

# OpenGL Volumizer 2.7

## Reference Pages

Version 2.7

---

<a href="#"><u>vzAppearance</u></a>	Appearance description of a shape node.
<a href="#"><u>vzBlock</u></a>	Volumetric geometry representing an axis-aligned cuboid.
<a href="#"><u>vzClipRenderAction</u></a>	Render action for rendering 3D clip-textures.
<a href="#"><u>vzError</u></a>	Error logging and reporting facilities
<a href="#"><u>vzErrorSeverity</u></a>	Error severity used to log errors
<a href="#"><u>vzErrorType</u></a>	Error types used to print error messages
<a href="#"><u>vzExternalTextureFormat</u></a>	External format of texture data.
<a href="#"><u>vzGeometry</u></a>	Geometry of a shape node.
<a href="#"><u>vzGraphicsState</u></a>	A graphics state class.
<a href="#"><u>vzIndexArray</u></a>	An array of integral indices.
<a href="#"><u>vzInternalTextureFormat</u></a>	Internal format of textures.
<a href="#"><u>vzMemory</u></a>	Memory allocation and de-allocation routines.
<a href="#"><u>vzObject</u></a>	Reference counting and deletion notification.
<a href="#"><u>vzPTLUTShader</u></a>	Shader for volume rendering with lookup tables.
<a href="#"><u>vzPTRenderAction</u></a>	Projected Tetrahedra Render Action.
<a href="#"><u>vzParameter</u></a>	Shader parameter for a shape's appearance.
<a href="#"><u>vzParameterClipTexture</u></a>	Parameter representing a 3-dimensional clip-texture hierarchy.
<a href="#"><u>vzParameterLookupTable</u></a>	Parameter representing a lookup table.
<a href="#"><u>vzParameterVec3f</u></a>	Parameter representing a vector of three floating point values.
<a href="#"><u>vzParameterVertexData</u></a>	Parameter representing per vertex values.
<a href="#"><u>vzParameterVolumeTexture</u></a>	Parameter representing a 3-dimensional texture.
<a href="#"><u>vzPolyGeometry</u></a>	Polygonal geometry associated with a shape node.
<a href="#"><u>vzRenderAction</u></a>	Renderer for drawing shape nodes.
<a href="#"><u>vzShader</u></a>	Shader for generating a desired visual effect from an appearance.
<a href="#"><u>vzShape</u></a>	Container node for a volume's geometry and appearance.
<a href="#"><u>vzShapeSet</u></a>	A set of volumetric shapes.
<a href="#"><u>vzSlicePlaneSet</u></a>	A set of slicing planes.
<a href="#"><u>vzStructuredHexaMesh</u></a>	Volumetric geometry representing a structured hexhedral mesh.
<a href="#"><u>vzTMFragmentProgram</u></a>	Shader for using fragment programs.
<a href="#"><u>vzTMGradientShader</u></a>	Built-in gradient shader to be used with the vzTMRenderAction.
<a href="#"><u>vzTMISLShader</u></a>	TMShader using the Interactive Shading Language (ISL).

<a href="#"><u>vzTMLUTShader</u></a>	Shader for volume rendering 3D textures with lookup tables applied.
<a href="#"><u>vzTMRenderAction</u></a>	Texture-mapping render action.
<a href="#"><u>vzTMShader</u></a>	A general purpose shader to be used with the vzTMRenderAction.
<a href="#"><u>vzTMShaderData</u></a>	Shader data used by the vzTMShader class.
<a href="#"><u>vzTMSimpleShader</u></a>	Simple shader for volume rendering of 3D textures.
<a href="#"><u>vzTMTagShader</u></a>	Built-in tagging shader to be used with the vzTMRenderAction.
<a href="#"><u>vzTMTangentSpaceShader</u></a>	Shader for volume rendering 3D textures with lookup tables and gradientless lighting.
<a href="#"><u>vzTextureType</u></a>	Data types for texture data.
<a href="#"><u>vzUnstructuredHexaMesh</u></a>	Volumetric geometry representing an unstructured hexahedral mesh.
<a href="#"><u>vzUnstructuredMesh</u></a>	Unstructured volumetric geometry.
<a href="#"><u>vzUnstructuredTetraMesh</u></a>	Volumetric geometry representing an unstructured tetrahedral mesh.
<a href="#"><u>vzVertexArray</u></a>	An array of floating-point vertex coordinates.
<a href="#"><u>vzVolumeGeometry</u></a>	Volumetric geometry associated with a shape node.

**NAME**

**vzAppearance** - [Appearance description of a shape node.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/Appearance.h>
```

**PUBLIC METHOD SUMMARY**

```
vzAppearance (vzShader* shader);
void setParameter (const char* name, vzParameter* parameter);
vzParameter* getParameter (const char* name) const;
void setShader (vzShader* shader);
vzShader* getShader ( ) const;
vzParameter* getParameter (int index) const;
const char* getParameterName (int index) const;
int getNumParameters ( ) const;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzAppearance ( );
```

**CLASS DESCRIPTION**

The [vzAppearance](#) class describes how the shape should look when it is rendered. A [vzShader](#) must be specified, together with a set of shading parameters that affect the appearance. Each parameter associated with a [vzAppearance](#) has a name which is used by the [vzShader](#) to extract the corresponding values and take the appropriate actions. To get information about the specific shaders that are available for each render action, refer to the specific render action documentation: (e.g. [vzTMRRenderAction](#))

**EXAMPLES**

The following sample code shows how to use the [vzAppearance](#) class to describe the appearance of a [vzShape](#). The following example initializes an appearance using the built-in shader [vzTMLUTShader](#), which expects two parameters; One of type [vzParameterVolumeTexture](#) and name "volume" and two of type [vzParameterLookupTable](#) and name "lookup\_table"

```
// Create the shader
vzShader *shader = new vzTMLUTShader();

// Create the appearance
vzAppearance *appearance = new vzAppearance(shader);

// Create the volume texture
vzParameterVolumeTexture *texture = myInitVolumeTexture();

// Create the lookup table
vzParameterLookupTable *table = myInitLookupTable();

// Set the respective parameters for the appearance
appearance->setParameter("volume", texture);
appearance->setParameter("lookup_table", table);
```

## METHOD DESCRIPTIONS

### **vzAppearance()**

```
vzAppearance (vzShader* shader);
```

Appearance constructor. The *shader* specifies the the volumetric shader to be used to render the shape. For example, [vzTMLUTShader](#) is a built-in shader which provides volume rendering of 3D textures using lookup tables. The *shader* is ref-counted by the appearance in the constructor.

Passing a NULL shader to the constructor will cause a `vzError::error()`.

### **~vzAppearance()**

```
virtual ~vzAppearance ();
```

Appearance destructor. The shader and all the parameters for this appearance are unref'ed in the destructor.

### **getNumParameters()**

```
int getNumParameters () const;
```

Retrieve the number of parameters attached to the [vzAppearance](#).

### **getParameter()**

```
vzParameter* getParameter (const char* name) const;
```

Retrieve the parameter with *name*. If a parameter with *name* is not attached to the appearance, than a NULL is returned.

### **getParameter()**

```
vzParameter* getParameter (int index) const;
```

Retrieve a parameter by its index in the list. The first element in the list is the 0th element. A NULL pointer is returned if *index* is not in the range [0 .. numParameters-1]. This method can be useful for iterating through the list of parameters. For example -

```
// Get the number of parameters -
int numParams = appearance->getNumParameters();

// Now iterate through the parameters
for(int i = 0; i < numParams; i++) {

    // Get the individual parameter by index
    vzParameter *param = appearance->getParameter(i);

    // Get the parameter's name
    char *name = appearance->getParameterName(i);

    .....
}
```

### **getParameterName()**

```
const char* getParameterName (int index) const;
```

Retrieve a parameter name by its index in the list. The first element in the list is the 0th element. A NULL pointer is returned if *index* is not in the range [0 .. numParameters-1].

### **getShader()**

```
vzShader* getShader ( ) const;
```

Retrieve the shader attached to this appearance.

### setParameter()

```
void setParameter (const char* name, vzParameter* parameter);
```

Adds the given parameter to the list of parameters for this appearance. The *parameter* will be ref counted by the class. If a parameter with the given *name* already exists, then its reference count will be decremented and it will be replaced by the new one.

### setShader()

```
void setShader (vzShader* shader);
```

Sets the [vzShader](#) to be used with this appearance. The *shader* is ref-counted by the appearance and will be used for all future draw calls. Also, the initial shader's reference count will be decremented. The psuedocode for the method looks like -

```
void vzAppearance::setShader (vzShader *shader) {  
  
    // Ref the new shader  
    shader->ref();  
  
    // unref the old shader  
    currentShader->unref();  
  
    // set the old shader to the new one  
    currentShader = shader;  
}
```

### SEE ALSO

[vzAppearance](#), [vzObject](#), [vzParameterLookupTable](#), [vzParameterVolumeTexture](#), [vzShader](#), [vzShape](#), [vzTMLUTShader](#), [vzTMRenderAction](#)

---

[Back to Index](#)

### NAME

**vzBlock** - [Volumetric geometry representing an axis-aligned cuboid.](#)

### INHERITS FROM

[vzVolumeGeometry](#)

### HEADER FILE

```
#include <Volumizer2/Block.h>
```

### PUBLIC METHOD SUMMARY

```
vzBlock ( );  
void setOffsets (const float offsets[3]);  
void setDimensions (const float dims[3]);  
void getOffsets (float offsets[3]) const;  
void getDimensions (float dims[3]) const;  
vzUnstructuredTetraMesh* tessellate ( ) const;
```

### PROTECTED METHOD SUMMARY

```
virtual ~vzBlock ( );
```

### CLASS DESCRIPTION

*vzBlock* provides a simple abstraction for an axis aligned cuboid. This class is the simplest and one of the most commonly used volumetric geometry types in volume rendering applications. The cuboid is described by setting the offsets and dimensions along each of the X, Y and Z axes.

### METHOD DESCRIPTIONS

#### **vzBlock()**

```
vzBlock ( );
```

Constructor for [vzBlock](#). The [vzBlock](#) is constructed with default offset of (0, 0, 0) and dimensions of (1, 1, 1).

#### **~vzBlock()**

```
virtual ~vzBlock ( );
```

Destructor for the [vzBlock](#).

#### **getDimensions()**

```
void getDimensions (float dims[3]) const;
```

Get the dimensions of the [vzBlock](#).

#### **getOffsets()**

```
void getOffsets (float offsets[3]) const;
```

Get the offset of the [vzBlock](#) (the minimum x,y,z values.)

#### **setDimensions()**

```
void setDimensions (const float dims[3]);
```

Set the dimensions of the [vzBlock](#). The dimensions of the cuboid correspond to the lengths of each of its axes. The following function reduces the dimensions of the cuboid by half -

```
// Function to change the dimensions of the given vzBlock
void shrinkBlock(vzBlock *block) {

    // Dimensions
    float dimensions[3];

    // Get the current dimensions
    block->getDimensions(dimensions);

    // Divide the dimensions by half
    for(int i = 0; i < 3; i++)
        dimensions[i] /= 2;

    // Set the new dimensions
    block->setDimensions(dimensions);
}
```

### **setOffsets()**

```
void setOffsets (const float offsets[3]);
```

Set the offset of the [vzBlock](#) which correspond to the minimum X, Y and Z values for the geometry.

### **tessellate()**

```
vzUnstructuredTetraMesh* tessellate ( ) const;
```

Tessellate the [vzBlock](#) into a [vzUnstructuredTetraMesh](#). The tessellation generates a simple tetrahedral mesh with 5 tetrahedra and 8 vertices.

### **SEE ALSO**

[vzBlock](#), [vzUnstructuredTetraMesh](#), [vzVolumeGeometry](#)

---

[Back to Index](#)

**NAME**

**vzClipRenderAction** - [Render action for rendering 3D clip-textures.](#)

**INHERITS FROM**

[vzRenderAction](#)

**HEADER FILE**

```
#include <Volumizer2/ClipRenderAction.h>
```

**PUBLIC METHOD SUMMARY**

**Constructor/Destructor**

```
vzClipRenderAction ( );  
virtual ~vzClipRenderAction ( );
```

**Set methods**

```
void setSamplingRate (const float samplingRate[3]);  
void setLODThreshold (const float threshold);  
void setTextureMemorySize (const unsigned long texMemSize);  
void setMaxDownloadSize (const unsigned long downloadSize);  
void setMaxDrawSize (const unsigned long downloadSize);  
void enableMultiResolution ( );  
void disableMultiResolution ( );
```

**Get methods**

```
unsigned long getTextureMemorySize ( );  
unsigned long getMaxDownloadSize ( );  
unsigned long getMaxDrawSize ( );  
bool isMultiResolutionEnabled ( );  
int getMaxLevel (vzShape* shape);
```

**Performance related methods**

```
double getDownloadTime ( );  
double getDrawTime ( );  
unsigned long getDownloadSize ( );  
unsigned long getDrawSize ( );
```

**Manage/Unmanage methods**

```
virtual void manage (vzShape* shape);  
virtual void unmanage (vzShape* shape);
```

**Draw methods**

```
virtual void draw (vzShape* shape);  
virtual void draw (vzShapeSet* shapeSet);  
virtual void beginDraw (unsigned int rendererFlags);  
virtual void endDraw ( );  
void drawWireframe (vzShape* shape);  
void setRoam (const bool roaming);
```



## CLASS DESCRIPTION

`vzClipRenderAction` implements a renderer for shapes with 3D clip-textures. The renderer can also be used to render shapes with simple volume textures, in which case, its behaviour is similar to that of the [vzTMRenderAction](#). The renderer also provides an interface for rendering shape sets (see [vzShapeSet](#)) which can be comprised of multiple shapes.

The renderer uses the texture management system built inside the [vzTMRenderAction](#) to implement view-dependent rendering of clip-textures. It traverses the clip-texture hierarchy and supports both looking at the visible frustum along with roaming the data set using multiple levels-of-detail textures.

It uses intelligent texture paging mechanisms to improve user interaction and frame rates under certain conditions. Depending on the geometry for the shape, the renderer is capable of rendering the whole volume or a sub-cubical roaming window in the data. In this case, the render action culls all the bricks which do not intersect the shapes's geometry. It also performs view-frustum culling of bricks.

The renderer provides performance monitoring and control functionality in form of controlling the draw and texture download times. The application can explicitly set the LOD threshold values to perform resolution selection or the sampling rate for the polygonization process.

## NOTES

The following are some important issues to keep in mind while using the `vzClipRenderAction`:

- Shaders: When rendering 3D clip-textures, only single volume shaders provided by the `TMRenderAction` are currently supported, namely `vzTMLUTShader` and `vzTMTangentSpaceShader`. The `vzTMSimpleShader` shader cannot be used since the clip-texture uses LUTs to compensate for varying sampling rates.
- Roaming vs. Multi-resolution mode: The primary difference is that during roaming mode, only textures of the same resolution are rendered, while in multi-resolution mode, textures closer to the viewpoint have higher resolution and those further away have lower resolution. Roaming mode is more efficient when using a smaller geometry as a probe, while multi-resolution gives comparatively better image quality when using large geometry.
- Controlling the frame rate: Frame rate can be controlled by rendering low resolution textures during interaction with multi-resolution mode using LOD threshold value. Texture download rate can be controlled by limiting number of bricks which are downloaded during navigation with roaming mode. Total texture memory size can be limited to allow other application specific textures to be co-resident in texture memory.
- Rendering Volume textures and 3D Clip-Textures: `vzClipRenderAction` can be used to render shapes that have volume textures as well as clip-textures. The renderer pre-allocates the resources required to render the shapes with volume textures and then uses the remaining resources (like texture memory, draw size, download size, etc.) to do view-dependent rendering of the clip-textures.
- Rendering Shape sets: `vzClipRenderAction` provides an interface for directly rendering a collection of shapes ([vzShapeSet](#)) using the [draw\(\)](#) method. In this case, the renderer implements the manage/ unmanage/draw pipeline internal to the draw method.

## METHOD DESCRIPTIONS

### **vzClipRenderAction()**

```
vzClipRenderAction ( );
```

Constructor for the render action.

### **~vzClipRenderAction()**

```
virtual ~vzClipRenderAction ( );
```

Destructor.

### **beginDraw()**

```
virtual void beginDraw (unsigned int rendererFlags);
```

Sets the appropriate OpenGL state for drawing.

### **disableMultiResolution()**

```
void disableMultiResolution ( );
```

Disable multi-resolution mode rendering. When multi-resolution rendering is disabled, only texture bricks at the same level of resolution are drawn. The clip render action traverses the clip-hierarchy from the highest resolution to lower resolutions until it finds a clip-level which can be rendered in the available resources. This mode is usually more efficient when roaming the data set using a small geometry or region-of-interest, since in this mode the render action performs predictive texture downloads and provides finer control over the frame rate. Depending on the size of the data set, for large geometry, this mode will result in lower resolutions being rendered since the higher resolutions will exceed the available system resources. The amount of texture memory to be used can be modified using the [setTextureMemorySize\(\)](#) and [setMaxDrawSize\(\)](#) methods. Additionally, the total size of textures downloaded in a given frame can also be controlled using the [setMaxDownloadSize\(\)](#) method.

### **draw()**

```
virtual void draw (vzShape* shape);
```

Draw the given shape. The drawing will be done using the current center of interest and clip window of the clip texture. The renderer will however cull out the texture bricks which do not lie in the visible frustum of the current window or intersect the geometry for the shape. The draw method, sorts and renders the bricks in a back-to-front sorted order. The traversal scheme is governed by the mode being used to render the clip-texture (roaming vs. multi-resolution).

### **draw()**

```
virtual void draw (vzShapeSet* shapeSet);
```

Draw the given [vzShapeSet](#). The shape set consists of one or more [vzShapes](#) which will be rendered in a sorted order. When using this method, the render action takes control of managing, unmanaging and drawing the shapes in the shape in the correct visibility sorted order.

### **drawWireframe()**

```
void drawWireframe (vzShape* shape);
```

A debug utility method which draws the wireframe for the bounding boxes of the texture bricks being rendered in the current frame. Use this only as a debugging tool since it can add some rendering overhead to the application.

### **enableMultiResolution()**

```
void enableMultiResolution ( );
```

Enable multi-resolution mode rendering. When running in "multi-resolution" mode, the [vzClipRenderAction](#) will render the clip-texture using multiple levels of detail. In this case, the bricks which are closer to the viewpoint are rendered at the highest resolution possible within the available resources. The bricks which are further away will be rendered at a lower resolution. The resolution level can be controlled using the [setLODThreshold\(\)](#) method. The render action will only render the highest resolution bricks which fit in texture memory. The amount of texture memory to be used can be modified using the [setTextureMemorySize\(\)](#) and [setMaxDrawSize\(\)](#) methods. This mode will usually be useful when the geometry or the region-of-interest for the shape node is large, for example when rendering the whole volume data in a scene.

### **endDraw()**

```
virtual void endDraw ( );
```

Restores the OpenGL state if the `rendererFlags` were set appropriately.

### **getDownloadSize()**

```
unsigned long getDownloadSize ( );
```

Return the size of textures in bytes downloaded in the current frame.

### **getDownloadTime()**

```
double getDownloadTime ( );
```

Return the time taken in microseconds for texture download in the previous frame. This only an approximate value and would not be correct if the total texture being rendered in the frame exceed the size of texture memory.

### **getDrawSize()**

```
unsigned long getDrawSize ( );
```

Return the total size of textures in bytes rendered in the current frame.

### **getDrawTime()**

```
double getDrawTime ( );
```

Return the time taken in microseconds for the actual draw of the shapes. This would include the time for the polygonization and the polygon rendering.

### **getMaxDownloadSize()**

```
unsigned long getMaxDownloadSize ( );
```

Get the maximum amount of texture data that the render action is allowed to download in a given frame (see [setMaxDownloadSize\(\)](#)).

### **getMaxDrawSize()**

```
unsigned long getMaxDrawSize ( );
```

Get the maximum size of texture data that the render action is allowed to draw in a given frame (see [setMaxDrawSize\(\)](#)).

### **getMaxLevel()**

```
int getMaxLevel (vzShape* shape);
```

Returns the level for the highest resolution rendered in the previous frame if multi-resolution mode is disabled.

### **getTextureMemorySize()**

```
unsigned long getTextureMemorySize ( );
```

Get the total size of texture memory that the render action is currently allowed to use (see [setTextureMemorySize\(\)](#)).

### **isMultiResolutionEnabled()**

```
bool isMultiResolutionEnabled ( );
```

Returns true if the multi-resolution mode rendering is enabled and vice-versa.

### **manage()**

```
virtual void manage (vzShape* shape);
```

Manage the given *shape*. Currently, the renderer can manage and render only one shape. Hence, an attempt to manage a different shape would result in overwriting the previous shape. In order to render multiple clip-textures, currently you would need to create multiple render actions and divide the texture memory resources appropriately among them.

### **setLODThreshold()**

```
void setLODThreshold (const float threshold);
```

Set the threshold value for selecting the level-of-detail textures to be rendered. Before selecting a brick to be rendered, the value *threshold* is used as a factor in comparing the brick dimensions to it's screen space projection size. Lower values imply rendering the higher resolution textures closer to the viewpoint and higher values imply rendering the coarse resolution textures. The default value for the threshold value is 5.0.

### **setMaxDownloadSize()**

```
void setMaxDownloadSize (const unsigned long downloadSize);
```

Set an upper bound on the total size of textures that renderer can download in a given frame, i.e. a beginDraw/endDraw pair. This constraint, in conjunction with the LOD threshold, helps improve the interactivity of the render action by reducing the time spend in downloading textures. Setting a low value might result in rendering of lower resolution textures. This maximum texture download limit is effective only when rendering in roaming mode. The default value is 1/8th of the total texture memory size.

### **setMaxDrawSize()**

```
void setMaxDrawSize (const unsigned long downloadSize);
```

Set an upper bound on the total size of textures that the renderer can draw in a given frame. This constraint, in conjunction with the LOD threshold, helps improve the interactivity of the render action by reducing the time spend in drawing textures. Setting a low value might result in rendering of lower resolution textures. The default value is equal to the texture memory size.

### **setRoam()**

```
void setRoam (const bool roaming);
```

Deprecated method: Use [enableMultiResolution\(\)](#) and [disableMultiResolution\(\)](#) to turn on/off multi-resolution modes.

### **setSamplingRate()**

```
void setSamplingRate (const float samplingRate[3]);
```

Set a sampling rate for volume rendering, defined in texture space. The default sampling rate of (1.0, 1.0, 1.0) would sample once per voxel in each texture dimension. If the application modifies this value, it should also modify the alpha component of the lookup tables, if any, to compensate for the change in the sampling density of the volume data.

### **setTextureMemorySize()**

```
void setTextureMemorySize (const unsigned long texMemSize);
```

Set the total size of texture memory that the render action is allowed to use to store texture data. If not set, the render action would try to infer the texture memory size on the graphics subsystem and by default use all of the texture memory.

### **unmanage()**

```
virtual void unmanage (vzShape* shape);
```

Unmanage the given *shape*.

## **SEE ALSO**

[vzClipRenderAction](#), [vzRenderAction](#), [vzShape](#), [vzShapeSet](#), [vzTMRenderAction](#)

**NAME****vzError** - [Error logging and reporting facilities](#)**HEADER FILE**

#include &lt;Volumizer2/Error.h&gt;

**PUBLIC METHOD SUMMARY****Logging Errors**

```
static void error (vzErrorType error, const char* format=NULL, ...);
static void warn (vzErrorType error, const char* format=NULL, ...);
static void log (vzErrorSeverity severity, vzErrorType error, const char* format=NULL, ...);
```

**Printing Debug Messages**

```
static void message (int debugLevel, const char* format, ...);
static void setDebugLevel (int debugLevel);
static int getDebugLevel ( );
```

**Querying Last Error**

```
static void clear ( );
static vzErrorType getError ( );
static vzErrorSeverity getSeverity ( );
```

**Changing Error Handler**

```
static void setHandler (ErrorHandler* handler, void* data=NULL);
static void getHandler (ErrorHandler*& handler, void*& data);
```

**CLASS DESCRIPTION**

This class implements a mechanism for logging errors. It consists of a collection of static methods that allow a user-defined error routine to be setup to handle all logged errors. The default error handler simply prints out the error message and then calls [abort\(\)](#) if the error severity is VZ\_ERROR. The error handler installed applies to all threads.

Regardless of the error handler in effect, the first error encountered will be recorded and can be queried later using [getError\(\)](#). The [clear\(\)](#) can be used to reset the saved error to VZ\_NO\_ERROR. Errors are recorded and cleared on a per-thread basis.

Each problem is classified as either an error or a warning using the enum values VZ\_ERROR or VZ\_WARNING, respectively.

The [vzError::message\(\)](#) method produces debug messages that are neither errors nor warnings. Each debug message is given a particular debug level, passed as an integer parameter to the [message\(\)](#) method. The message will be output to stderr only if the debug level of the message is less than or equal to the current debug level (see [setDebugLevel\(\)](#)). Therefore, the higher you set this debug level, the more debug information will be printed.

**METHOD DESCRIPTIONS****clear()**

```
static void clear ( );
```

This method clears the saved error returned by [getError\(\)](#) back to the initial state of VZ\_NO\_ERROR.

**error()**

```
static void error (vzErrorType error, const char* format=NULL, ...);
```

This convenience method is used by the library to log errors. It is equivalent to calling [log\(\)](#) with the severity passed in as VZ\_ERROR. The default behavior is to print the error message and then call abort(). The following demonstrates the use of this method -

```
// Check for NULL pointers
void checkForNull(void *pointer) {
    if(pointer == NULL)
        vzError::error(VZ_INVALID_VALUE, "NULL pointer");
}
```

### **getDebugLevel()**

```
static int getDebugLevel ( );
```

Retrieve the current debug level.

### **getError()**

```
static vzErrorType getError ( );
```

This method returns the first error logged since the calling thread started or last called [clear\(\)](#). Error codes are stored independently for each thread. If there have been no errors then VZ\_NO\_ERROR will be returned.

### **getHandler()**

```
static void getHandler (ErrorHandler*& handler, void*& data);
```

This method returns the values last passed to [setHandler\(\)](#) in *handler* and *data*.

### **getSeverity()**

```
static vzErrorSeverity getSeverity ( );
```

This method returns the severity associated with the error returned by [getError\(\)](#).

### **log()**

```
static void log (vzErrorSeverity severity, vzErrorType error, const char* format=NULL, ...);
```

This method is used by the library to log errors (and warnings). The actual handling of the error can be controlled using [setHandler\(\)](#). The severity level and type of error is indicated by the *severity* and *error* parameters. In addition a message may be encoded using the *format* parameter and any [printf\(3\)](#) style optional arguments that follow it.

Errors are classified as either an error or a warning using the enum values VZ\_ERROR or VZ\_WARNING respectively. These severity levels are intended as guidelines to an error handler on whether execution should continue after the error is reported.

### **message()**

```
static void message (int debugLevel, const char* format, ...);
```

This method prints the specified message to stderr, assuming that *debugLevel* is less than or equal to the current debug level (set by [setDebugLevel\(\)](#)). The debug level can also be set by setting the environment variable VOLUMIZER\_DEBUG\_LEVEL to the appropriate value.

### **setDebugLevel()**

```
static void setDebugLevel (int debugLevel);
```

Set the current debug level. All calls to [message\(\)](#) with a debug level above the current level will be ignored.

## setHandler()

```
static void setHandler (ErrorHandler* handler, void* data=NULL);
```

This method is used to define an error handler routine that will be called whenever an error is logged. The function pointed to be *handler* must have the prototype:

```
void handler(vzErrorSeverity severity, vzErrorType error,  
            const char *format, va_list args, void* data);
```

The handler is passed the severity level in *severity*, the actual error code in *error*, and an optional message to be formatted using a routine like [vsprintf\(3\)](#) in *format* and *args*. The final parameter *data* is the same value that was passed to this routine in the parameter of the same name.

The following example adds user defined error handler to be used for the vzError class.

```
// Set the error handler for vzError::log()  
vzError::setHandler (myHandler, NULL);
```

The error handler might look like the following -

```
// User defined error handler  
static void myHandler(vzErrorSeverity severity, vzErrorType type,  
                    const char *format, va_list args, void* data)  
{  
    if(severity == VZ_ERROR)  
        cerr<<"myHandler::Error!!!";  
    else if(severity == VZ_WARNING)  
        cerr<<"myHandler::Warning!!!";  
  
    // Print the error message  
    fprintf(stderr, format, args);  
  
    // Use the vzErrorType to do whatever else is needed!!!  
    .....  
}
```

## warn()

```
static void warn (vzErrorType error, const char* format=NULL, ...);
```

This convenience method is used by the library to log warnings. It is equivalent to calling [log\(\)](#) with the severity passed in as VZ\_WARNING. The default handler just prints an error message and returns.

```
// Check for equal values  
void areSame(int value1, int value2 ) {  
    if(value1 == value2)  
        vzError::warn(VZ_UNSPECIFIED_ERROR, "Both indices are same");  
}
```

**SEE ALSO**

[printf\(3\)](#), [vsprintf\(3\)](#)

---

[Back to Index](#)



### NAME

**vzErrorSeverity** - [Error severity used to log errors.](#)

### HEADER FILE

```
#include <Volumizer2/Error.h>
```

### CLASS DESCRIPTION

Enumerated type representing the severity of an error that might occur. This is used by the [vzError](#) class.

```
enum vzErrorSeverity {  
    VZ_ERROR = 0,           // serious error, probably can't continue  
    VZ_WARNING = 1         // warning, probably OK to continue  
};
```

### SEE ALSO

[vzError](#)

---

[Back to Index](#)

### NAME

**vzErrorType** - [Error types used to print error messages.](#)

### HEADER FILE

```
#include <Volumizer2/Error.h>
```

### CLASS DESCRIPTION

Enumerated type representing the types of errors that might have occurred. Depending on the type of error, a message string is printed with the corresponding error() or warn() message. Use VZ\_UNSPECIFIED\_ERROR for errors which are not of any of the following types.

```
enum vzErrorType {
    VZ_NO_ERROR = 0,           // no error
    VZ_INVALID_ERROR = 1,     // invalid error
    VZ_INVALID_ENUM = 2,      // invalid enum provided
    VZ_INVALID_VALUE = 3,     // invalid value supplied
    VZ_INVALID_OPERATION = 4, // operation illegal in current state
    VZ_OUT_OF_MEMORY = 5,     // not enough memory left
    VZ_OUT_OF_RESOURCE = 6,   // some resource is exhausted
    VZ_OS_FAILURE = 7,        // some system service failed
    VZ_UNIMPLEMENTED = 8,     // some feature is unimplemented
    VZ_UNSPECIFIED_ERROR = 9, // unspecified error
    VZ_MAX_ERROR = 9          // maximum valid error code
};
```

---

[Back to Index](#)

### NAME

**vzExternalTextureFormat** - [External format of texture data.](#)

### HEADER FILE

```
#include <Volumizer2/VolEnums.h>
```

### CLASS DESCRIPTION

Enumerated type representing the external format of texture data. This is the format of the data that is supplied to the API, not the internal format of the data stored in texture memory. Used by the `vzParameterVolumeTexture` and `vzParameterLookupTable` classes.

```
enum vzExternalTextureFormat {  
    VZ_RGBA           = GL_RGBA,  
    VZ_RGB            = GL_RGB,  
    VZ_LUMINANCE_ALPHA = GL_LUMINANCE_ALPHA,  
    VZ_LUMINANCE      = GL_LUMINANCE  
};
```

---

[Back to Index](#)

## NAME

**vzGeometry** - [Geometry of a shape node.](#)

## INHERITS FROM

[vzObject](#)

## HEADER FILE

```
#include <Volumizer2/Geometry.h>
```

## PUBLIC METHOD SUMMARY

```
vzGeometry ( );  
void getBoundingBox (float bbox[6]);
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzGeometry ( ) = 0;  
void setBoundingBox (float bbox[6]);
```

## CLASS DESCRIPTION

The *vzGeometry* class represents any geometry attached to a [vzShape](#) node. The geometry of a shape node is used to describe the spatial attributes of the node.

## METHOD DESCRIPTIONS

### **vzGeometry()**

```
vzGeometry ( );
```

Constructor for geometry. The bounding box of the geometry is initialized illegal values inside the constructor. This forces a [computeBoundingBox\(\)](#) on calling the [getBoundingBox\(\)](#) method.

### **~vzGeometry()**

```
virtual ~vzGeometry ( ) = 0;
```

Destructor for geometry.

### **getBoundingBox()**

```
void getBoundingBox (float bbox[6]);
```

A method to compute the bounding box of the given geometry data. The bounding box is computed and coordinates are copied into *bbox*. The first three values give the x, y and z coordinates of the lower bound and the last three values give the coordinates of the upper bound. The computed bounding box is cached internally and hence subsequent calls to [getBoundingBox](#) return the cached value.

### **setBoundingBox()**

```
void setBoundingBox (float bbox[6]);
```

Set the bounding box for the geometry class. This is a protected method which is used by derived classes to update the bounding box for the geometry class whenever the bounding box needs to be computed.

## SEE ALSO

[vzObject](#), [vzShape](#)

---

[Back to Index](#)

**NAME**

**vzGraphicsState** - [A graphics state class.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/GraphicsState.h>
```

**PUBLIC METHOD SUMMARY**

```
vzGraphicsState ( );
void setModelviewMatrix (const GLdouble modelMat[16]);
void setProjectionMatrix (const GLdouble projMat[16]);
void getModelviewMatrix (GLdouble modelMat[16]);
void getProjectionMatrix (GLdouble projMat[16]);
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzGraphicsState ( );
```

**CLASS DESCRIPTION**

[vzGraphicsState](#) provides an encapsulation for graphics state associated with volumetric shapes. The class provides an interface for applications to pass the necessary viewing and other graphics state related information to other Volumizer objects. Currently, the graphics state associated with a [vzShapeSet](#) is used by the [vzParameterClipTextures](#) in the set to update their center of interest when the view direction changes.

**METHOD DESCRIPTIONS****vzGraphicsState()**

```
vzGraphicsState ( );
```

Constructor for a graphics state object. By default, all state variables are uninitialized, meaning that a request to retrieve them causes the invocation of a `glGet(...)` function. Once a state variable is set by a user, it is used directly by the object associated with this graphics state.

**~vzGraphicsState()**

```
virtual ~vzGraphicsState ( );
```

Destructor for a `vzGraphicsState` object.

**getModelviewMatrix()**

```
void getModelviewMatrix (GLdouble modelMat[16]);
```

Get the current modelview matrix. If it has previously been initialized using [setModelviewMatrix\(\)](#), then the cached value will be retrieved. Otherwise, the latest modelview matrix entries will be retrieved using the OpenGL `glGetDoublev(GL_MODELVIEW_MATRIX)` command.

**getProjectionMatrix()**

```
void getProjectionMatrix (GLdouble projMat[16]);
```

Get the current projection matrix. If it has previously been initialized using [setProjectionMatrix\(\)](#), then this cached value will be retrieved. Otherwise, the latest projection matrix entries will be retrieved using the OpenGL `glGetDoublev(GL_PROJECTION_MATRIX)` command.

**setModelviewMatrix()**

```
void setModelviewMatrix (const GLdouble modelMat[16]);
```

Set the modelview matrix. The viewing information associated with the object will be updated accordingly.

**setProjectionMatrix()**

```
void setProjectionMatrix (const GLdouble projMat[16]);
```

Set the projection matrix. The viewing information associated with the object will be updated accordingly.

**SEE ALSO**

[vzGraphicsState](#), [vzObject](#), [vzParameterClipTexture](#), [vzShapeSet](#)

---

[Back to Index](#)

**NAME**

**vzIndexArray** - [An array of integral indices.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/IndexArray.h>
```

**PUBLIC METHOD SUMMARY**

```
vzIndexArray (int numIndices, int* indices);
void setDataPtr (int numIndices, int* indices);
int* getDataPtr ( ) const;
int getNumIndices ( ) const;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzIndexArray ( );
```

**CLASS DESCRIPTION**

*vzIndexArray* provides an abstraction to store an array of integer indices. Used in geometry classes where each primitive is represented by a list of indices which point to entries in a [vzVertexArray](#).

**METHOD DESCRIPTIONS****vzIndexArray()**

```
vzIndexArray (int numIndices, int* indices);
```

Constructor for [vzIndexArray](#). Calls [setDataPtr\(\)](#) with *numIndices* and *indices* as arguments.

**~vzIndexArray()**

```
virtual ~vzIndexArray ( );
```

Destructor for [vzIndexArray](#).

**getDataPtr()**

```
int* getDataPtr ( ) const;
```

Return the pointer to the array of indices.

**getNumIndices()**

```
int getNumIndices ( ) const;
```

Return the number of indices.

**setDataPtr()**

```
void setDataPtr (int numIndices, int* indices);
```

[setDataPtr](#) performs a shallow copy of *indices*: no memory allocation is performed. You **must** call this method whenever you modify index data to let Volumizer know of the change. You might want to do something like -

```
// A function which modifies the index values of an index array.
void modifyIndexValues(vzIndexArray *indices) {
```



```
// Get the index values of the array
int *indexValues = indices->getDataPtr();

// Get the number of indices in the array
int numIndices = indices->getNumIndices();

// Modify the value of a particular index
indexValues[someIndex] = someValue;

// Call setDataPtr() to notify the API of the change
indices->setDataPtr(numIndices, indexValues);
}
```

**SEE ALSO**

[vzIndexArray](#), [vzObject](#), [vzVertexArray](#)

---

[Back to Index](#)

**NAME**

**vzInternalTextureFormat** - [Internal format of textures.](#)

**HEADER FILE**

```
#include <Volumizer2/VolEnums.h>
```

**CLASS DESCRIPTION**

Enumerated type representing the internal format of textures. Used by the vzParameterVolumeTexture class. Specifying a default value for this type allows the respective render action to infer the appropriate internal format from the external texture format and type.

```
enum vzInternalTextureFormat {
    VZ_DEFAULT_INTERNAL_FORMAT      = -1,
    VZ_INTENSITY8                   = GL_INTENSITY8,
    VZ_INTENSITY12                  = GL_INTENSITY12,
    VZ_RGBA4                        = GL_RGBA4,
    VZ_RGBA5                        = GL_RGBA5,
    VZ_INTENSITY16                  = GL_INTENSITY16,
    VZ_LUMINANCE8_ALPHA8           = GL_LUMINANCE8_ALPHA8,
    VZ_LUMINANCE12_ALPHA4         = GL_LUMINANCE12_ALPHA4,
    VZ_RGBA4                        = GL_RGBA4,
    VZ_RGBA5_A1                    = GL_RGBA5_A1,
    VZ_LUMINANCE12_ALPHA12        = GL_LUMINANCE12_ALPHA12,
    VZ_RGBA8                        = GL_RGBA8,
    VZ_RGBA10_A2                   = GL_RGBA10_A2,
    VZ_RGBA12                       = GL_RGBA12,
    VZ_RGBA16                       = GL_RGBA16,
    VZ_DUAL_INTENSITY8             = GL_DUAL_INTENSITY8_SGIS,
    VZ_DUAL_LUMINANCE_ALPHA4       = GL_DUAL_LUMINANCE_ALPHA4_SGIS,
    VZ_QUAD_INTENSITY4             = GL_QUAD_INTENSITY4_SGIS,
    VZ_INTENSITY_FLOAT16           = GL_INTENSITY_FLOAT16_ATI,
    VZ_LUMINANCE_ALPHA_FLOAT16     = GL_LUMINANCE_ALPHA_FLOAT16_ATI,
    VZ_RGBA_FLOAT16                = GL_RGBA_FLOAT16_ATI,
    VZ_INTENSITY_FLOAT32           = GL_INTENSITY_FLOAT32_ATI,
    VZ_LUMINANCE_ALPHA_FLOAT32     = GL_LUMINANCE_ALPHA_FLOAT32_ATI,
    VZ_RGBA_FLOAT32                = GL_RGBA_FLOAT32_ATI,
};
```

## NAME

**vzMemory** - [Memory allocation and de-allocation routines.](#)

## HEADER FILE

```
#include <Volumizer2/Memory.h>
```

## PUBLIC METHOD SUMMARY

```
static void operator delete (void* ptr);
static void* operator new (size_t nbytes);
static void setMemoryManagementCallbacks (vzMemoryAllocationCallback allocateCB,
                                           vzMemoryDeletionCallback deleteCB, void* userData);
static void getMemoryManagementCallbacks (vzMemoryAllocationCallback& allocateCB,
                                           vzMemoryDeletionCallback& deleteCB, void*& userData);

static void* malloc (size_t nbytes);
static void free (void* ptr);
```

## PROTECTED METHOD SUMMARY

```
vzMemory ( );
virtual ~vzMemory ( );
```

## CLASS DESCRIPTION

This class provides arbitrary user-defined memory management functionality. `vzMemory` is the base class for all object classes in Volumizer. By installing memory allocation and deletion callbacks, the user can override the default operators `new()` and `delete()`. This could be useful in allocating objects in shared memory, for example.

## METHOD DESCRIPTIONS

### **vzMemory()**

```
vzMemory ( );
```

Constructor.

### **~vzMemory()**

```
virtual ~vzMemory ( );
```

Destructor.

### **free()**

```
static void free (void* ptr);
```

Deallocates the data pointer *ptr* using the memory deletion callback.

### **getMemoryManagementCallbacks()**

```
static void getMemoryManagementCallbacks (vzMemoryAllocationCallback& allocateCB,
                                           vzMemoryDeletionCallback& deleteCB, void*& userData);
```

Use this method to get the user-defined memory management functions.

### **malloc()**

```
static void* malloc (size_t nbytes);
```

Allocates *nbytes* of memory using the memory allocation callback and returns pointer to the allocated memory.

### **operator delete()**

```
static void operator delete (void* ptr);
```

Memory deletion operator for object classes in Volumizer. The behavior of the operator is dependent upon the deletion callback specified by [setMemoryManagementCallbacks\(\)](#). The default behavior is to call the `free()` function.

### **operator new()**

```
static void* operator new (size_t nbytes);
```

Memory allocation operator for object classes in Volumizer. The behavior of the operator is dependent upon the allocation callback specified by [setMemoryManagementCallbacks\(\)](#). The default behavior is to call the `malloc()` function.

### **setMemoryManagementCallbacks()**

```
static void setMemoryManagementCallbacks (vzMemoryAllocationCallback allocateCB,  
                                           vzMemoryDeletionCallback deleteCB, void* userData);
```

Use this method to specify arbitrary user-defined memory management functions. The parameters specify a memory allocation callback, a memory deletion callback, and a user data pointer. The callbacks are invoked within the [operator new\(\)](#) and [operator delete\(\)](#) methods, respectively.

```
typedef void * (*vzMemoryAllocationCallback) (size_t nbytes, void *userData);  
typedef void (*vzMemoryDeletionCallback) (void *dataPtr, void *userData);
```

The following method sets the memory allocation and de-allocation routines to use the [mpkMalloc\(\)](#) and [mpkFree\(\)](#) functions respectively.

```
// Set the allocation and de-allocation callback functions  
vzMemory::setMemoryManagementCallbacks(allocate, deallocate, NULL);  
  
// The allocator callback function  
void *allocate(size_t size, void *userData) {  
    return mpkMalloc(size);  
}  
  
// The de-allocator callback function  
void deallocate(void *pointer, void *userData) {  
    mpkFree(pointer);  
}
```

## NAME

**vzObject** - [Reference counting and deletion notification.](#)

## INHERITS FROM

[vzMemory](#)

## HEADER FILE

```
#include <Volumizer2/Object.h>
```

## PUBLIC METHOD SUMMARY

```
void ref ();  
void unref ();  
void addDeletionCallback (vzObjectCallback callback, void* userData);  
void removeDeletionCallback (vzObjectCallback callback, void* userData);
```

## PROTECTED METHOD SUMMARY

```
vzObject ();  
virtual ~vzObject ();
```

## CLASS DESCRIPTION

The *vzObject* class provides simple reference counting for each Volumizer object type. It also provides a callback mechanism for deletion notification: A user can install callbacks that will be invoked whenever an object is deleted. The class is derived from [vzMemory](#), which provides additional memory management functionality.

## METHOD DESCRIPTIONS

### **vzObject()**

```
vzObject ();
```

Constructor for objects. The reference count field is initialized to one.

### **~vzObject()**

```
virtual ~vzObject ();
```

Destructor for objects. The destructor is protected to enforce that all deletion must be performed through the use of the [unref\(\)](#) method.

### **addDeletionCallback()**

```
void addDeletionCallback (vzObjectCallback callback, void* userData);
```

Adds the specified callback to the list of callbacks to be executed just before the object is finally deleted. The deletion callback has the following form:

```
typedef void (*vzObjectCallback) (vzObject *object, void *userData);
```

When the deletion callback is invoked, two parameters are passed: the object which is about to disappear, and the user data pointer.

The deletion callback can be a useful tool in performing memory management for user-allocated data structures. The following code sets a deletion callback to free the volume data when the corresponding parameter is about to be deleted -

```

// Add a deletion callback to the given volume texture
void addDeletionCB(vzParameterVolumeTexture *texture) {

    // Get the data pointer to be deleted
    void *dataPtr = texture->getDataPtr();

    // Add the deletion callback with the dataPtr as the user data
    texture->addDeletionCallback(textureDeletionCB, dataPtr);
}

// The texture deletion callback function
void textureDeletionCB(vzObject *object, void *userData) {

    // The user data pointer
    delete [] userData;
}

```

Note: It is legal to add multiple deletion callbacks with the same function pointer but different user data pointers. Hence the above callback function could be used for multiple textures, each with a different user data pointer. If this method is called with the same function and user data pointers, no action is taken.

#### **ref()**

```
void ref ();
```

Increases the reference count for the object by one.

#### **removeDeletionCallback()**

```
void removeDeletionCallback (vzObjectCallback callback, void* userData);
```

Removes the specified callback from the deletion callback list. For a callback to be removed, both its function pointer and user data pointer must match the input arguments. If the given callback is not found, a `vzError::warn()` is issued.

#### **unref()**

```
void unref ();
```

Decreases the reference count for the object by one. Deletes the object if reference count drops to zero.

#### **SEE ALSO**

[vzMemory](#)

[Back to Index](#)

### NAME

**vzPTLUTShader** - [Shader for volume rendering with lookup tables.](#)

### INHERITS FROM

[vzShader](#)

### HEADER FILE

```
#include <Volumizer2/PTLUTShader.h>
```

### PUBLIC METHOD SUMMARY

[vzPTLUTShader](#) ();

### PROTECTED METHOD SUMMARY

virtual [~vzPTLUTShader](#) ();

### CLASS DESCRIPTION

The *vzPTLUTShader* is a built-in shader to be used with the [vzPTRenderAction](#). The shader provides the ability to interactively modify the transfer function, which controls the mapping from per-vertex values to colors and transparencies.

### PARAMETERS

The shader makes use of two parameters:

- o Name - "vertex\_data", type - [vzParameterVertexData](#)
- o Name - "lookup\_table", type [vzParameterLookupTable](#)

### METHOD DESCRIPTIONS

#### **vzPTLUTShader()**

```
vzPTLUTShader ();
```

Constructor.

#### **~vzPTLUTShader()**

```
virtual ~vzPTLUTShader ();
```

Destructor.

### SEE ALSO

[vzPTRenderAction](#), [vzParameterLookupTable](#), [vzParameterVertexData](#), [vzShader](#)

**NAME**

**vzPTRenderAction** - [Projected Tetrahedra Render Action](#).

**INHERITS FROM**

[vzRenderAction](#)

**HEADER FILE**

```
#include <Volumizer2/PTRenderAction.h>
```

**PUBLIC METHOD SUMMARY**

```
vzPTRenderAction ( );
~vzPTRenderAction ( );
virtual void draw (vzShape* shape);
virtual void drawCells (vzShape* shape, bool useTransferFunction=false);
virtual void drawBoundary (vzShape* shape, bool useTransferFunction=false);
virtual void manage (vzShape* shape);
virtual void beginDraw (unsigned int rendererFlags);
virtual void endDraw ( );
virtual void unmanage (vzShape* shape);
void setSorter (vzSorter sorter);
```

**CLASS DESCRIPTION**

[vzPTRenderAction](#) implements a direct volume rendering algorithm for irregular grids.

[vzPTRenderAction](#) uses the Projected Tetrahedra Algorithm to volume render a given irregular grid. The volumetric geometry for the grid is first tessellated into a tetrahedral mesh. This tetrahedral mesh is sorted in a back-to-front order. Each tetrahedron is then rendered one at a time using the projected tetrahedra algorithm. Each tetrahedron is projected to screen space and approximated with a set of triangles. Per vertex data associated with the grid is linearly interpolated for the projected vertices and is used to set color and opacity values for that vertex.

**EXAMPLES**

The following code represents a sample use of the [vzPTRenderAction](#) class:

```
vzPTRenderAction renderer();

renderer.setSorter(VZ_CONCAVE);

renderer.manage(shape1);           // manage two shapes.
renderer.manage(shape2);           //

renderer.beginDraw(0);
renderer.draw (shape1);             // draw shape 1
renderer.draw (shape2);             // draw shape 2
renderer.endDraw();
```

Note that the volume shapes are rendered in the order the draw() methods are called, so it is the application's responsibility to sort them in the correct visibility order.



```

vzGeometry *geometry = myInitGeometry();

vzPTLUTShader *shader = new vzPTLUTShader();

vzAppearance *appearance = new vzAppearance(shader);

vzShape *shape = new vzShape(geometry, appearance);

vzParameterVertexData *vertexData = myInitVertexData();
vzParameterLookupTable *lut = myInitLUT();

appearance->setParameter("vertex_data", vertexData);
appearance->setParameter("lookup_table", lut);

```

## METHOD DESCRIPTIONS

### **vzPTRenderAction()**

```
vzPTRenderAction ( );
```

Constructor for [PTRenderAction](#) class.

### **~vzPTRenderAction()**

```
~vzPTRenderAction ( );
```

Destructor for [PTRenderAction](#) class.

### **beginDraw()**

```
virtual void beginDraw (unsigned int rendererFlags);
```

This method is used to tell the render action that the application is now ready to issue draw commands on the shapes. Following this, all the methods invoked on the render action must be [draw\(\)](#) methods until the invocation of the next [endDraw\(\)](#).

### **draw()**

```
virtual void draw (vzShape* shape);
```

This method renders a [vzShape](#). If the shape being drawn has not already been managed, then the render action will implicitly call [manage\(\)](#).

### **drawBoundary()**

```
virtual void drawBoundary (vzShape* shape, bool useTransferFunction=false);
```

This method renders the boundary of the geometry in [vzShape](#) as polygons. If *useTransferFunction* is true, then the transfer function will be applied to the rendered geometry to assign per-vertex color values.

### **drawCells()**

```
virtual void drawCells (vzShape* shape, bool useTransferFunction=false);
```

This method renders the cells of the geometry in [vzShape](#) as polygons. If *useTransferFunction* is true, then the transfer function will be applied to the rendered geometry to assign per-vertex color values.

### **endDraw()**

```
virtual void endDraw ( );
```

This method tells the render action that the application is finished executing [draw\(\)](#) commands. Following this, any calls to

[draw\(\)](#) are not legal until a new [beginDraw\(\)](#) command is issued.

### **manage()**

virtual void manage (vzShape\* *shape*);

Manages the specified shape, so that later it can be drawn. This call increases the reference count for the shape. The effects of this command are not immediate, but are rather delayed until the next call to [draw\(\)](#).

### **setSorter()**

void setSorter (vzSorter *sorter*);

Set the sorter mode for the next call to [draw\(\)](#).

### **unmanage()**

virtual void unmanage (vzShape\* *shape*);

Removes the specified shape from the list of currently managed shapes. Unmanaging a shape frees up its resources to be used for drawing other shapes.

### **SEE ALSO**

[PTRenderAction](#), [vzPTRenderAction](#), [vzRenderAction](#), [vzShape](#)

---

[Back to Index](#)

### NAME

**vzParameter** - [Shader parameter for a shape's appearance.](#)

### INHERITS FROM

[vzObject](#)

### HEADER FILE

```
#include <Volumizer2/Parameter.h>
```

### PUBLIC METHOD SUMMARY

[vzParameter](#) ( );

### PROTECTED METHOD SUMMARY

virtual [~vzParameter](#) ( ) = 0;

### CLASS DESCRIPTION

Provides a simple interface for defining parameters to be used by the [vzAppearance](#) class. Each [vzShader](#) accepts certain parameters which should be derived from the vzParameter base class.

### METHOD DESCRIPTIONS

#### **vzParameter()**

[vzParameter](#) ( );

Constructor for a parameter.

#### **~vzParameter()**

virtual [~vzParameter](#) ( ) = 0;

Destructor for a parameter.

### SEE ALSO

[vzAppearance](#), [vzObject](#), [vzShader](#)

**NAME**  
**vzParameterClipTexture** - [Parameter representing a 3-dimensional clip-texture hierarchy.](#)

**INHERITS FROM**  
[vzParameter](#)

**HEADER FILE**  
#include <Volumizer2/ParameterClipTexture.h>

## PUBLIC METHOD SUMMARY

### Constructor

[vzParameterClipTexture](#) (const int *dataDimensions*[3], vzTextureType *dataType*, vzExternalTextureFormat *externalFormat*, vzInternalTextureFormat *internalFormat*=VZ\_DEFAULT\_INTERNAL\_FORMAT);

### Visible region settings

void [setCenterOfInterest](#) (const int *coi*[3]);  
void [setRoamingWindowSize](#) (const int *dimensions*[3]);  
void [setRegionDirty](#) (const int *offset*[3], const int *level*, const int *dimensions*[3]);

### Data Loader methods

void [setDataLoaderCB](#) (const vzDataLoaderCB *loaderCB*, void\* *userData*);  
void [setMaxThreads](#) (const int *numThreads*);

### Other parameters

void [setPhysicalMemorySize](#) (const unsigned long *maxMemorySize*);  
void [setBrickDimensions](#) (const int *dims*[3]);  
void [setGeometryROI](#) (const double *geometryROI*[6]);

### Get methods

void [getDataDimensions](#) (int *dim*[3]);  
void [getGeometryROI](#) (double *geometryROI*[6]);  
int [getNumLevels](#) ( );  
void [getBrickDimensions](#) (int *dim*[3]);  
vzTextureType [getDataType](#) ( );  
vzExternalTextureFormat [getDataFormat](#) ( );  
vzInternalTextureFormat [getInternalTextureFormat](#) ( );  
void [getCenterOfInterest](#) (int *coi*[3]);  
void [getRoamingWindowSize](#) (int *dimensions*[3]);  
void [getDataLoaderCB](#) (vzDataLoaderCB& *loaderCB*, void\*& *userData*);  
unsigned long [getPhysicalMemorySize](#) ( );

**PROTECTED METHOD SUMMARY**  
virtual [~vzParameterClipTexture](#) ( );

## CLASS DESCRIPTION

vzParameterClipTexture represents a 3-dimensional clip-texture. The clip-texture is internally represented as a collection of mip-mapped texture levels, clipped to fit in the main memory of the system. The texture data is organized into separate texture bricks to organize the data at the various levels of the hierarchy. The complete volume data does not need to be in main memory and is paged in from disk asynchronously. This is done by using a loader callback provided by the application

(see ).

The following other parameters effect the way the hierarchy is set up and so, these values can either be set using the constructor or immediately after the construction when the clip-texture has not been managed and or drawn using the [vzClipRenderAction](#).

- Data dimensions - Size of the whole volume data. Used to compute the total number of bricks in the volume and the number of levels in the hierarchy.
- Texture type and formats - Includes the texture format, type and internal format. Used to compute the number of bytes per texel and create the volume textures representing the sub-bricks.
- Texture brick dimensions - Size of the texture brick. Must be defined in terms of {X, Y, Z} texels.
- Geometry ROI - Used to map the texture data onto world coordinates. Each of the sub-textures will be assigned a roi accordingly. Default values are {0, 0, 0} to {1, 1, 1}.
- Data load callback - Used to load data from the disk. Sample callbacks are provided with the demo code

The following other parameters are used by the application to update the clip-texture depending on the user interaction, potentially on a per-frame basis:

- Center of interest - The {X, Y, Z} index in volume data space, corresponding to the current center of interest.
- Roaming window size - The dimensions of the actual roaming window containing the visible volume data. By default, all the data is assumed to be visible.

The following parameters are computed by the clip-texture class from the parameters specified above:

- Number of levels - Number of levels of resolution for the clip texture. The number of levels depends on the texture brick size and the original data dimensions. Level zero corresponds to the original data resolution and higher levels correspond to sub-sampled versions of the original data.
- Number of threads - Maximum number of threads to be used to page in texture data from the disk. The default number is equal to the number of clip-levels in the heirarchy (see [setMaxThreads\(\)](#))

All the set methods also have corresponding get methods.

## METHOD DESCRIPTIONS

### **vzParameterClipTexture()**

```
vzParameterClipTexture (const int dataDimensions[3], vzTextureType dataType, vzExternalTextureFormat externalFormat,  
vzInternalTextureFormat internalFormat=VZ_DEFAULT_INTERNAL_FORMAT);
```

Constructor for the clip-texture heirarchy.

### **~vzParameterClipTexture()**

```
virtual ~vzParameterClipTexture ( );
```

Destructor for the vzParameterClipTexture. Destroys the clip-texture hierarchy and the data loader threads used to load volume data.

### **getBrickDimensions()**

```
void getBrickDimensions (int dim[3]);
```

Get the dimensions of the texture bricks in the heirarchy.

### **getCenterOfInterest()**

```
void getCenterOfInterest (int coi[3]);
```

Get the current center of interest for the clip-texture.

### **getDataDimensions()**

```
void getDataDimensions (int dim[3]);
```

Get the data dimensions of the highest level-of-resolution in the heirarchy.

#### **getDataFormat()**

```
vzExternalTextureFormat getDataFormat ( );
```

Get the data format for the texture.

#### **getDataLoaderCB()**

```
void getDataLoaderCB (vzDataLoaderCB& loaderCB, void*& userData);
```

Get the data load callback and the user data pointer.

#### **getDataType()**

```
vzTextureType getDataType ( );
```

Get the data type for the texture.

#### **getGeometryROI()**

```
void getGeometryROI (double geometryROI[6]);
```

Get the geometry ROI for the complete clip-texture.

#### **getInternalTextureFormat()**

```
vzInternalTextureFormat getInternalTextureFormat ( );
```

Get the texture internal format for the texture.

#### **getNumLevels()**

```
int getNumLevels ( );
```

Get the number of levels in the heirarchy.

#### **getPhysicalMemorySize()**

```
unsigned long getPhysicalMemorySize ( );
```

Get the size of the physical memory for this clip-texture. Note that this value is only approximate and the actual amount of memory used can be greater than or smaller than this value.

#### **getRoamingWindowSize()**

```
void getRoamingWindowSize (int dimensions[3]);
```

Get the current roaming window size for the clip-texture.

#### **setBrickDimensions()**

```
void setBrickDimensions (const int dims[3]);
```

Set the dimensions of the texture bricks. This method should be called before the clip-texture is managed/drawn for the first time. The default value for the brick size is { 128, 128, 128 }.

#### **setCenterOfInterest()**

```
void setCenterOfInterest (const int coi[3]);
```

Set the center of interest for the current frame. Texture bricks closer to the center of interest will be paged in earlier compared to the bricks which are further away. This can be computed using the viewer's position of viewing direction. The position needs to be specified in the "data space", i.e. in the range [0 ... dataDimensions[i] - 1].

#### **setDataLoaderCB()**

```
void setDataLoaderCB (const vzDataLoaderCB loaderCB, void* userData);
```

The callback for loading volume data from disk has the following format:

```
typedef bool (*vzDataLoaderCB)(const int offset[3], const int level, const int dim[3],  
                               void *data, void *userData);
```

Callback used to load in volume data is passed the position *offset* and level *level*. The dimensions of the data to be loaded is given by *dimensions*. The resulting data will be set in *data*. The following code uses the provided sample IFL loaders to load in the volume data.

```
void loadDataCB(int offset[3], int level, int dimensions[3], void *data, void *userData) {  
    IFLLoader *loader = ((ClipLoaders **)userData)->getLoaderForLevel(level);  
    loader->loadBrick(data, offset, dimensions);  
}
```

### setGeometryROI()

```
void setGeometryROI (const double geometryROI[6]);
```

Set the geometry ROI for the whole clip-texture. This defines the mapping of the texture data onto geometry space. The default value is {0, 0, 0} - {1, 1, 1}. All the bricks in the clip-texture are scaled appropriately.

### setMaxThreads()

```
void setMaxThreads (const int numThreads);
```

Maximum number of threads to be used for paging textures from the disk. The default number of threads created is equal to the number of levels in the hierarchy.

*Note on Thread-safe loaders:* When using multiple loader threads, multiple threads might be scheduled to load different bricks from the same clip-level. In this case, the loader used for the clip-level needs to be thread safe. For example, if the loader uses the same file for all the bricks in a clip-level and multiple invocations of the loader callback use the same file descriptor, the loader might not be thread-safe. The clip-texture implementation currently assumes that the loaders are *not* thread safe and *does not* schedule multiple threads to load bricks from the same clip-level simultaneously. This can however reduce the performance of the disk paging algorithm. In order to allow multiple threads to load bricks from the same clip-level, use thread-safe loaders (e.g. RoamLoader version 3.0 shipped with Volumizer 2.4 onwards) and then enable concurrent reads from a clip-level by setting the environment variable VOLUMIZER\_THREAD\_SAFE\_LOADER. The above is done automatically by ClipGen3d shipped with Volumizer 2.4 when you select the -level0 option and the remaining options are default (-roamVersion 3.0). The ClipLoader class in src/lib/loaders/ checks for thread-safe loaders and sets the above environment variable, if loaders are thread-safe.

### setPhysicalMemorySize()

```
void setPhysicalMemorySize (const unsigned long maxMemorySize);
```

Limit the size of the volumetric data to be kept in main memory to *maxMemorySize*. This value should be set depending on the total memory available on the system. Specifying a small value will mean that less high-resolution bricks will be resident in main memory at any given instant and hence lower resolution levels of the clip-texture will be rendered and vice-versa. The default value is computed to be four times the texture memory available on the graphics sub-system. For example, the physical memory size when running on InfiniteReality3 graphics systems (256 MB texture memory) will be 1 GB. Note that the value specified here is an approximate value and actual memory usage would normally differ from the specified value.

### setRegionDirty()

```
void setRegionDirty (const int offset[3], const int level, const int dimensions[3]);
```

Mark the volumetric data in the given region as dirty. This would force the bricks which intersect the given region to be

invalidated and re-loaded as needed. The interface for this is similar to the data loader callback. This can be used by applications to implement techniques like volumetric tagging to mask out the volume data which is no longer needed. This method is not implemented currently.

### **setRoamingWindowSize()**

```
void setRoamingWindowSize (const int dimensions[3]);
```

Set the size of roaming window. This window is used as the roaming window in conjunction with the center of interest. The dimensions are assumed to be specified according to the original data dimensions. The actual roaming window dimensions for each clip-level is computed depending on the total physical memory that clip-texture is allowed to use.

### **SEE ALSO**

[vzClipRenderAction](#), [vzParameter](#)

---

[Back to Index](#)



**NAME**

**vzParameterLookupTable** - [Parameter representing a lookup table.](#)

**INHERITS FROM**

[vzParameter](#)

**HEADER FILE**

```
#include <Volumizer2/ParameterLookupTable.h>
```

**PUBLIC METHOD SUMMARY**

[vzParameterLookupTable](#) (int *width*, void\* *data*, vzTextureType *type*, vzExternalTextureFormat *format*);

int [getWidth](#) ( ) const;

void [setDataFormat](#) (vzExternalTextureFormat *format*);

vzExternalTextureFormat [getDataFormat](#) ( ) const;

void [setDataType](#) (vzTextureType *type*);

vzTextureType [getDataType](#) ( ) const;

void [setDataPtr](#) (int *width*, void\* *data*);

void\* [getDataPtr](#) ( ) const;

**PROTECTED METHOD SUMMARY**

[~vzParameterLookupTable](#) ( );

**CLASS DESCRIPTION**

*vzParameterLookupTable* provides a simple abstraction for a lookup table. The lookup table can be used to apply transfer functions to map texel values to color values.

**METHOD DESCRIPTIONS****vzParameterLookupTable()**

[vzParameterLookupTable](#) (int *width*, void\* *data*, vzTextureType *type*, vzExternalTextureFormat *format*);

Constructor for a lookup table parameter. Sets the width of the table to *width* and sets the data pointer to *data*. *type* defines the OpenGL data type of the table (FLOAT, UNSIGNED CHAR, etc.). *format* defines the OpenGL external format of the table (RGBA, LUMINANCE\_ALPHA, etc.). See [vzTextureType](#) and [vzExternalTextureFormat](#) for a list of data types and data formats.

**~vzParameterLookupTable()**

[~vzParameterLookupTable](#) ( );

Destructor for texture tables.

**getDataFormat()**

vzExternalTextureFormat [getDataFormat](#) ( ) const;

Retrieve the texture format of the lookup table

**getDataPtr()**

void\* [getDataPtr](#) ( ) const;

Retrieve the data pointer for the lookup table data.

**getDataType()**

```
vzTextureType getDataType ( ) const;
```

Retrieve the data type of the lookup table

### **getWidth()**

```
int getWidth ( ) const;
```

Get the size of the table.

### **setDataFormat()**

```
void setDataFormat (vzExternalTextureFormat format);
```

Set the texture format of the lookup table. See [vzExternalTextureFormat](#) for a list of supported data formats.

### **setDataPtr()**

```
void setDataPtr (int width, void* data);
```

Set the data pointer for the lookup table data. Note: data is NOT copied, so all memory allocation must be done by the user.

*width* specifies the number of entries in the table, while *data* specifies the data itself. Data should be provided in interleaved format. In other words, for RGBA data, the data should be formatted as { {R1, G1, B1, A1}, {R2, G2, B2, A2}, ...}. Each RGBA four-tuple is treated as a single entry within the lookup table. If you modify the data pointer, you must call this method explicitly to let the API know of the change.

### **setDataType()**

```
void setDataType (vzTextureType type);
```

Set the data type of the lookup table. See [vzTextureType](#) for a list of supported data types.

## **SEE ALSO**

[vzExternalTextureFormat](#), [vzParameter](#), [vzTextureType](#)

---

[Back to Index](#)

### NAME

**vzParameterVec3f**  - [Parameter representing a vector of three floating point values.](#)

### INHERITS FROM

[vzParameter](#)

### HEADER FILE

```
#include <Volumizer2/ParameterVec3f.h>
```

### PUBLIC METHOD SUMMARY

```
vzParameterVec3f ( );  
vzParameterVec3f (const float val[3]);  
void setValue (const float val[3]);  
void getValue (float val[3]) const;
```

### PROTECTED METHOD SUMMARY

```
~vzParameterVec3f ( );
```

### CLASS DESCRIPTION

*vzParameterVec3f* is a parameter which represents a vector of three floating point values. Methods are provided to set and get the values. This class is useful for shaders which require parameters like light direction, material color, etc.

### METHOD DESCRIPTIONS

#### **vzParameterVec3f()**

```
vzParameterVec3f ( );
```

Constructor for a [vzParameterVec3f](#). The default value for the vector is set to (1, 0, 0).

#### **vzParameterVec3f()**

```
vzParameterVec3f (const float val[3]);
```

Constructor for a [vzParameterVec3f](#).

#### **~vzParameterVec3f()**

```
~vzParameterVec3f ( );
```

Destructor for the [vzParameterVec3f](#).

#### **getValue()**

```
void getValue (float val[3]) const;
```

Get the value of the parameter.

#### **setValue()**

```
void setValue (const float val[3]);
```

Set the value of the parameter.

### SEE ALSO

[vzParameter](#), [vzParameterVec3f](#)

---

[Back to Index](#)

**NAME**

**vzParameterVertexData** - [Parameter representing per vertex values.](#)

**INHERITS FROM**

[vzParameter](#)

**HEADER FILE**

```
#include <Volumizer2/ParameterVertexData.h>
```

**PUBLIC METHOD SUMMARY**

```
vzParameterVertexData (int numVertices, int valuesPerVertex, float* solutionValues);
```

```
void setDataPtr (int numVertices, int valuesPerVertex, float* solutionValues);
```

```
float* getDataPtr ( ) const;
```

```
int getNumValuesPerVertex ( ) const;
```

```
int getNumVertices ( ) const;
```

```
void getDataRange (int value, float range[2]);
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzParameterVertexData ( );
```

**CLASS DESCRIPTION**

*vzParameterVertexData* represents per vertex values associated with a mesh. Each vertex in the mesh can have any number of floating point values associated with it. These values are specified in an interleaved manner for each of the vertices. For example, the Plot3D solution file format stores the following values: density, x\_momentum, y\_momentum, z\_momentum and energy.

**METHOD DESCRIPTIONS****vzParameterVertexData()**

```
vzParameterVertexData (int numVertices, int valuesPerVertex, float* solutionValues);
```

Constructor for [vzParameterVertexData](#) class. The constructor does not allocate any memory either for the vertex values. Calls [setDataPtr\(\)](#) with *numVertices*, *valuesPerVertex* and *solutionValues* as arguments.

**~vzParameterVertexData()**

```
virtual ~vzParameterVertexData ( );
```

Destructor for vertex data parameters.

**getDataPtr()**

```
float* getDataPtr ( ) const;
```

Return the array of solution values for all the vertices. The data is returned in the form: (vertex 1 solution 1, vertex 1 solution 2, vertex 1 solution 3 . . . , vertex 2 solution 1, vertex 2 solution 2, ...)

**getDataRange()**

```
void getDataRange (int value, float range[2]);
```

Get the range of data values in the solution field.

**getNumValuesPerVertex()**

```
int getNumValuesPerVertex ( ) const;
```

Returns the number of values per vertex in the mesh.

### **getNumVertices()**

```
int getNumVertices ( ) const;
```

Get the number of vertices in the solution field.

### **setDataPtr()**

```
void setDataPtr (int numVertices, int valuesPerVertex, float* solutionValues);
```

Set all of the *valuesPerVertex* solution values for all the vertices. *solutionValues* should be of size *numValues* \* *numVertices*, where *numVertices* is the number of vertices in the given mesh and *numValues* is the number of solution values per vertex.

Note: The pointer *solutionValues* is shallow copied. No memory allocation takes place internally. The user is responsible for allocating and deallocating the memory for the pointer.

The data should be passed in the form:

(vertex 1 solution 1, vertex 1 solution 2, vertex 1 solution 3 . . . ,  
vertex 2 solution 1, vertex 2 solution 2, ...)

### **SEE ALSO**

[vzParameter](#), [vzParameterVertexData](#)

---

[Back to Index](#)

**NAME**

**vzParameterVolumeTexture** - [Parameter representing a 3-dimensional texture.](#)

**INHERITS FROM**

[vzParameter](#)

**HEADER FILE**

```
#include <Volumizer2/ParameterVolumeTexture.h>
```

**PUBLIC METHOD SUMMARY**

```
vzParameterVolumeTexture (const int dataDimensions[3], void* dataPtr, vzTextureType dataType,
    vzExternalTextureFormat externalFormat,
    vzInternalTextureFormat internalFormat=VZ_DEFAULT_INTERNAL_FORMAT);
vzParameterVolumeTexture (const int dataDimensions[3], const int dataROI[6], void* dataPtr, vzTextureType dataType,
    vzExternalTextureFormat externalFormat,
    vzInternalTextureFormat internalFormat=VZ_DEFAULT_INTERNAL_FORMAT);
void getDataDimensions (int dims[3]) const;
vzExternalTextureFormat getDataFormat ( ) const;
vzInternalTextureFormat getInternalTextureFormat ( ) const;
vzTextureType getDataType ( ) const;
void setDataPtr (void* data);
void* getDataPtr ( ) const;
void getDataROI (int dataROI[6]) const;
void setGeometryROI (const double geometryROI[6]);
void optimize ( );
bool isOptimized ( ) const;
void getGeometryROI (double geometryROI[6]) const;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzParameterVolumeTexture ( );
```

**CLASS DESCRIPTION**

*vzParameterVolumeTexture* provides an abstraction for a 3-D texture, together with a transformation which maps the texture into geometry space (see [setGeometryROI](#)(.)). The class stores a pointer to the volume data, along with methods that control the format and type of the data.

**METHOD DESCRIPTIONS****vzParameterVolumeTexture()**

```
vzParameterVolumeTexture (const int dataDimensions[3], void* dataPtr, vzTextureType dataType,
    vzExternalTextureFormat externalFormat,
    vzInternalTextureFormat internalFormat=VZ_DEFAULT_INTERNAL_FORMAT);
```

Constructor for a volume texture parameter. *dataDimensions* specify the x,y, and z dimensions of the data stored in *dataPtr*. *dataType* describes the OpenGL data type of the texture. *externalFormat* describes the external (user) format of the texture. *internalFormat* describes the internal (texture memory) format of the texture. If the internal format is not specified, a default will be inferred from the external format and data type. The following shows an example use of the above constructor -

```
// Create a volume texture of size (256, 256,256) given the data pointer
vzParameterVolumeTexture *createTexture(void *dataPtr) {
```

```

// Data dimensions
int dataDimensions[3] = {256, 256, 256};

// Create the texture
vzParameterVolumeTexture *texture = new vzParameterVolumeTexture (
    dataDimensions,
    dataPtr,
    VZ_LUMINANCE,
    VZ_UNSIGNED_BYTE,
    VZ_DEFAULT_INTERNAL_FORMAT);

return(texture);
}

```

See [vzTextureType](#), [vzExternalTextureFormat](#) and [vzInternalTextureFormat](#) for a list of supported texture types and formats. The default values for the geometry ROI is (0, 0, 0) to (1, 1, 1).

### **vzParameterVolumeTexture()**

```

vzParameterVolumeTexture (const int dataDimensions[3], const int dataROI[6], void* dataPtr, vzTextureType dataType,
    vzExternalTextureFormat externalFormat,
    vzInternalTextureFormat internalFormat=VZ_DEFAULT_INTERNAL_FORMAT);

```

Alternate constructor for a volume texture parameter, for cases when a sub-cubic texture region of interest (*dataROI*) must be specified. *dataDimensions* specify the x,y, and z dimensions of the data stored in *dataPtr*. *dataROI* defines a sub-region of interest within the data pointer. Texture samples outside the dataROI will be ignored during all subsequent operations.

The Data ROI dimensions should be specified in the order (xmin, ymin, zmin, xmax, ymax, zmax). Remaining arguments are the same as the other constructor. The following constructor can be used to create a texture of dimensions (32, 32, 32) at an offset of (64, 64, 64) in the above example -

```

// Define a data ROI
int dataROI[6] = {64, 64, 64, 95, 95, 95};

// Create the texture
vzParameterVolumeTexture *texture = new vzParameterVolumeTexture (
    dataDimensions,
    dataROI,
    dataPtr,
    VZ_LUMINANCE,
    VZ_UNSIGNED_BYTE,
    VZ_DEFAULT_INTERNAL_FORMAT);

```

### **~vzParameterVolumeTexture()**

```

virtual ~vzParameterVolumeTexture ();

```

Destructor for a volume texture parameter.

### **getDataDimensions()**

```

void getDataDimensions (int dims[3]) const;

```

Get the dimensions of the volume data.

### **getDataFormat()**



`vzExternalTextureFormat getDataFormat ( ) const;`

Get the external format of the volume texture.

### **getDataPtr()**

`void* getDataPtr ( ) const;`

Retrieve the data pointer for the texture data.

### **getDataROI()**

`void getDataROI (int dataROI[6]) const;`

Retrieve the current region of interest for this volume texture data. The region of interest is specified in the form (xmin, ymin, zmin, xmax, ymax, zmax).

### **getDataType()**

`vzTextureType getDataType ( ) const;`

Get the data type of the volume texture.

### **getGeometryROI()**

`void getGeometryROI (double geometryROI[6]) const;`

Returns the current geometryROI by retrieving the coordinates (xmin, ymin, zmin, xmax, ymax, zmax).

### **getInternalTextureFormat()**

`vzInternalTextureFormat getInternalTextureFormat ( ) const;`

Get the internal format of the volume texture. This indicates the format the data is stored in texture memory.

### **isOptimized()**

`bool isOptimized ( ) const;`

Returns true if [optimize\(\)](#) was called on this texture. Note that this returns true only if the application has requested this texture to be optimized explicitly by calling [optimize\(\)](#). If a render action does other transparent optimizations on the texture data, then this method would still return false.

### **optimize()**

`void optimize ( );`

Optimize the texture data in order to improve performance. On Infinite Reality systems, this will interleave the texture data if needed. The texture is interleaved according to the external and internal texture formats specified by the constructor. If the internal format is VZ\_DEFAULT\_INTERNAL\_FORMAT, then the appropriate formats would be inferred. The optimized data would be stored in the texture and will be re-usable for any subsequent operations by the render action.

Note: the interleaving process might be delayed until the shape containing the texture is actually managed for the first time. Hence, you cannot delete the original data pointer after calling this method.

If you do not call [optimize\(\)](#) and specify an interleaved internal format, then the render action assumes that your texture data is already interleaved. Also, the data should be compliant with OpenGL's texture specifications. For example, the textures should fit in texture memory and should have power-of-two dimensions.

### **setDataPtr()**

`void setDataPtr (void* data);`

Set the data pointer for the texture data. Note: data is NOT copied, so all memory allocation must be done by the user. This method marks the entire texture as dirty. It will be reloaded into texture memory the next time it is needed. Also, if the texture was marked to be optimized (see [optimize\(\)](#)), then the texture data would be re-processed.

**setGeometryROI()**

```
void setGeometryROI (const double geometryROI[6]);
```

Explicitly set the geometry ROI in terms of (xmin, ymin, zmin, xmax, ymax, zmax). This value is used to map the texture data ROI, to the geometry space. Changing the values allows you to arbitrarily scale and translate your texture to fit the geometry that you have defined. Specifically, voxel samples along the border of the texture are mapped so that they lie exactly along the boundaries of the *geometryROI*. Geometry outside the *geometryROI* will be clipped out during rasterization. The default values for the geometry ROI is (0, 0, 0) to (1, 1, 1).

**SEE ALSO**

[vzExternalTextureFormat](#), [vzInternalTextureFormat](#), [vzParameter](#), [vzTextureType](#)

---

[Back to Index](#)

**NAME**

**vzPolyGeometry**  - [Polygonal geometry associated with a shape node.](#)

**INHERITS FROM**

[vzGeometry](#)

**HEADER FILE**

```
#include <Volumizer2/PolyGeometry.h>
```

**PUBLIC METHOD SUMMARY**

```
vzPolyGeometry ( );  
virtual void draw (double geometryROI[6]) const = 0;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzPolyGeometry ( );
```

**CLASS DESCRIPTION**

The *vzPolyGeometry* class represents any polygonal geometry attached to a [vzShape](#) node. The class provides an pure virtual method [draw\(\)](#) which is invoked by the render action while rendering the shape node.

**METHOD DESCRIPTIONS**

**vzPolyGeometry()**

```
vzPolyGeometry ( );
```

Constructor for polygonal geometry.

**~vzPolyGeometry()**

```
virtual ~vzPolyGeometry ( );
```

Destructor for polygonal geometry.

**draw()**

```
virtual void draw (double geometryROI[6]) const = 0;
```

Pure virtual method which is invoked by the render action to draw the polygonal geometry. The argument *geometryROI* provides the geometric region-of-interest for the polygonal geometry.

**SEE ALSO**

[vzGeometry](#), [vzShape](#)

**NAME**

**vzRenderAction** - [Renderer for drawing shape nodes.](#)

**HEADER FILE**

```
#include <Volumizer2/RenderAction.h>
```

**PUBLIC METHOD SUMMARY**

```
virtual void manage (vzShape* shape) = 0;
virtual void unmanage (vzShape* shape) = 0;
virtual void draw (vzShape* shape) = 0;
virtual void beginDraw (unsigned int rendererFlags) = 0;
virtual void endDraw ( ) = 0;
```

**PROTECTED METHOD SUMMARY**

```
vzRenderAction ( );
virtual ~vzRenderAction ( );
```

**CLASS DESCRIPTION**

*vzRenderAction* provides an abstract base class for all render actions that can be applied to one or more [vzShape](#) nodes. By overriding the virtual methods in this class, it is possible to derive new render actions to render volumetric shapes. For instance, in order to write a new ray-casting volume renderer, the user can derive from the base class and override the virtual methods.

**METHOD DESCRIPTIONS****vzRenderAction()**

```
vzRenderAction ( );
```

Constructor for the [vzRenderAction](#) class.

**~vzRenderAction()**

```
virtual ~vzRenderAction ( );
```

Destructor for [vzRenderAction](#).

**beginDraw()**

```
virtual void beginDraw (unsigned int rendererFlags) = 0;
```

This pure virtual method is used to tell the render action that the application is done managing and un-managing all the shapes and is ready to issue [draw\(\)](#) commands for the shapes. Following this, all the commands invoked on the render action should be [draw\(\)](#) commands until the invocation of the next [endDraw\(\)](#). The *rendererFlags* can be used to pass renderer specific flags.

**draw()**

```
virtual void draw (vzShape* shape) = 0;
```

This pure virtual method provides the interface to render all of the volumetric data for a single shape node. The technique for rendering is determined by the derived class.

**endDraw()**

```
virtual void endDraw ( ) = 0;
```

This pure virtual method is used to tell the render action that the application is done drawing all the shapes for the current frame. Note that it is illegal to issue [manage\(\)](#) and [unmanage\(\)](#) commands between a [beginDraw\(\)/endDraw\(\)](#) pair.

### **manage()**

```
virtual void manage (vzShape* shape) = 0;
```

This pure virtual method provides the interface to add the specified shape to a list of "managed" shapes. Managed shapes are shapes that the renderer knows about before the `draw()` call. A shape **must** be managed before it is drawn for the first time.

The list of managed shapes is interpreted in a renderer-specific way. For instance, the texture-mapping render action treats the managed shapes as a list of shapes to keep resident in texture memory.

Note that the effect of managing a shape is not necessarily immediate: actions may be queued up until the next call to [draw\(\)](#).

### **unmanage()**

```
virtual void unmanage (vzShape* shape) = 0;
```

This method removes the specified shape from the list of "managed" shapes. The shape specified cannot be drawn until it is once again managed by calling [manage\(\)](#).

### **SEE ALSO**

[vzRenderAction](#), [vzShape](#)

---

[Back to Index](#)

### NAME

**vzShader** - [Shader for generating a desired visual effect from an appearance.](#)

### INHERITS FROM

[vzObject](#)

### HEADER FILE

```
#include <Volumizer2/Shader.h>
```

### PROTECTED METHOD SUMMARY

```
vzShader ();  
virtual ~vzShader ();
```

### CLASS DESCRIPTION

The *vzShader* class represents any shader attached to a [vzAppearance](#). Each shader represents a particular rendering technique to be used with a particular render action.

### METHOD DESCRIPTIONS

#### **vzShader()**

```
vzShader ();
```

Constructor for a shader.

#### **~vzShader()**

```
virtual ~vzShader ();
```

Destructor for a shader.

### SEE ALSO

[vzAppearance](#), [vzObject](#)

## NAME

**vzShape** - [Container node for a volume's geometry and appearance.](#)

## INHERITS FROM

[vzObject](#)

## HEADER FILE

```
#include <Volumizer2/Shape.h>
```

## PUBLIC METHOD SUMMARY

```
vzShape (vzGeometry* geometry, vzAppearance* appearance);  
void setGeometry (vzGeometry* geometry);  
vzGeometry* getGeometry ( ) const;  
void setAppearance (vzAppearance* appearance);  
vzAppearance* getAppearance ( ) const;
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzShape ( );
```

## CLASS DESCRIPTION

*vzShape* provides a complete abstraction of a volumetric shape. Each shape has a geometry ([vzGeometry](#)) and an appearance ([vzAppearance](#)) associated with it. The geometry controls the spatial attributes of the shape while the appearance controls the visual attributes of the shape.

Shape nodes are parameters to the draw() method of render actions ([vzRenderAction](#)). Shape nodes are atomic, by definition; it is not possible to render part of a shape. Hence, they can be used to represent the leaf nodes of a scene graph.

## METHOD DESCRIPTIONS

### **vzShape()**

```
vzShape (vzGeometry* geometry, vzAppearance* appearance);
```

Constructor for a volume shape.

### **~vzShape()**

```
virtual ~vzShape ( );
```

Destructor for a volume shape.

### **getAppearance()**

```
vzAppearance* getAppearance ( ) const;
```

Retrieve the appearance of this volume shape.

### **getGeometry()**

```
vzGeometry* getGeometry ( ) const;
```

Retrieve the geometry of this volume shape.

### **setAppearance()**

```
void setAppearance (vzAppearance* appearance);
```

Set the appearance of this volume shape. Increases the reference count of the appearance by one. Calls the [unref\(\)](#) method of

the previous appearance, if there was one.

### **setGeometry()**

```
void setGeometry (vzGeometry* geometry);
```

Set the geometry of this volume shape. Increases the reference count of the geometry by one. Calls the `unref()` method of the previous geometry, if there was one.

### **SEE ALSO**

[vzAppearance](#), [vzGeometry](#), [vzObject](#), [vzRenderAction](#)

---

[Back to Index](#)



**NAME**

**vzShapeSet** - [A set of volumetric shapes.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/ShapeSet.h>
```

**PUBLIC METHOD SUMMARY****Creation/Modification**

```
vzShapeSet ( );
static vzShapeSet* load (const char* filename);
void add (vzShape* shape);
void remove (vzShape* shape);
void setGraphicsState (vzGraphicsState* state);
```

**Get methods**

```
int getNumShapes ( ) const;
vzShape* get (int i) const;
void getBoundingBox (float bbox[6]) const;
vzGraphicsState* getGraphicsState ( ) const;
```

**Utility methods**

```
void sort (int* indices) const;
static void setLoaderCB (vzShapeSetLoaderCB loaderPtr);
```

**PROTECTED METHOD SUMMARY**

```
~vzShapeSet ( );
```

**CLASS DESCRIPTION**

vzShapeSet provides a simple abstraction for a collection of volumetric shapes ([vzShape](#)). Individual shapes can be added to and removed from the set. The class provides utility routines for computing the bounding box and visibility sorting of the set, among others. The ShapeSet provides a static [load\(\)](#) method which uses the libvzxml loader library (based on Apache's Xerces library) to load and create a shape set.

**METHOD DESCRIPTIONS****vzShapeSet()**

```
vzShapeSet ( );
```

Constructor for the shape set.

**~vzShapeSet()**

```
~vzShapeSet ( );
```

Destructor for the shape set.

**add()**

```
void add (vzShape* shape);
```

Add the *shape* to the shape set. The shape is added to the end of the set.

### **get()**

```
void vzShape* get (int i) const;
```

Get the *i* 'th shape from the set.

### **getBoundingBox()**

```
void getBoundingBox (float bbox[6]) const;
```

Return the bounding box of the set of shapes.

### **getGraphicsState()**

```
void vzGraphicsState* getGraphicsState () const;
```

Get the graphics state for the shape set.

### **getNumShapes()**

```
int getNumShapes () const;
```

Returns the number of shapes in the set

### **load()**

```
static void vzShapeSet* load (const char* filename);
```

Load a *filename*. For any plug-in modules, the plug-in DSO has to define the callback functions with the name *createObjectName*, where the *ObjectName* is the name of the class *without* the *vz* prefix.

```
// Type definition for the object loader callback
typedef void* vzObject * (*vzObjectLoaderCB)(const char *filename,
                                             const char *userString);

// Example for loading class vzPolyGeometry
extern "C" void* vzObject *createPolyGeometry(const char *filename,
                                             const char *userString)
{
    vzPolyGeometry *geom = MyPolyLoader::load(filename);
    if(!geom) {
        fprintf(stderr, "Cannot load file %s using MyPolyLoader\n", filename);
        return NULL;
    }

    return (void*)geom;
}
```

### **remove()**

```
void remove (void* shape);
```

Remove the *shape* from the shape set.

### **setGraphicsState()**

```
void setGraphicsState (void* state);
```

Set the current graphics state to be used for controlling LOD parameters for this shape set. The graphics state would be used to update the center-of-interest of any clip-textures in the shape set. The application needs to update the graphics state depending on user interaction, etc. If the application does not use the above method to set a graphics state, the class would

create a default one internally.

### **setLoaderCB()**

```
static void setLoaderCB (vzShapeSetLoaderCB loaderPtr);
```

Set the loader method for this shape set. The loader method is defined as

```
typedef vzShapeSet * (*vzShapeSetLoaderCB)(const char *filename);
```

This callback is set by the libvzxml library automatically if the application links against the library or dlopen's the library. In this case, the shape set will be loaded using the XML based Volumizer loader. Applications can implement their own loaders for loading a [vzShapeSet](#) by setting the appropriate loader callback.

### **sort()**

```
void sort (int* indices) const;
```

Sort the set of shapes in a back-to-front visibility sorted order. The sorted order is returned as the indices of the shapes in *indices* in a back-to-front sorted order. Note: The method does not allocate any memory for *indices* so the length of the array should atleast be equal to the value returned by [getNumShapes\(\)](#). The sort method assumes a valid OpenGL context and queries the OpenGL transformation matrices using the *glGetDoublev* calls.

*Note on Sorting:* The ShapeSet currently uses an approximate sorting algorithm which sorts the shapes by distance from the view point. This will not give accurate results if the shapes are of different sizes and/or if they are intersecting. If accurate sorting is critical, the application would need to implement its own sorting algorithm.

### **SEE ALSO**

[vzObject](#), [vzShape](#), [vzShapeSet](#)

---

[Back to Index](#)

**NAME**

**vzSlicePlaneSet** - [A set of slicing planes.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/SlicePlaneSet.h>
```

**PUBLIC METHOD SUMMARY**

```
vzSlicePlaneSet (int maxPlanes);
int getNumPlanes ( );
int setPlaneEquation (int index, double planeEqn[4]);
int getPlaneEquation (int index, double planeEqn[4]);
int enable (int index);
bool isEnabled (int index);
int disable (int index);
```

**PROTECTED METHOD SUMMARY**

```
~vzSlicePlaneSet ( );
```

**CLASS DESCRIPTION**

*vzSlicePlaneSet* provides a simple abstraction for specifying and rendering arbitrary slice planes. The slicing planes are specified by using the four coefficients of each of the planes. The slice planes are rendered in a sorted order using a recursive clipping algorithm. Methods are provided to manipulate, enable and disable individual slice planes.

**METHOD DESCRIPTIONS****vzSlicePlaneSet()**

```
vzSlicePlaneSet (int maxPlanes);
```

Constructor for [vzSlicePlaneSet](#). One can specify any number of slicing planes using the *maxPlanes* parameter. A default value is selected for the plane equations of each of planes. These values can be modified later using the set methods. All the slice planes are enabled by default.

**~vzSlicePlaneSet()**

```
~vzSlicePlaneSet ( );
```

Destructor for [vzSlicePlaneSet](#).

**disable()**

```
int disable (int index);
```

Disable the slice plane given by *index*. Returns *index* on success, -1 otherwise.

**enable()**

```
int enable (int index);
```

Enable the slice plane given by *index*. Returns *index* on success, -1 otherwise.

**getNumPlanes()**

```
int getNumPlanes ( );
```

Return the number of slice planes in the slice plane set.

**getPlaneEquation()**

int getPlaneEquation (int *index*, double *planeEqn*[4]);

Get the plane equation for plane number *index* in *planeEqn*. Returns *index* on success, -1 otherwise.

**isEnabled()**

bool isEnabled (int *index*);

Returns true if the plane *index* is enabled. Returns false otherwise.

**setPlaneEquation()**

int setPlaneEquation (int *index*, double *planeEqn*[4]);

Set the plane equation for plane number *index* to *planeEqn*. Returns *index* on success, -1 otherwise.

**SEE ALSO**

[vzObject](#), [vzSlicePlaneSet](#)

---

[Back to Index](#)

**NAME**  
**vzStructuredHexaMesh** - [Volumetric geometry representing a structured hexhedral mesh.](#)

**INHERITS FROM**  
[vzVolumeGeometry](#)

**HEADER FILE**  
#include <Volumizer2/StructuredHexaMesh.h>

**PUBLIC METHOD SUMMARY**  
[vzStructuredHexaMesh](#) (const int *dim*[3], vzVertexArray\* *vertices*);  
void [setVertexArray](#) (const int *dim*[3], vzVertexArray\* *vertices*);  
vzVertexArray\* [getVertexArray](#) ( ) const;  
void [getDimensions](#) (int *dim*[3]) const;  
vzUnstructuredTetraMesh\* [tessellate](#) ( ) const;

**PROTECTED METHOD SUMMARY**  
virtual [~vzStructuredHexaMesh](#) ( );

**CLASS DESCRIPTION**  
*vzStructuredHexaMesh* provides a simple mechanism for storing and manipulating structured hexahedral meshes. Structured meshes consist of hexahedral cells such that each node in the mesh can be indexed by three indices, just as one would index a three-dimensional array. The connectivity information for the hexahedra is implicit in structured meshes and hence no index information needs to be stored.

### METHOD DESCRIPTIONS

**vzStructuredHexaMesh()**  
vzStructuredHexaMesh (const int *dim*[3], vzVertexArray\* *vertices*);

Constructor for a StructuredHexaMesh. The dimensions of the mesh are set to (*dim*[0], *dim*[1], *dim*[2]). The [setVertexArray\(\)](#) method is used to set the vertices. *vertices* should contain (*dim*[0], *dim*[1], *dim*[2]) vertices arranged in a row major order.

**~vzStructuredHexaMesh()**  
virtual [~vzStructuredHexaMesh](#) ( );

Destructor for a StructuredHexaMesh.

**getDimensions()**  
void [getDimensions](#) (int *dim*[3]) const;

Get the dimensions of the mesh. The dimensions are copied into *dim*.

**getVertexArray()**  
vzVertexArray\* [getVertexArray](#) ( ) const;

Return a pointer to the class [vzVertexArray](#) containing the vertex coordinates.

**setVertexArray()**  
void [setVertexArray](#) (const int *dim*[3], vzVertexArray\* *vertices*);

Explicitly assign the vertex coordinates for the structured mesh to *vertices*. Also, set the dimensions for the structured mesh

to *dim*. Increases the reference count of *vertices*. The number of vertices in the *vertices* should be equal to (*dim[0]*, *dim[1]*, *dim[2]*)

### **tessellate()**

```
vzUnstructuredTetraMesh* tessellate ( ) const;
```

Tessellate the StructuredHexaMesh into a tetrahedral mesh. The tessellation generates five tetrahedra per hexahedral cell. The tessellation is done in a manner so that the faces of the tetrahedra in one cell match with those of the adjacent cells. If the dimensions of the mesh is (X, Y, Z), then the number of tetrahedra generated would be  $5 * (X-1) * (Y-1) * (Z-1)$ .

### **SEE ALSO**

[vzVertexArray](#), [vzVolumeGeometry](#)

---

[Back to Index](#)

## NAME

**vzTMFragmentProgram** - [Shader for using fragment programs.](#)

## INHERITS FROM

[vzTMShader](#)

## HEADER FILE

```
#include <Volumizer2/TMFragmentProgram.h>
```

## PUBLIC METHOD SUMMARY

```
static vzTMFragmentProgram* load (const char* filename);
vzTMFragmentProgram (char* fragmentProgram);
void setProgram (char* fragmentProgram);
char* getProgram ();
void setMultiTexCallbacks (int numCB, vzTMShaderCB* callbacks, void* userData);
void getMultiTexCallbacks (int& numCB, vzTMShaderCB*& callbacks, void*& userData);
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzTMFragmentProgram ();
```

## PROTECTED MEMBER SUMMARY

```
TMFragmentProgramImpl* \_impl;
```

## CLASS DESCRIPTION

The *vzTMFragmentProgram* is a built-in shader to be used with the [vzTMRenderAction](#). The class is derived from [vzTMShader](#) and provides an interface for programmable shading using the ARB\_fragment\_program extension. The shader accepts the fragment program string as an argument to its constructor. The fragment program is applied to each polygon being rendered for the shape's geometry. The program string can be updated using the appropriate set method. The OpenGL state for the shader can be set and restored using the pre and post shape callbacks. The volume is rendered using 3D texture mapping with the given "volume" texture as the currently bound texture.

## PARAMETERS

The shader uses a single parameter:

- o Name - "volume", type - [vzParameterVolumeTexture](#)

## METHOD DESCRIPTIONS

### **vzTMFragmentProgram()**

```
vzTMFragmentProgram (char* fragmentProgram);
```

Constructor. The program string is shallow copied internally to the fragment program and hence should not be deleted.

### **~vzTMFragmentProgram()**

```
virtual ~vzTMFragmentProgram ();
```

Destructor.

### **getMultiTexCallbacks()**

```
void getMultiTexCallbacks (int& numCB, vzTMShaderCB*& callbacks, void*& userData);
```

Get the multi-texture callbacks.



### **getProgram()**

```
char* getProgram ();
```

Get the program string for this shader.

### **load()**

```
static vzTMFragmentProgram* load (const char* filename);
```

Factory used to load the fragment program from the given file name.

### **setMultiTexCallbacks()**

```
void setMultiTexCallbacks (int numCB, vzTMShaderCB* callbacks, void* userData);
```

Sets the multi-texture callbacks for the fragment program. These callbacks can be used to bind the volume textures and lookup tables to specific texture units corresponding to the callback number. The callbacks might be used to set texture unit/object specific OpenGL state by directly calling the corresponding OpenGL functions. The default number of callbacks is 0. The callbacks are defined as -

```
typedef void (*vzTMShaderCB)(vzTMShaderData *shaderData);
```

For example, the following code initializes the callbacks for a fragment program implementing a dependent texture lookup using the "lookup\_table" parameter.

```
static char FragmentProgram[] = "!!ARBfp1.0\n"
                                "TEMP volume;\n"
                                "TEX volume, fragment.texcoord[0], texture[0], 3D;\n"
                                "TEX result.color, volume, texture[1], 1D;\n"
                                "END"

// Load the fragment program
vzTMFragmentProgram *fp = new vzTMFragmentProgram(FragmentProgram);

const int texCallbacks = 2;
vzTMShaderCB *multiTexCB = new vzTMShaderCB[texCallbacks];
multiTexCB[0] = tex0;
multiTexCB[1] = tex1;

// Set the multi-texture callbacks as slice callbacks
fp->setMultiTexCallbacks(texCallbacks, multiTexCB, fp);
```

The multi-texture callbacks might look like the following two callbacks -

```
void tex0(vzTMShaderData *data) {
    // enable "volume" texture
    data->bindVolumeTextureCB("volume", data, VZ_TM_ENABLE);

    // bind "volume" texture
    if(!data->bindVolumeTextureCB("volume", data)) {
        cerr<<"slice0: Error binding texture 'volume'"<<endl;
    }
}
```

```
}  
  
void tex1(vzTMShaderData *data) {  
  
    // enable "lookup_table" parameter (1D texture on Onyx4 systems)  
    data->bindLookupTableCB("lookup_table", data, VZ_TM_ENABLE);  
  
    // bind "lookup_table" parameter  
    if(!data->bindLookupTableCB("lookup_table", data)) {  
        cerr<<"slice1: Error binding texture 'lookup_table'"<<endl;  
    }  
}
```

### setProgram()

```
void setProgram (char* fragmentProgram);
```

Update the program string for the fragment program.

### MEMBER DESCRIPTIONS

#### **\_impl**

```
TMFragmentProgramImpl* _impl;
```

### SEE ALSO

[vzParameterVolumeTexture](#), [vzTMRenderAction](#), [vzTMShader](#)

---

[Back to Index](#)

### NAME

**vzTMGradientShader** - [Built-in gradient shader to be used with the vzTMRenderAction.](#)

### INHERITS FROM

[vzShader](#)

### HEADER FILE

```
#include <Volumizer2/TMGradientShader.h>
```

### PUBLIC METHOD SUMMARY

[vzTMGradientShader](#) ( );

### PROTECTED METHOD SUMMARY

virtual [~vzTMGradientShader](#) ( );

### CLASS DESCRIPTION

The *vzTMGradientShader* is a built-in shader to be used with the [vzTMRenderAction](#). The shader implements a 3-pass shading algorithm to perform gradient shading of volume data. The shader uses two perfectly overlapping volumes to perform the gradient shading. The first volume defines the actual volume data, while the other volume defines the gradient for the first volume. The RGB values of the gradient texture provide the (a, b, c) coefficients for the gradient at each texel in the original volume.

The shader computes the dot product of the gradient values with the light direction using the OpenGL Imaging pipeline. This is done efficiently by setting the appropriate color matrix before downloading the gradient texture. Changing the light direction forces the gradient texture to be re-downloaded in order to re-compute the dot products with the new light direction. In addition, the shader accepts a lookup table parameter to apply transfer functions to the volume data. The shader allows controlling the diffuse and ambient coefficients using two optional parameters. If not specified, the diffuse coefficient is assumed to be (1.0, 1.0, 1.0) and the ambient to be (0.0, 0.0, 0.0).

### PARAMETERS

The shader expects six parameters. Two of these parameters, namely ambient and diffuse, are optional parameters:

- o Name - "volume", type - [vzParameterVolumeTexture](#)
- o Name - "gradient", type - [vzParameterVolumeTexture](#)
- o Name - "lightdir", type - [vzParameterVec3f](#)
- o Name - "lookup\_table", type - [vzParameterLookupTable](#)
- o Name - "ambient", type - [vzParameterVec3f](#) (optional)
- o Name - "diffuse", type - [vzParameterVec3f](#) (optional)

Note: The shading algorithm uses destination alpha to compute the gradient lighting. Hence, the application should make sure that the appropriate visual is selected.

### METHOD DESCRIPTIONS

**vzTMGradientShader()**  
`vzTMGradientShader ( );`

Constructor.

**~vzTMGradientShader()**  
`virtual ~vzTMGradientShader ( );`

Destructor.

**SEE ALSO**

[vzParameterLookupTable](#), [vzParameterVec3f](#), [vzParameterVolumeTexture](#), [vzShader](#), [vzTMRenderAction](#)

---

[Back to Index](#)

## NAME

**vzTMISLShader** - [TMShader using the Interactive Shading Language \(ISL\)](#).

## INHERITS FROM

[vzTMShader](#)

## HEADER FILE

```
#include <Volumizer2/TMISLShader.h>
```

## PUBLIC METHOD SUMMARY

```
static vzTMISLShader* load (const char* filename);  
vzTMISLShader (islAppearance* appearance);  
void setISLAppearance (islAppearance* appearance);  
islAppearance* getISLAppearance (void);
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzTMISLShader ( );
```

## CLASS DESCRIPTION

Class vzTMISLShader provides an abstraction for a ISL shader to be used with the TMRenderAction. The shader class takes an islAppearance as an argument to it's constructor and creates the appropriate shader representation internally.

## METHOD DESCRIPTIONS

### **vzTMISLShader()**

```
vzTMISLShader (islAppearance* appearance);
```

Constructor. Accepts an islAppearance as its arguments. The appearance need not be compiled. The shader will automatically compile the appearance as needed.

### **~vzTMISLShader()**

```
virtual ~vzTMISLShader ( );
```

Destructor for the vzTMISLShader.

### **getISLAppearance()**

```
islAppearance* getISLAppearance (void);
```

Get the islAppearance for the shader.

### **load()**

```
static vzTMISLShader* load (const char* filename);
```

A utility factory to load an ISL shader from the given filename and return the appropriate vzTMISLShader object.

### **setISLAppearance()**

```
void setISLAppearance (islAppearance* appearance);
```

Set the islAppearance for the shader to *appearance*.

## SEE ALSO

[vzTMShader](#)

---

[Back to Index](#)

### NAME

**vzTMLUTShader** - [Shader for volume rendering 3D textures with lookup tables applied.](#)

### INHERITS FROM

[vzShader](#)

### HEADER FILE

```
#include <Volumizer2/TMLUTShader.h>
```

### PUBLIC METHOD SUMMARY

[vzTMLUTShader](#) ( );

### PROTECTED METHOD SUMMARY

virtual [~vzTMLUTShader](#) ( );

### CLASS DESCRIPTION

The *vzTMLUTShader* is a built-in shader to be used with the [vzTMRenderAction](#). The LUT shader provides the ability to interactively modify a texture lookup table, which controls the mapping from texture values to colors and transparencies.

### PARAMETERS

The shader makes use of two parameters:

- o Name - "volume", type - [vzParameterVolumeTexture](#)
- o Name - "lookup\_table", type [vzParameterLookupTable](#)

### METHOD DESCRIPTIONS

#### **vzTMLUTShader()**

[vzTMLUTShader](#) ( );

Constructor.

#### **~vzTMLUTShader()**

virtual [~vzTMLUTShader](#) ( );

Destructor.

### SEE ALSO

[vzParameterLookupTable](#), [vzParameterVolumeTexture](#), [vzShader](#), [vzTMRenderAction](#)

---

[Back to Index](#)

**NAME**

**vzTMRenderAction** - [Texture-mapping render action.](#)

**INHERITS FROM**

[vzRenderAction](#)

**HEADER FILE**

```
#include <Volumizer2/TMRenderAction.h>
```

**PUBLIC METHOD SUMMARY**

```
vzTMRenderAction (int maxThreads);
virtual ~vzTMRenderAction ( );
void setSamplingRate (const float samplingRate[3]);
virtual void manage (vzShape* shape);
virtual void unmanage (vzShape* shape);
virtual void draw (vzShape* shape);
virtual void beginDraw (unsigned int rendererFlags);
virtual void endDraw ( );
```

**CLASS DESCRIPTION**

The Texture Mapping Render Action implements the volume rendering pipeline using 3D texture mapping. For each shape drawn, the renderer clips the geometry to texture boundaries, tessellates the geometry into tetrahedra, sorts the tetrahedra, and then rasterizes each tetrahedron in back-to-front order. Rasterization proceeds by slicing each tetrahedron into polygons (polygonization) and texture mapping each slice with a 3-D texture. The computed polygons are then passed to the shader which employs the appropriate OpenGL state and rendering passes to generate the desired visual effect.

The Texture Mapping render action provides resource management for the texture data, which is provided as parameters to the shape's appearance. It is legal to draw shapes whose volume textures do not fit within texture memory. The render action divides the texture into bricks that fit in texture memory and renders them appropriately.

**NOTES**

The following are some important issues to keep in mind while using the Texture Mapping render action.

- Thread safety and multi-pipe issues: Only one render action per graphics pipe should be constructed. This insures that texture memory usage is optimal for each graphics pipe. The render action is not thread-safe, hence the application would need to implement the appropriate thread-safety mechanisms if it intends to share it across multiple threads. The render action also assumes a valid OpenGL context when the [beginDraw\(\)](#)/[draw\(\)](#)/[endDraw\(\)](#) methods are invoked.
- Visibility sorting: The order in which the [draw\(\)](#) method is called determines the order in which shapes are drawn. Therefore it is the user's responsibility to visibility-sort the shapes to be drawn.
- Texture dimensions and sizes: The shapes can have textures with arbitrary textures dimensions and sizes. The render action will pad and brick the textures appropriately in order to render the shape. However, if the application uses multiple shapes which are paged in and out of texture memory constantly (see [manage\(\)](#), [unmanage\(\)](#)), it would be advisable to use shapes with textures of the same sizes and power-of-two dimensions in order to improve the rendering performance.
- Texture optimization: If the shapes being rendered by the render action have volume textures with a default internal texture format, the render action tries to infer an appropriate format for the texture. For example, on InfiniteReality systems, the render action would select an internal format of VZ\_DUAL\_INTENSITY8 for textures with a data format of VZ\_LUMINANCE and data type of VZ\_UNSIGNED\_BYTE or VZ\_BYTE. The render action would also "interleave" the textures appropriately in order to download the texture. This process is completely transparent to the application. This process improves the texture download performance and also reduces the texture memory consumption. You can force the render action to use a particular internal format by passing the appropriate parameters to the constructor for the volume texture.



## SHADERS

The following built-in volume shaders work with the `TMRenderAction` class: [vzTMSimpleShader](#), [vzTMLUTShader](#), [vzTMTangentSpaceShader](#), [vzTMGradientShader](#), [vzTMTagShader](#).

## EXAMPLES

The following code represents a sample use of the [vzTMRenderAction](#) class:

```
vzTMRenderAction renderer(0);    // create a render action

renderer.manage (shape1);        // manage two shapes. this
renderer.manage (shape2);        // allocates texture objects.

renderer.beginDraw(userFlags);   // begin draw

renderer.draw (shape1);          // draw shape 1
renderer.draw (shape2);          // draw shape 2
1
renderer.endDraw();              // end draw
```

## NOTES

### METHOD DESCRIPTIONS

#### **vzTMRenderAction()**

`vzTMRenderAction (int maxThreads);`

Constructor for the [vzTMRenderAction](#) class. Constructor for the `vzTMRenderAction` class. The only parameter, *maxThreads*, specifies how many additional threads the render action is allowed to create, at maximum. This parameter is ignored right now and the render action runs in single-threaded mode.

#### **~vzTMRenderAction()**

`virtual ~vzTMRenderAction ();`

Destructor for [vzTMRenderAction](#).

#### **beginDraw()**

`virtual void beginDraw (unsigned int rendererFlags);`

This method is used to tell the render action that the application is now ready to issue draw commands on the shapes. Following this, all the methods invoked on the render action must be [draw\(\)](#) methods until the invocation of the next [endDraw\(\)](#).

On the invocation of [beginDraw\(\)](#), the render action will perform all the texture management necessary for the currently managed shapes. Thus, the [beginDraw\(\)](#) marks the end of the texture management phase and the beginning of a new draw phase.

The *rendererFlags* can be used to pass hints to the render action by setting bit flags. e.g. If the flag `VZ_RESTORE_GL_STATE_BIT` is set, then at the end of the [endDraw\(\)](#), the render action restores the GL state to the one prior to calling [beginDraw\(\)](#).

#### **draw()**

`virtual void draw (vzShape* shape);`

This method performs the actual volume-rendering for a shape. The volumetric geometry for the shape is sliced using viewport aligned planes and rendered in a back-to-front order. If the volume geometry has slice planes, then the slice planes are simply rendered in the appropriate order.

The draw method sets the OpenGL state necessary for the using the volume textures contained in the shape. The remaining state settings are done by the specific shaders. The application needs to set the blending related OpenGL state before calling [draw\(\)](#) or [beginDraw\(\)](#). For example, the blending function for the common back-to-front blending using the over operator would be `glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

Note: The *shape* must have been managed previously by calling [manage\(\)](#). In addition, the [draw\(\)](#) method must be called within a [beginDraw\(\)](#) / [endDraw\(\)](#) pair.

### **endDraw()**

```
virtual void endDraw ();
```

This method tells the render action that the application is finished executing [draw\(\)](#) commands. Following this, any calls to [draw\(\)](#) are not legal until a new [beginDraw\(\)](#) command is issued.

### **manage()**

```
virtual void manage (vzShape* shape);
```

Forces the vzShape *shape* to be kept resident in texture memory, if possible. The effects of this command are not immediate, but rather are delayed until the next [beginDraw\(\)](#) command is issued.

Given multiple calls to manage and unmanage, the texture manager attempts to optimize the state changes from one frame to the next.

Note: It is illegal to call this method inside of a [beginDraw\(\)/endDraw\(\)](#) pair.

### **setSamplingRate()**

```
void setSamplingRate (const float samplingRate[3]);
```

Set a sampling rate for volume rendering, defined in texture space. The default sampling rate of (1.0, 1.0, 1.0) would sample once per voxel in each texture dimension.

### **unmanage()**

```
virtual void unmanage (vzShape* shape);
```

Removes the specified shape from the list of currently-managed shapes. Un-managing a shape frees up its resources to be used for drawing other shapes.

Note: It is illegal to call this method inside of a [beginDraw\(\)/endDraw\(\)](#) pair.

## **SEE ALSO**

[vzRenderAction](#), [vzTMGradientShader](#), [vzTMLUTShader](#), [vzTMRenderAction](#), [vzTMSimpleShader](#), [vzTMTagShader](#), [vzTMTangentSpaceShader](#)

## NAME

**vzTMShader** - [A general purpose shader to be used with the vzTMRenderAction.](#)

## INHERITS FROM

[vzShader](#)

## HEADER FILE

```
#include <Volumizer2/TMShader.h>
```

## PUBLIC METHOD SUMMARY

```
vzTMShader ();  
void setShapeCallbacks (vzTMShaderCB preCB, vzTMShaderCB postCB, void* userData);  
void getShapeCallbacks (vzTMShaderCB& preCB, vzTMShaderCB& postCB, void*& userData);  
void setSliceCallbacks (int numCB, vzTMShaderCB* callbacks, void* userData);  
void getSliceCallbacks (int& numCB, vzTMShaderCB*& callbacks, void*& userData);
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzTMShader ();
```

## CLASS DESCRIPTION

The vzTMShader class provides a general purpose multi-pass shader which allows applications to apply arbitrary shading to shapes being rendered using the vzTMRenderAction. The vzTMRenderAction can be used to bind the appropriate resources that it manages for the application. This can be done using the following callback -

```
typedef bool (*vzTMBindParameterCB)(const char *name, vzTMShaderData *shaderData,  
                                     vzTMShaderOp operation = VZ_TM_BIND);
```

The above callbacks are used to bind the volume texture and the lookup table (given by the respective parameter names). The callbacks can also be used to simply enable or disable the OpenGL state for the parameters managed by the TMRenderAction, i.e. volume textures and lookup tables.

## METHOD DESCRIPTIONS

### **vzTMShader()**

```
vzTMShader ();
```

Constructor.

### **~vzTMShader()**

```
virtual ~vzTMShader ();
```

Destructor.

### **getShapeCallbacks()**

```
void getShapeCallbacks (vzTMShaderCB& preCB, vzTMShaderCB& postCB, void*& userData);
```

Get the pre and post shape callbacks.

### **getSliceCallbacks()**

```
void getSliceCallbacks (int& numCB, vzTMShaderCB*& callbacks, void*& userData);
```

Get the per-slice callbacks.

## setShapeCallbacks()

void setShapeCallbacks (vzTMShaderCB *preCB*, vzTMShaderCB *postCB*, void\* *userData*);

Sets the callbacks invoked for each shape before and after the shape is rendered. The callbacks are defined as -

```
typedef void (*vzTMShaderCB)(vzTMShaderData *shaderData);
```

For example -

```
void preShape(vzTMShaderData *shaderData) {  
    // Enable volume texture parameter  
    shaderData->bindVolumeTextureCB("volume", shaderData, VZ_TM_ENABLE);  
  
    // Bind the "volume" texture  
    if(!shaderData->bindVolumeTextureCB("volume", shaderData, VZ_TM_BIND))  
        vzError::error(VZ_OPERATION_FAILED, "Could not bind volume texture 'volume'");  
}  
  
void postShape(vzTMShaderData *shaderData) {  
    // Disable the volume texture  
    shaderData->bindVolumeTextureCB("volume", shaderData, VZ_TM_DISABLE);  
}
```

## setSliceCallbacks()

void setSliceCallbacks (int *numCB*, vzTMShaderCB\* *callbacks*, void\* *userData*);

Sets the pass description for the vzTMShader. The default number of callbacks is 0. The callbacks are defined as -

```
typedef void (*vzTMShaderCB)(vzTMShaderData *shaderData);
```

For example -

```
void pass1(vzTMShaderData *shaderData) {  
    if(!shaderData->bindVolumeTextureCB("volume", shaderData))  
        vzError::error(VZ_OPERATION_FAILED, "Could not bind volume texture 'volume'");  
}  
  
void pass2(vzTMShaderData *shaderData) {  
    if(!shaderData->bindVolumeTextureCB("volume2", shaderData))  
        vzError::error(VZ_OPERATION_FAILED, "Could not bind volume texture 'volume2'");  
}
```

## SEE ALSO

[vzShader](#)

---

[Back to Index](#)

## NAME

**vzTMShaderData** - [Shader data used by the vzTMShader class.](#)

## HEADER FILE

```
#include <Volumizer2/TMShader.h>
```

## PUBLIC MEMBER SUMMARY

```
vzTMBindParameterCB bindVolumeTextureCB;  
vzTMBindParameterCB bindLookupTableCB;  
vzAppearance* appearance;  
void* userData;
```

## CLASS DESCRIPTION

The vzTMShaderData class is used by the vzTMShader class to provide shader specific information to the application. This information is passed to the various callbacks used by the vzTMShader to describe multi-pass shading techniques. Objects of this class are allocated by each render action being used to render the shape nodes. Along with important information like the current appearance, the object also passes callbacks which can be used by the shader to bind the appropriate 3D texture and lookup tables in the appearance.

## MEMBER DESCRIPTIONS

### appearance

```
vzAppearance* appearance;
```

The current appearance being used to render the shape node. The appearance contains the shader parameters for the shape node which can be used by the shader to enable different OpenGL state.

### bindLookupTableCB

```
vzTMBindParameterCB bindLookupTableCB;
```

A per-render action callback which can be used by the shader to bind lookup tables in the appearance. The callback takes the parameter name for the lookup table and the vzTMShaderData as arguments and returns true or false depending on success or failure of the callback. The callback is defined as -

```
typedef bool (*vzTMBindParameterCB)(const char *name, vzTMShaderData *shaderData,  
                                     vzTMShaderOp operation = VZ_TM_BIND);
```

The callback can also be used to simply enable or disable the OpenGL state for the lookup table parameter. This is done by specifying the shader operation to be VZ\_TM\_ENABLE or VZ\_TM\_DISABLE respectively.

### bindVolumeTextureCB

```
vzTMBindParameterCB bindVolumeTextureCB;
```

A per-render action callback which can be used by the shader to bind volume textures in the appearance. The callback takes the parameter name for the volume texture and the vzTMShaderData as arguments and returns true or false depending on success or failure of the callback. The callback is defined as -

```
typedef bool (*vzTMBindParameterCB)(const char *name, vzTMShaderData *shaderData,  
                                     vzTMShaderOp operation = VZ_TM_BIND);
```

The callback can also be used to simply enable or disable the OpenGL state for the volume texture parameter. This is done

by specifying the shader operation to be VZ\_TM\_ENABLE or VZ\_TM\_DISABLE respectively.

**userData**

```
void* userData;
```

User data pointer for the shader.

---

[Back to Index](#)

**NAME**

**vzTMSimpleShader** - [Simple shader for volume rendering of 3D textures.](#)

**INHERITS FROM**

[vzShader](#)

**HEADER FILE**

```
#include <Volumizer2/TMSimpleShader.h>
```

**PUBLIC METHOD SUMMARY**

[vzTMSimpleShader](#) ( );

**PROTECTED METHOD SUMMARY**

virtual [~vzTMSimpleShader](#) ( );

**CLASS DESCRIPTION**

The *vzTMSimpleShader* is a built-in shader to be used with the [vzTMRenderAction](#). The volume is rendered using 3D texture mapping with the given "volume" texture as the currently bound texture.

**PARAMETERS**

The shader uses a single parameter:

- o Name - "volume", type - [vzParameterVolumeTexture](#)

**METHOD DESCRIPTIONS**

**vzTMSimpleShader()**

[vzTMSimpleShader](#) ( );

Constructor.

**~vzTMSimpleShader()**

virtual [~vzTMSimpleShader](#) ( );

Destructor.

**SEE ALSO**

[vzParameterVolumeTexture](#), [vzShader](#), [vzTMRenderAction](#)

---



### NAME

**vzTMTagShader** - [Built-in tagging shader to be used with the vzTMRenderAction.](#)

### INHERITS FROM

[vzShader](#)

### HEADER FILE

```
#include <Volumizer2/TMTagShader.h>
```

### PUBLIC METHOD SUMMARY

[vzTMTagShader](#) ();

### PROTECTED METHOD SUMMARY

virtual [~vzTMTagShader](#) ();

### CLASS DESCRIPTION

The *vzTMTagShader* is a built-in shader to be used with the [vzTMRenderAction](#). The shader implements a two pass algorithm to perform volumetric tagging. This is accomplished using two perfectly overlapping volumes. The first volume defines the actual volume data, while the other volume defines the 3-dimensional stencil buffer for the data. Each value in the tag volume contains the mask for the corresponding texel in the volume data. If the alpha value of the tag texel is greater than 0.5, then the corresponding texel in the volume data is rendered otherwise the texel is masked out.

The tagging algorithm uses stencil and alpha tests to perform the task of tagging. Ideally, the tag volume should require only one bit to represent each texel. But, on most graphics hardware, each texel will use atleast one byte to represent a texel. On InfiniteReality graphics, the application can specify the internal texture format to be VZ\_QUAD\_INTENSITY4 and ask the [vzTMRenderAction](#) to optimize the texture. The render action would then interleave the texture so that each texel requires only 4-bits to represent it, considerably improving texture memory consumption and texture download rate.

### PARAMETERS

The shader expects three parameters:

- o Name - "volume", type - [vzParameterVolumeTexture](#)
- o Name - "tag", type - [vzParameterVolumeTexture](#)
- o Name - "lookup\_table", type - [vzParameterLookupTable](#)

Note: The tagging algorithm uses the stencil buffer to mask out the volume data. Hence, the application should make sure that the appropriate visual is selected.

### METHOD DESCRIPTIONS

#### **vzTMTagShader()**

```
vzTMTagShader ();
```

Constructor.

#### **~vzTMTagShader()**

```
virtual ~vzTMTagShader ();
```

Destructor.

### SEE ALSO

[vzParameterLookupTable](#), [vzParameterVolumeTexture](#), [vzShader](#), [vzTMRenderAction](#)

---

[Back to Index](#)

### NAME

**vzTMTangentSpaceShader** - [Shader for volume rendering 3D textures with lookup tables and gradientless lighting.](#)

### INHERITS FROM

[vzShader](#)

### HEADER FILE

```
#include <Volumizer2/TMTangentSpaceShader.h>
```

### PUBLIC METHOD SUMMARY

[vzTMTangentSpaceShader](#) ( );

### PROTECTED METHOD SUMMARY

virtual [~vzTMTangentSpaceShader](#) ( );

### CLASS DESCRIPTION

The *vzTMTangentSpaceShader* is a built-in shader to be used with the [vzTMRenderAction](#). This shader implements a gradientless-shading algorithm to provide volumetric lighting. The given volume texture is rendered with an infinite directional light source. The shader also uses texture lookup tables to modulate the output color values.

The shader uses a two-pass algorithm to approximate the dot product of the gradient vector with the light vector, at each point within the volume. This dot product is then used to modulate the texture values during rendering. For details, refer to Peercy et.al, 'Efficient Bump Mapping Hardware', Computer Graphics, Proc. SIGGRAPH '97.

### PARAMETERS

The shader expects three parameters:

- o Name - "volume", type - [vzParameterVolumeTexture](#)
- o Name - "lookup\_table", type - [vzParameterLookupTable](#)
- o Name - "lightdir", type - [vzParameterVec3f](#)

### METHOD DESCRIPTIONS

#### **vzTMTangentSpaceShader()**

```
vzTMTangentSpaceShader ( );
```

Constructor.

#### **~vzTMTangentSpaceShader()**

```
virtual ~vzTMTangentSpaceShader ( );
```

Destructor.

### SEE ALSO

[vzParameterLookupTable](#), [vzParameterVec3f](#), [vzParameterVolumeTexture](#), [vzShader](#), [vzTMRenderAction](#)

### NAME

**vzTextureType** - [Data types for texture data.](#)

### HEADER FILE

```
#include <Volumizer2/VolEnums.h>
```

### CLASS DESCRIPTION

Enumerated type representing the type of texture data (e.g. float, short, byte). This enum is used by the [vzParameterVolumeTexture](#) and [vzParameterLookupTable](#) classes.

```
enum vzTextureType {  
    VZ_UNSIGNED_BYTE      = GL_UNSIGNED_BYTE ,  
    VZ_UNSIGNED_SHORT     = GL_UNSIGNED_SHORT ,  
    VZ_UNSIGNED_INT       = GL_UNSIGNED_INT ,  
    VZ_FLOAT               = GL_FLOAT ,  
    VZ_BYTE                = GL_BYTE ,  
    VZ_SHORT               = GL_SHORT ,  
    VZ_INT                 = GL_INT  
};
```

### SEE ALSO

[vzParameterLookupTable](#), [vzParameterVolumeTexture](#)

---

[Back to Index](#)

## NAME

**vzUnstructuredHexaMesh** - [Volumetric geometry representing an unstructured hexahedral mesh.](#)

## INHERITS FROM

[vzUnstructuredMesh](#)

## HEADER FILE

```
#include <Volumizer2/UnstructuredHexaMesh.h>
```

## PUBLIC METHOD SUMMARY

```
vzUnstructuredHexaMesh (vzVertexArray* vertices, vzIndexArray* indices);  
vzUnstructuredTetraMesh* tessellate ( ) const;
```

## PROTECTED METHOD SUMMARY

```
virtual ~vzUnstructuredHexaMesh ( );
```

## CLASS DESCRIPTION

*vzUnstructuredHexaMesh* represents indexed sets of hexahedra. Each hexahedron is represented by 8 indices which point to a list of vertex coordinates. Note: To represent structured hexahedral meshes, use [vzStructuredHexaMesh](#), since the connectivity information is implicit in structured meshes.

## METHOD DESCRIPTIONS

### **vzUnstructuredHexaMesh()**

```
vzUnstructuredHexaMesh (vzVertexArray* vertices, vzIndexArray* indices);
```

Constructor for indexed tetra sets. Does not allocate any memory for vertex or index data. Each consecutive set of eight indices provides a single hexahedron. Use [vzUnstructuredMesh::setIndexArray\(\)](#) and [vzUnstructuredMesh::setVertexArray\(\)](#) to change the index and vertex array pointers after initialization.

### **~vzUnstructuredHexaMesh()**

```
virtual ~vzUnstructuredHexaMesh ( );
```

Destructor for indexed hexa sets.

### **tessellate()**

```
vzUnstructuredTetraMesh* tessellate ( ) const;
```

A method to perform a simplicial decomposition of the given geometry data. Tessellates the geometry into a tetrahedral mesh and returns a pointer to `vzUnstructuredTetraMesh`.

## SEE ALSO

[vzStructuredHexaMesh](#), [vzUnstructuredMesh](#)

**NAME**

**vzUnstructuredMesh**  - [Unstructured volumetric geometry.](#)

**INHERITS FROM**

[vzVolumeGeometry](#)

**HEADER FILE**

```
#include <Volumizer2/UnstructuredMesh.h>
```

**PUBLIC METHOD SUMMARY**

```
vzUnstructuredMesh (vzVertexArray* vertices, vzIndexArray* indices);
void setVertexArray (vzVertexArray* vertices);
void setIndexArray (vzIndexArray* indices);
vzVertexArray* getVertexArray ( ) const;
vzIndexArray* getIndexArray ( ) const;
virtual vzUnstructuredTetraMesh* tessellate ( ) const = 0;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzUnstructuredMesh ( );
```

**CLASS DESCRIPTION**

*vzUnstructuredMesh* serves as an abstract base class for all of the unstructured mesh types. Unlike structured meshes, unstructured meshes do not have any restrictions on the number of polyhedra that can share a vertex in the mesh. The mesh is represented with a list of vertices and a list of polyhedra. The list of vertices is represented by a [vzVertexArray](#), and the list of polyhedra is represented by a [vzIndexArray](#). Each polyhedron in the mesh is specified by *n* index values, where *n* is the number of vertices in the polyhedron. Each index corresponds to a vertex in the vertex array.

**METHOD DESCRIPTIONS****vzUnstructuredMesh()**

```
vzUnstructuredMesh (vzVertexArray* vertices, vzIndexArray* indices);
```

Constructor for unstructured meshes. Does not allocate memory for vertices and indices. Only stores the pointers *vertices* and *indices* by calling [setVertexArray\(\)](#) and [setIndexArray\(\)](#) respectively.

**~vzUnstructuredMesh()**

```
virtual ~vzUnstructuredMesh ( );
```

Destructor for unstructured meshes.

**getIndexArray()**

```
vzIndexArray* getIndexArray ( ) const;
```

Return a pointer to the index data.

**getVertexArray()**

```
vzVertexArray* getVertexArray ( ) const;
```

Return a pointer to the vertex data.

**setIndexArray()**

```
void setIndexArray (vzIndexArray* indices);
```

Set the indices of the model to the specified values. Index values refer to individual vertex records. Note: Just the data pointer is stored and no memory is allocated for the indices. This increases the reference count of *indices*.

#### **setVertexArray()**

```
void setVertexArray (vzVertexArray* vertices);
```

Set the vertices of the model to the specified values. Note: Just the data pointer is stored and no memory is allocated for the vertices. This increases the reference count of *vertices*.

#### **tessellate()**

```
virtual vzUnstructuredTetraMesh* tessellate ( ) const = 0;
```

A pure-virtual method to tessellate the given primitive into a set of tetrahedra. This must be defined for each subclass of primitive.

#### **SEE ALSO**

[vzIndexArray](#), [vzVertexArray](#), [vzVolumeGeometry](#)

---

[Back to Index](#)

## NAME

**vzUnstructuredTetraMesh** - [Volumetric geometry representing an unstructured tetrahedral mesh.](#)

## INHERITS FROM

[vzUnstructuredMesh](#)

## HEADER FILE

```
#include <Volumizer2/UnstructuredTetraMesh.h>
```

## PUBLIC METHOD SUMMARY

[vzUnstructuredTetraMesh](#) (vzVertexArray\* *vertices*, vzIndexArray\* *indices*);  
virtual vzUnstructuredTetraMesh\* [tessellate](#) ( ) const;

## PROTECTED METHOD SUMMARY

virtual [~vzUnstructuredTetraMesh](#) ( );

## CLASS DESCRIPTION

[vzUnstructuredTetraMesh](#) represents indexed sets of tetrahedra. Each tetrahedron is represented by 4 indices that point to a list of vertex coordinates.

## METHOD DESCRIPTIONS

### **vzUnstructuredTetraMesh()**

[vzUnstructuredTetraMesh](#) (vzVertexArray\* *vertices*, vzIndexArray\* *indices*);

Constructor for indexed sets of tetrahedra. Does not perform any memory allocation. Each consecutive set of four indices provides a single tetrahedron. Use [vzUnstructuredMesh::setIndexArray\(\)](#) and [vzUnstructuredMesh::setVertexArray\(\)](#) to change the index and vertex array pointers after initialization.

### **~vzUnstructuredTetraMesh()**

virtual [~vzUnstructuredTetraMesh](#) ( );

Destructor for indexed tetra sets.

### **tessellate()**

virtual vzUnstructuredTetraMesh\* [tessellate](#) ( ) const;

Method to tessellate the volumetric geometry into a [vzUnstructuredTetraMesh](#). In this case, creates a new [vzUnstructuredTetraMesh](#) with the same [vzVertexArray](#) and [vzIndexArray](#) as the original one.

## SEE ALSO

[vzIndexArray](#), [vzUnstructuredMesh](#), [vzUnstructuredTetraMesh](#), [vzVertexArray](#)



**NAME**

**vzVertexArray** - [An array of floating-point vertex coordinates.](#)

**INHERITS FROM**

[vzObject](#)

**HEADER FILE**

```
#include <Volumizer2/VertexArray.h>
```

**PUBLIC METHOD SUMMARY**

```
vzVertexArray (int numVertices, float* vertices);
void setDataPtr (int numVertices, float* vertices);
float* getDataPtr ( ) const;
int getNumVertices ( );
void computeBoundingBox (float bbox[6]) const;
```

**PROTECTED METHOD SUMMARY**

```
virtual ~vzVertexArray ( );
```

**CLASS DESCRIPTION**

*vzVertexArray* provides an abstraction for storing an array of vertex coordinates. Stores a user data pointer for an array of floats, but does not perform any memory allocation.

**METHOD DESCRIPTIONS**

**vzVertexArray()**

```
vzVertexArray (int numVertices, float* vertices);
```

Constructor for [vzVertexArray](#). Calls [setDataPtr\(\)](#) with *numVertices* and *vertices* as arguments.

**~vzVertexArray()**

```
virtual ~vzVertexArray ( );
```

Destructor for [vzVertexArray](#).

**computeBoundingBox()**

```
void computeBoundingBox (float bbox[6]) const;
```

Compute and return the bounding box for the given set of vertices. The first three floating point values in *bbox* give the corresponding x, y and z coordinates of the lower bound and the next three give the upper bound.

**getDataPtr()**

```
float* getDataPtr ( ) const;
```

Return the pointer to the array of vertices.

**getNumVertices()**

```
int getNumVertices ( );
```

Return the number of vertices in the array.

**setDataPtr()**

```
void setDataPtr (int numVertices, float* vertices);
```

Set the list of *numVertices* vertices of the model to the specified values. Values should be provided in the form (x1,y1,z1,x2,y2,z2,...) setDataPtr performs a shallow copy of *vertices*: no memory allocation is performed. The method also marks the vertex data as dirty, invoking all notification callbacks. You **must** call this method whenever you modify vertex data.

**SEE ALSO**

[vzObject](#), [vzVertexArray](#)

---

[Back to Index](#)

### NAME

**vtkVolumeGeometry** - [Volumetric geometry associated with a shape node.](#)

### INHERITS FROM

[vtkGeometry](#)

### HEADER FILE

```
#include <Volumizer2/VolumeGeometry.h>
```

### PUBLIC METHOD SUMMARY

```
vtkVolumeGeometry ( );
virtual vtkUnstructuredTetraMesh* tessellate ( ) const = 0;
void setSlicePlanes (vtkSlicePlaneSet* planeSet);
vtkSlicePlaneSet* getSlicePlanes ( );
```

### PROTECTED METHOD SUMMARY

```
virtual ~vtkVolumeGeometry ( );
```

### CLASS DESCRIPTION

The *vtkVolumeGeometry* class represents any volumetric geometry attached to a [vtkShape](#) node. Volumetric geometry primitives cannot be rendered directly by OpenGL. Hence, the render actions generate polygonal geometry to approximate the volumetric geometry of the shape, before it can be rendered. One can also render slicing planes passing through the volumetric geometry by using the [setSlicePlanes\(\)](#) method.

### METHOD DESCRIPTIONS

#### **vtkVolumeGeometry()**

```
vtkVolumeGeometry ( );
```

Constructor for volumetric geometry.

#### **~vtkVolumeGeometry()**

```
virtual ~vtkVolumeGeometry ( );
```

Destructor for volumetric geometry.

#### **getSlicePlanes()**

```
vtkSlicePlaneSet* getSlicePlanes ( );
```

Get the set of slice planes to be rendered for this volume geometry. The default value for the slice planes is NULL. Hence, if not set, this method will return NULL.

#### **setSlicePlanes()**

```
void setSlicePlanes (vtkSlicePlaneSet* planeSet);
```

Set a set of slice planes to be rendered for this volume geometry. If *planeSet* is not NULL, this method disable the "normal volume rendering" mode for the render action. The render action will render the planes in *planeSet* \_clipped against\_ this volumetric geometry. The default value for *planeSet* is NULL.

#### **tessellate()**

```
virtual vtkUnstructuredTetraMesh* tessellate ( ) const = 0;
```

A pure virtual method to perform a simplicial decomposition of the given geometry data. Tessellates the geometry into a

tetrahedral mesh and returns a pointer to `vzUnstructuredTetraMesh`.

**SEE ALSO**

[vzGeometry](#), [vzShape](#)

---

[Back to Index](#)