previewing, or to filter low−resolution data sets to a higher resolution with a high−quality filter before rendering.

Result Codes and Error Handling

Almost all of the routines in VolPack return a result of type `vpResult` which is an integer. Routines return the value `VP_OK` to indicate success. Any other value indicates an error. See the man page for each function for the possible error codes and their specific meanings.

When an error occurs VolPack also records the error code in the rendering context. You can retrieve the error code later by calling `vpGetError()`. If another error occurs before you call `vpGetError()` then only the first one is returned. The recorded value is then reset.

The routine `vpGetErrorString()` can be used to convert an error code into a printable string.

Section 3: Tips and Pointers

Maximizing Rendering Speed

There are several techniques to keep in mind to get the maximum possible performance out of VolPack. First of all, use the appropriate rendering algorithm for the task at hand. If you want to render a volume from several viewpoints without changing the classification function then it is well worth the time to preprocess the volume into the run−length encoded format before rendering. Use the min−max octree data structure if the classification function does change for every rendering but the volume data remains fixed.

Second, choose the various thresholds carefully. Changing the minimum opacity threshold for classification and the maximum ray opacity for rendering can have a big impact on rendering speed. Changing the parameter range thresholds for the min−max octree can also improve performance.

Third, minimize the need for reallocating internal data structures by predeclaring their sizes. Internal buffers are used to store an intermediate image during rendering and a depth cueing lookup table. The sizes of these tables can change as the viewing parameters change, so the tables may have to be reallocated over the course of a multi−frame rendering loop. You can give VolPack "hints" for the sizes of these data structures using `vpSeti()` with the `VP_INT_WIDTH_HINT`, `VP_INT_HEIGHT_HINT` and `VP_DEPTH_CUE_SIZE_HINT` options.

Finally, if you are using `vpRenderRawVolume()` with a large volume then you may need to transpose the volume as the viewing direction changes from one principal axis to another.

Maximizing Image Quality

There are two important techniques which will help you to produce images free from distracting aliasing artifacts. The first is to choose classification functions that are fairly smooth. Functions with discontinuities or very abrupt transitions introduce very sharp transitions in the classified volume, and these transitions may be too sharp to be properly sampled. The result can be jagged boundaries and spurious patterns in the rendered image. These artifacts may be difficult to distinguish from the information in the data set. To diagnose this problem, try extracting slices of classified volume data with `vpExtract()` and check if the opacity images contain a lot of aliasing. Smooth transitions will produce the best images.

The second technique is to prefilter the volume data with a high−quality filter before scaling or zooming, rather than using the viewing transformation to do the scaling. There are two reasons that prefiltering may help. The rendering routines use a simple bilinear reconstruction filter, but if you prefilter you can use a higher−quality filter which does a better job of reconstruction. Furthermore, the resolution of the rendered image is limited by the number of samples in the volume, so very large magnification factors produce visible aliasing artifacts. Upscaling the volume with a high−quality filter before rendering can solve this problem. Several utility routines, described in the `vpResample()` man page, are provided for prefiltering a volume.

Software Support

If you have problems, bug reports or bug fixes, please send mail to:

volpack@graphics.stanford.edu

The author makes no commitment to fix bugs or provide support. However, future releases with fixes and enhancements are planned.

If you wish to be informed of future updates to the software then you should subscribe to the volpack−announce mailing list. To do so, send an email message to

majordomo@lists.stanford.edu

with the following message body:

subscribe volpack−announce

To be removed from the list, send the message:

unsubscribe volpack−announce

Mail will be sent to the list only to announce bug fixes and new releases.

If you like the library then drop us a note describing what you use it for!

Obtaining the Software

VolPack is available from the Stanford Computer Graphics Laboratory’s Web page (http://www−graphics.stanford.edu/software/volpack/#Distribution) or via anonymous ftp (ftp://www−graphics.stanford.edu/pub/volpack/).

Last update: 16 December 1994
volpack@graphics.stanford.edu