

3. Assignment: Volume Visualization

Markus Höferlin

Gregor Mückl

Institut für Visualisierung und Interaktive Systeme

1 Topics

- Volume Visualization
- Texturing
- Blending
- Shader Programming with GLSL
- Lighting

2 Introduction

This assignment will introduce you into the topic of volume visualization. A texture based volume visualization technique harnessing 3D texture hardware support through GLSL fragment shaders will be implemented. The program builds upon the viewer we developed in the previous two assignments. Thereby a new scene graph node realizes the new functionality. In contrast to the already implemented scene graph nodes which handled geometric objects, the new node will be able to render volume data sets in a transparent way. Our framework skeleton will take care of loading the 3D data set from a file. Like the other objects, the volume can be rotated, transformed and scaled by means of mouse interactions.

The application will support variable transfer functions. Such functions describe a mapping between the scalar values defined at each grid point of the volume data set to color and transparency. With the help of transfer functions it becomes possible to hide uninteresting parts of the volume by making them transparent. In medical applications e.g. it is a common approach to hide soft tissue or noise but emphasize bone structures at the same time. Therefore an appropriate transfer function has to be modeled by the user of the volume visualization tool. A new QT dialog will enable us to load, modify and save the transfer functions. The user can modify the functions interactively. Effects of changes to the transfer function onto the displayed volume data set will become visible instantly.

The executable `Volume2` of our solution can be found in `/proj/fapra/examples`. Like in the previous assignments, OpenGL 2.0 hardware support is necessary. Reminder: You can query the OpenGL version number with `glxinfo | grep -i 'opengl version'`.

3 First Steps

We prepared another source skeleton for this assignment. Just update your SVN work space folder to obtain the `Volume2` folder. The following table lists the classes and files contained in the package:

File	Class	Description
<code>VolNode.C</code>	<code>VolNode</code>	New Scene Graph Node
<code>LightNode.C</code>	<code>LightNode</code>	Light node for CSG
<code>DatFile.C</code>	<code>DatFile</code>	IO-Interface for volume data sets
<code>VolumeDlg.qt.C</code>	<code>VolumeDlg</code>	Dialog for modifying the volume representation
<code>DrawArea.qt.C</code>	<code>DrawArea</code>	Drawing area for transfer functions
<code>Volume*.glsl</code>		GLSL Shader for volume rendering

You will notice that the skeleton does not contain an executable. Reuse the code of Assignment 1 and 2 to make it work. Just copy your code from the `Modeller` directory into the `Volume2` folder without overwriting any files. To add those files to the svn repository, you have to execute `svn add <fileName>` for each copied file. `svn status` will show you the state of the individual files where a `?` indicates that the file is currently not in the repository.

It does not matter, if you did not finish the first two assignments completely. The rating of your finished program will be based mostly upon the new parts that have to be implemented this time. However, speaking of the basic functionality that has to work, your viewer has to support interactive translation, rotation and scaling of the objects. The feature that enables the user to change camera parameters is also obligatory. On demand—if you did not already implement those features into the viewer—we will give you selected code samples.

To get into the assignment at hand, some changes and extensions to the viewer have to be implemented at first.

Exercise 1: Extension of the Viewer

Two new entries have to be added to the File-Menu. Extend the menu like you did in the previous assignments. The menu has to contain the following entries:

Load Volume
VolumeDialog
OptionsDialog
Quit Viewer

The entry `LoadVolume` opens the file dialog, which enables the user to select `*.dat` files containing volume data sets. Create a new scene graph node of class `VolNode` with the filename as parameter.

The entry `VolumeDialog` opens a dialog, which is implemented by the class `VolumeDlg`. Create its object—analogously to the options dialog.

The light source was positioned at a fixed location in assignment 2. Such an inflexible static behavior can't be accepted any more. Create a menu entry `Adjust Light` and `Reset Light` in the Edit menu. Appropriate slots and keyboard shortcuts should also be defined for the new entries.

Exercise 2: Integrating the Light Node

The class `LightNode` implements a directional light source for CSG. Insert the light node into the scene graph in such a way, so that it is called prior to all other nodes when rendering a new frame.

Insert a rotation node before the light node to be able to change the direction of the light. The user should be able to change the light direction in a way similar to rotating geometric csg objects. This means, that another interaction mode `Adjust Light` has to be added to the menu and the program. Please assure, that `Reset Light` works in a reasonable way and that resetting the whole scene includes resetting the light source.

4 Volume Visualization

4.1 The Basics of Volume Visualization

Starting point for volume visualization is a three dimensional data set. In the medical field, data usually is given on a cartesian grid. This means that a scalar value is given for each point of the grid that is defined as a regular raster with a given resolution in x-, y- and z-direction. The scalar value describes the “density” of the volume at the associated 3D location. The scalar value of arbitrary points lying in between the samples of the volume can be reconstructed with the help of trilinear interpolation.

It is common to use the physical model of a luminous gas for visualizing volume data. Two effects can be observed in an infinitesimal volume element: Light is emitted into the direction of the viewer `Emission`. Second, such light travels along a viewing ray through the volume element to the eye position of the viewer and along its way the light is attenuated. The `attenuation` is directly correlated to the density of the volume, which is defined at each grid point. Other physical effects like diffusion, diffraction and interferences are usually not considered by volume rendering models.

On the foundation of this model the volume representation can be calculated in a similar way to ray tracing. Therefore, viewing rays are started at the eye’s position and then are send through the image plane into the space of the volume’s bounding box. However, varying from standard ray casting, not only the rays’ intersections with the objects along their way have to be taken into account. Moreover, we have to integrate mathematically along each viewing ray. To capture the whole behavior of the volume data along one ray, the volume is sampled along the ray at equidistant locations and the results of the lighting model evaluated at each sample are accumulated.

Sampling and integration is started at the back of the volume, proceeding to its front. In a simplified model this can be expressed by a iterative equation, the so called *compositing equation*:

$$C_{out} = C_{in}(1 - \alpha_k) + \alpha_k C_k \quad (1)$$

At a sampling point s_k , the already accumulated light intensity along the ray C_{in} is weighted according to its transparency $(1 - \alpha_k)$. Then the intensity weighted with the opacity α_k of the volume element at the current sampling position is added.

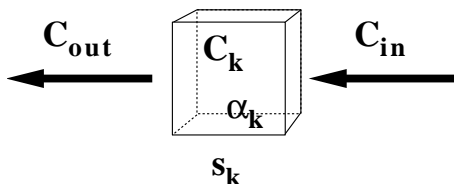


Figure 1: Compositing

In most cases volume samples are not given in the form of emission- and absorption values, but as simple scalar values—like e.g. attenuation in computer tomography. As there is no natural mapping between those scalar values onto intensity and transparency, which would produce suitable results in the general case, we need to define such a mapping by hand. The function describing the mapping is called *transfer function*. It can be implemented as a simple lookup table that specifies a color C_k and opacity α_k value for each scalar value contained in the data set.

4.2 Texture-based Volume Visualization

The evaluation of the compositing equation, which we introduced above, including necessary lighting calculations, can be done directly on modern programmable GPUs. The fundamental idea of texture-based volume visualization is to simulate the sampling along the viewing rays by drawing textured equidistant polygons beginning in the most distant part of the volume data set and ending in the part nearest to the viewer.

This technique is called *slicing*. The color of the rasterized polygons fragments which will be calculated by a fragment shader, will depend on the transfer function and the scalar value found at the fragments 3D location within the volume. Both, the scalar volume data and the transfer function are stored in texture objects that can be accessed directly by the fragment shader through texture lookups. The compositing equation is matched by drawing the slices one after the other from back to front and thereby appropriately using OpenGL’s *Blending* functionality to add the intensity of the new fragment to the already accumulated intensity stored in the frame buffer. With the advent of flexible programmable shader hardware, dependent-texture-lookups like e.g. looking up the associated color/opacity value belonging to a specific scalar value in a transfer function texture, is no problem any more.

Before 3D textures were introduced the volume had to be sampled by a heap of 2D textures mapped on polygons oriented in parallel to the principal axes. Such an (*axis aligned*) approach does not deliver the quality achieved by the technique we will implement in the assignment at hand. Today its common to draw (*viewport aligned*) slices that are parallel to the viewing plane and texturing the slices by directly accessing the volume data stored in a 3D texture like we already hinted above.

5 The Volume Node

In this exercise your task will be to implement the described volume rendering technique. Therefore, a volume node `VolNode` has to be implemented. The following section guides you step by step through the necessary sub-tasks:

5.1 Reading the Volume Data

Exemplary volume data sets are located in the data directory located at `/proj/fapra/handout/A3/data`. You will notice that one data set consists of two files `<name>.raw` and `<name>.dat`. The `raw`-file contains the raw data stored as an three dimensional array of scalar values. Read the data with the following nested for-loops:

```
for (int z = 0; z < res_z; z++)
  for (int y = 0; y < res_y; y++)
    for (int x = 0; x < res_x; x++)
      voxel[...] = readValue();
```

However, the file `<name>.raw` does not contain any information about the resolution of the data set in x-, y- and z-direction, neither does it know the format (character, integer ...) of the stored

data values. Instead of using a file header, such information is written into an associated text file. Additionally, the grid distance is stored as the `SliceThickness` in this file. To avoid a distorted visualization of the volume, the slice thickness has to be taken into account later, when actually drawing the volume.

Reading the volume data from the file should not be your concern. The class `DatFile` handles the file operations. Just create an object of the class with the filename of the volume's `.dat`-file as parameter. On creation, the object reads the description file of the data set. The class also provides methods for querying parameters like the resolution or the grid distances. The method `readData` handles the actual reading of the raw data from the file into a buffer array that has to be specified as a parameter of the method.

5.2 Defining the Slices

Before you start to create texture objects of the data you just read in, you should invest some time into the drawing of the slice polygons.

Exercise 3: Rendering the Slices

Implement the method `VolNode::render`. Like for the other `csg`-nodes, the bounding box of the volume has to be drawn if the volume object is selected. The extents of the bounding box are given by the resolution of the volume data set in x - y - and z -direction and the slice thickness in each direction, respectively. Thereby, the longest of the three axes has to be scaled onto a range of -1 to 1 .

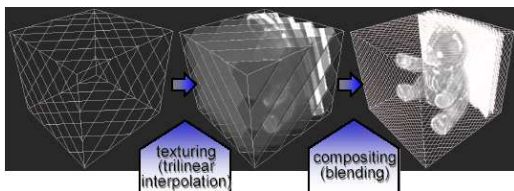


Figure 2: Volume rendering using viewport-aligned slices

Now, the contour of the polygon slices, which are required for the volume rendering, are drawn. Therefore, implement the method `VolNode::drawSlices` that is called in `VolNode::render`. Hint: For picking it is wise to draw filled polygons—otherwise picking only works when clicking on the contour lines.

It is necessary to calculate the normal of the image plane in world coordinates for the viewport-aligned slices. The normal can be retrieved by applying `gluUnProject` onto the view-vector. A more direct alternative looks at the modelview-matrix

$$M = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

and extract the normal $\vec{n} = (m_2 + m_3, m_6 + m_7, m_{10} + m_{11})^T$. Independently of your approach, don't forget to normalize the normal!

Now we use the normal to define equidistant planes that intersect the volume cube. The easiest approach is to use the hessian normal form $ax + by + cz + d = 0$ of the plane. Calculate the distance of the viewer to the nearest and the farthest plane only just intersecting the volume. Then you should use a for-loop to generate planes in fixed distances between both extrema. Now, the intersection points of each plane with the bounding box have to be calculated. It is

advisable to implement a separate helper function for this purpose. This function should return the intersection point - if there is one - given a plane equation and a straight line specified by two points. Please keep in mind, that

- there are three cases, namely no, one, or infinitely intersections.
- we are only interested in the `one`-case! Detect and discard lines running tangential to the image plane before trying to calculate an intersection that does not exist.
- there are intersections that may lie on a straight line, but not between the two points you specified. Discard these intersections.

Before drawing the contour of the slices using `GL_LINE_LOOP` geometry, you have to ensure that the points are ordered clockwise or anticlockwise. Such an order can be achieved in various ways—for convenience, we provide a method `sortVertices` which does the work. The method requires an array containing the intersection points as parameter, in which the elements of the points are ordered sequentially like in $x_1, y_1, z_1, x_2, y_2, z_2, \dots$. Additionally, the number of intersection points (not the number of elements!) has to be specified. The number of points must not exceed six—otherwise something went wrong in the intersection calculations.

Now, we actually draw the `GL_LINE_LOOP`. We recommend using vertex-array for this purpose. Vertex-arrays allow a more efficient block wise transfer of vertices from main memory to the graphics card than an approach using immediate-mode. In immediate mode each single vertices is drawn with a `glVertex`-call. Another advantage of using vertex-arrays is, that you can use the sorted array containing the vertex data directly. The documentation of `glVertexPointer` describes how this all works. The drawing can be done with `glDrawArrays` or even more efficiently with `glMultiDrawArrays`. Don't forget to activate vertex-arrays with the `glEnableClientState` function prior to rendering.

5.3 Textures and Texture coordinates

OpenGL textures are regular gridded 1D, 2D or 3D data structures that may contain scalar or vectorial data elements (texels) of a fixed type. A texel may contain up to 4 components. In the historical sense, textures are primarily used to store image data for standard texturing purposes. In such cases the texels usually contain three 8bit color components (RGB) and one 8bit alpha channel that is used to imitate transparency through blending. Modern graphics hardware internally computes with 32bit floating point accuracy and also allows 32bit texel data. The higher accuracy together with the flexible programmability allows to harness the GPU power for general purpose applications (like e.g. numerical simulations). For performance reasons textures usually are stored directly in graphics memory allowing efficient data access from within shader programs. Independent of the texture's resolution, texture space is always defined on the texture coordinate space $[0..1]^N$ (N = texture dimension).

Thus, it is necessary to set a correct texture coordinate for each vertex of the slice polygons, so that we can sample the 3D volume texture at the correct location in the fragment shader. After rasterization of the slice polygon the fragment shader will be executed for each rasterized fragment once. Using a varying variable it is possible to obtain an interpolated texture coordinate at the 3D location of each fragment.

The bounding box of the data set matches the extends of the volume texture we want to visualize. This is why the texture coordinates at the box vertices would be something like $(0, 0, 0)$, $(1, 0, 0)$, \dots , $(1, 1, 1)$. However, we don't want to draw the box but the slices.

So you have to use interpolated values depending on the position of the intersection vertices.

The following has to be considered:

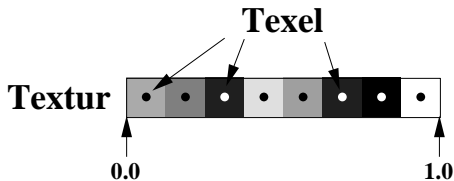


Figure 3: Problems at the texture boundaries

Like you can see in Fig. 3, texel values are defined at the center of a texel. Thus, it is necessary to consider an offset of half a texel in order to access the texture data at the correct location. Within the offset-areas at the border of the texture the data values are interpolated with a special value defined at the texture's border. In our application this can result in problems. This is why it is necessary to offset (increase/ decrease) the texture coordinates by the width of half a texel.

Exercise 4: Visualizing the Texture Coordinates

It is time to ensure that your texture coordinates are set right. Therefore write a simple vertex- and fragment shader that directly interprets the texture coordinates as color values (set alpha to one!). The shaders have to be activated/deactivated in `VolNode::render` before/after drawing. The vertex shader just needs to do the fixed-function-transformation and pass on the texture coordinates in a varying variable. In the fragment shader you can access the interpolated texture coordinate and set the output color `gl_FragColor` accordingly. If you draw your slice polygons as filled `GL_TRIANGLE_FANS`, instead of just drawing contour lines using `GL_LINE_LOOP`, a RGB-cube should become visible.

5.4 Using the 3D Texture and Blending

If you are sure that texture coordinates work please go on and read chapter 6 in the OpenGL Programming Guide to become familiar with the concepts of blending. If blending is activated and configured correctly, the rendered fragment colors will be combined in a way that satisfies the compositing equation we described above.

Exercise 5: Texturing the Slices

Now you should load the volume data set onto the graphics card. Therefore, implement the method `loadVolumeTexture` which uses the details in the `*.dat`-file to create the 3D texture. OpenGL texturing is described in chapter 9 of the OpenGL-Programming Guide. Please read the chapter.

Now you should know how textures are defined in OpenGL, how textured polygons are drawn, and how the mapping of the textures onto geometry is realized by means of texture coordinates. You should also be familiar with texture objects and how they are used. For a 3D texture you will need to use `glTexImage3D`. You can find the most recent specification for this function at <http://www.opengl.org/documentation/specs/>.

Create a texture object for your volume texture. Choose an appropriate format for the texture. Also set the texture mode correctly (hint: look at the description of the `glTexEnv` function). Choose a linear filter for the interpolation of the volume data (hint:

`glTexParameter` documentation). To check, if you set a consistent OpenGL state you can use our `CHECK_GL_ERROR` macro function which queries and outputs OpenGL's error state. Passing a string to the macro makes it possible to identify the location of the problem within your source code.

Now it is time to visualize volume data! Therefore, it is necessary to bind the volume texture to the shader before drawing the slice geometry in `VolNode::render`. First, using `glActiveTexture`, activate the texture hardware unit you want to bind your texture to. Then bind the texture to the active texture sampling unit by using `glBindTexture`. Additionally, it is necessary to tell the shader which shader unit to use when sampling a specific texture. This is done by setting the shader texture variable to the id of the texture sampling unit you bound your texture to.

Also change the fragment shader: For now, the color and the α -channel of `gl_FragColor` should be set to the scalar intensity value found in the volume texture. It is time to get a first glimpse onto the volume data sets which you find in the data directory.

Varying the distance between neighboring slices introduces problems in transparent regions. This is because of the varying number of samples taken along the ray. A large distances leads to sampling less often, thus resulting in a darker pixel. The so called opacity correction fixes such unwanted behavior. The opacity specified by the transfer function is given for a fixed distance d_f . When using different distances use the following function for calculating the corrected alpha value:

$$\alpha_{corrected} = 1 - (1 - \alpha)^{d_{new}/d_f}$$

6 Volume Visualization with Transfer Functions

In the current state your application allows basic volume visualization. The visualization follows the density function and directly maps density onto intensity and transparency. High density areas are displayed bright and opaque. Low density parts of the volume in contrast appear black and transparent.

In general, the function mapping scalar volume data to color and transparency is called the transfer function. We are using the identity function till now, which is only acceptable for a first impression of the volume data set. For a more detailed exploration of the volume, an adjustable transfer function is needed. To allow an efficient interactive analysis of the volume, the effects of changes to the transfer should become visible instantly.

Transfer functions can be very useful. One big advantage is, that noise—appearing most often in volume data sets—can be hidden. Moreover, certain features like bones or tissue can be highlighted in the visualization or uninteresting parts can be hidden. Transfer functions also allow to sharpen the edges at vascular borders to follow and highlight blood veins in MRI scans.

You could achieve changes of the transfer function by simply recalculating the scalar values before uploading the data texture to the graphics card. For such an approach it would be necessary to recalculate and upload the texture after each change. This would introduce severe latency into the visualization.

On the other hand, the programmable pipeline offers the possibility to do the mapping between the scalar volume data and color directly on-the-fly in the fragment shader. This is much more efficient. We only have to upload the mapping information stored in a second texture (which are only about 1000 color table entries) and don't need to upload the large volume texture (with possibly contains millions of data samples) every time we play around with the transfer function.

In the next chapters we will add color table functionality to our application.

6.1 The Volume Dialog

Being able to edit transfer functions interactively is an important aspect. For this purpose we included a nearly completed color table editor in the source skeleton package as part of the volume dialog. If you successfully finished the first part of the assignment you should be able to activate the dialog in the menu `File->VolumeDialog`. Part of the dialog is a menu bar which allows to load and save color curves. Four check boxes are located at the left. They enable the user to select the color curves that are affected by modifications. Additionally, the dialog contains two check boxes that control the rendering of the volume. Moreover, there is an `Apply-Button` which transfers state changes to the viewer, a `Reset-Button` that resets all settings, and a `Close-Button` that closes the dialog. You only have to add a few signal-slot connections and implement the missing functionality of the drawing area located at the right side of the dialog. The drawing area is implemented in an object of the class `DrawArea`. In the following you first have to add the missing signal-slot connections and functionality in the class `DrawArea`. Before doing so, start the example program. Play around with the dialog elements to become familiar with how they are expected to work.

Exercise 6: Various Representations

On the way to volume visualization you implemented the rendering of the slices as line loops and textured polygons. Connect the check boxes of the volume dialog to enable the user to switch between the volume and the silhouette representation. The slider should allow the user to change the slice distance.

Later, we are going to write a GLSL volume shader that adds lighting to the visualization. Properly connect the check box associated with this functionality to your `VolNode`.

6.2 The Drawing Area of the Volume Dialog

The widget will allow the user to modify the color curves—red, green, blue and alpha—independently from each other. The `DrawArea` already stores the curves internally as an array of color quadruples.

Exercise 7: Color Curves and Histograms

Implement the method `DrawArea::paintEvent`. This method is called every time when it becomes necessary to repaint the widget. You need an object of `QPainter` type to be able to draw directly into the Qt-window. Draw the four curves stored in the data fields of the array. Choose suitable colors. Also draw the histogram of the scalar volume values in the background. The histogram can be calculated with help of the function `VolNode::calcHistogram()`. As soon as a volume node is selected, the associated histogram data has to be transferred to the color table editor.

Exercise 8: Modifying the Curves

It should be possible to modify the curves with the mouse. Use the method `DrawArea::mouseMoveEvent` to implement the interaction. When modifying the color curves, linear interpolation needs to be applied to the y-values between neighboring mouse-events. Only the activated curves of the color table editor (check, which boxes are set!) must be affected. After a modification of a curve the updated color table values have to be sent to the selected volume node with help of the signal-slot mechanism. Then the volume node needs to update the color table texture.

6.3 Transfer Functions for Visualization

Now we are going to apply the transfer function which we created with the color table editor.

Exercise 9: Visualization with Color Tables

Therefore, create a 1D texture object of length `COLOR_TABLE_ELMS`, which is updated by the method `VolNode::setColorTable`. Don't forget to bind the texture prior to rendering! Texture coordinates are not needed this time, as we will calculate them directly in the fragment shader. Now change the fragment shader. It should output the color values you looked up in the color table texture instead of the intensity values.

6.4 Using Color Tables

Like mentioned above, many areas of application exist for the transfer function editor. In the previous sub exercise you got a feeling for the features of the color editor by playing around with it.

Exercise 10: Working with Transfer Functions

In this exercise you will model two color tables for two distinct data sets. Load the data set `00_data64x64x64.dat` containing a coordinate cross. Your transfer function has to map the three axes of the cross to different colors: x-axis red, y-axis green, z-axis blue. The spheres at the ends of the cross have to stay white! Store the color table in a file named `cross.table`. Now load the second data set (`Kopf_64x64x64.dat`). The brain needs to be extracted out of the head volume. Set its color to a nice bright green. Hide the other parts of the head by making them transparent. Store the result as `kopf.table`.

7 Volume Visualization with Lighting

The last exercise showed you the importance of lighting for getting a good impression of depth. That's why we would like to light the volume. In lighting geometric objects, vertex normals are used. In our case—of course—it does not make any sense to use the bounding box's vertex normals for lighting. We will use the volume gradients instead of that. At a voxel position $(x, y, z)^T$ the gradient of the scalar field \vec{s}

$$\vec{s}'(x, y, z) = \begin{pmatrix} \vec{s}(x+d, y, z) & - & \vec{s}(x-d, y, z) \\ \vec{s}(x, y+d, z) & - & \vec{s}(x, y-d, z) \\ \vec{s}(x, y, z+d) & - & \vec{s}(x, y, z-d) \end{pmatrix}$$

can be calculated with the help of central differences in a first approximation. Please observe that the step width d depends on the resolution of the data set.

Exercise 11: Lighted Volume

For lighting, the gradient has to be calculated on-the-fly in the fragment shader. You have to use another uniform parameter to tell the correct step width to the fragment shader. We want to implement a simple diffuse lighting model. Intensity, given by the transfer function, is used as the diffuse term. Additionally, the gradient enters the lighting equation as the normal. Don't forget to normalize the gradient. Now, diffuse lighting of the volume can be done analogously to standard per pixel lighting, which we came in touch with in the previous assignment. Keep in mind that the gradient is calculated in texture coordinate space! All the transformations applied to the volume's bounding box are not considered. This is why the light direction has to be transformed into the space of the normal

before evaluating the lighting equation. This is done by applying the inverse model view transformation onto the light direction. The volume should exhibit the same behavior as the geometric objects when changing the light's direction.

8 Rating Criteria

It is possible to achieve a total of 20 points in this assignment. Therefore the following criteria have to be fulfilled:

1 Point	The documentation of your source code is compatible to doxygen, extremely meaningful and done well in most of all respects.
1 Point	The viewer is extended like specified. The new menu entries have been added and the light source direction can be modified as described.
1 Point	The implementation of the scene graph is correct. Picking of the objects still works.
5 Point	The viewport-aligned slices are drawn correctly in back to front order. The distance of the slices can be modified from within the volume dialog. The bounding box is normalized and rescaled to a $-1...1$ range. A feature which allows to activate contour line drawing of the slices is implemented.
1 Point	The texture coordinates are set correctly. The boundary texel offset is respected.
4 Point	The volume can be viewed with a transfer function. Blending works and the blend function was chosen appropriately. Alpha correction is implemented. The 3D volume texture uses an appropriate texture format. Texture objects are used exclusively. Textures uploads to the graphics card only happen when their content changes (not every frame!).
3 Point	The color table editor works as specified. The color curves are drawn and displayed correctly and can be modified with the mouse in the specified way. The modification can be applied to individual curves, but as an alternative also simultaneously for all curves. The background of the editor shows the correct histogram of the data set. The transfer function also works at the boundaries of the data range.
3 Point	Diffuse lighting of the volume works. The gradients are calculated in the fragment shader. Thereby the resolution of the data set is taken into account. It is possible to enable/disable lighting in the volume dialog.
1 Point	Both sample color tables have been created. Their effect matches the one we described.

Precondition for grading is the error-free compilation of the program on the computers of the VISGS pool. There will be deduction of points if the program compilation leads to warnings (Warnings of QT in the style of “...has virtual functions but non-virtual destructor” are excluded therefrom)! The annotation of the source code should be that extensive to provide enough information to the author to explain the functionality of the code to the supervisors.