Karlsruhe Institute of Technology
Computer Graphics Group

Graphics Programming Lab
Assignment 3

Volume Visualization

Abstract

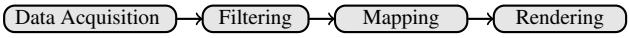
This assignment introduces you to the very basics of volume visualization. You will load data sets into your application that contain scalar density values for volumes. These density values are interpreted using a transfer function which maps them to RGBA values. Eventually, volumetric data sets are rendered using ray marching.

1 Introduction

As always, we will introduce you to the core parts of the assignment first. Note that giving a complete overview on visualization is far beyond the scope of this course, and we will only skim over volume visualization. Nevertheless, visualization is closely related to computer graphics and requires effective rendering techniques to create informative images. We encourage you also to consider a visualization topic for your final freestyle assignment.

1.1 The Visualization Pipeline

For the most part, visualization deals with processing large amounts of (possibly high-dimensional) data, and providing insightful visual representations thereof. This sequence of operations is summarized in *the visualization pipeline*. You will find several notions of this pipeline in the literature; we generalize these in the following way:



What kind of data has to be acquired is dependent on the purpose of a particular visualization task. The real world provides manifold data sources, and by using sensors, data such as temperature, velocity, or flow can be acquired. In contrast, synthetic data could be generated and used as well, which is useful for evaluating the result of simulations. Filtering fulfills two important tasks: first, only regard data you need and you are interested in; on the technical side, filter out noise and outliers due to measurement errors. Mapping is the most important step of the visualization pipeline: processed data is mapped to graphical primitives in order to provide an overall meaningful visual representation. Eventually, rendering is the part where computer graphics techniques are used to synthesize images.

1.2 Volume Visualization Basics

We deal with *volume visualization*, one of the many parts of the broad area of visualization, which creates images out of scalar data. (Higher dimensional data, such as vector data sets, are not covered in this assignment.) We project this 3D data set onto a 2D image plane to gain an understanding of the structure contained within the data [2]. Inspecting volumes and their characteristics is interesting for various scientific fields, such as physics, chemistry, and mechanical engineering; it is also very significant for medical diagnostics. In the field of medical visualization, data is usually obtained using methods such as X-ray computed tomography (CT) or magnetic resonance imaging (MRI).

1.3 Transfer Functions

We will discuss the purpose of *transfer functions* next. Let us stick to medical volume visualization on this occasion: we now have acquired data using CT or MRI scans, arranged in a three-dimensional grid, in the form of scalar values that are related to the density of tissue, bones, and organs. Assume we already filtered the data to diminish measurement errors. How do we project this data to the screen now? How should we interpret the data? To this end, we use transfer functions to map the density values to RGBA values.

Figure 1 gives an example. We start with plain scalar values that describe density; see the gray slice at the bottom left corner. Then, a transfer function maps the scalar values defined at each grid point of the volume to color and transparency. By including transparency, it becomes possible to hide uninteresting parts of the volume by making them transparent; for instance, one could hide soft tissue, yet emphasize bone structures at the same time. In Figure 1, the scalar values corresponding to the density of the dental crown are mapped to an opaque yellowish-white color. Soft tissue of the dental neck is mapped to a semi-transparent bluish color, resulting in an appealing outline. The dental nerve is colored red. See the final, composed outcome to the right.

Your application will support variable transfer functions that can be modelled by the user. A new Qt dialog will enable us to load, modify and save transfer functions. Moreover, the user can modify the functions interactively: changing the transfer function will have an immediate visual effect on the displayed volume data set.

1.4 Ray Marching

Now let's shift attention to *rendering*, the final step of the visualization pipeline. We choose *ray marching* as the image synthesis method, which allows us to sample the volume easily.

Figure 2 illustrates the concept of ray marching. Viewing rays spawn at the eye's position and intersect the image plane and the volume's bounding box. In contrast to standard ray casting, not only the rays' intersections with the objects have to be taken into

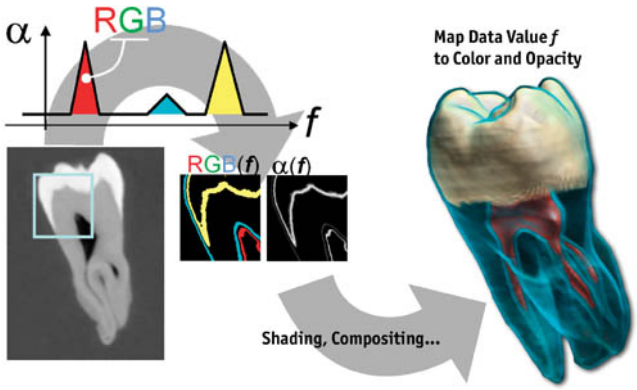


Figure 1: *The process of volume rendering as presented in the GPU Gems II chapter [3]: scalar values are mapped to RGBA values.*

account; we also 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.

A ray can be sampled in both opposing directions, starting at either the front or the back faces of a bounding box.

Back to Front: Here, sampling and integration starts at the back of the volume, proceeding to its front. In a simplified model this can be expressed by an iterative equation, the so called *compositing equation*: $C_{out} = C_{in}(1 - \alpha_k) + \alpha_k C_k$.

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.

Front to Back: Alternatively, you can start the ray marching process at the front faces of a volume's bounding box. Then, it is enough to accumulate values: $C_{out} = C_{in} + \alpha_k C_k$. However, it is very important not to override values with irrelevant values at the back of a volume. Therefore, you need to keep track of the current α-value of the ray: if you reach 1, you have to stop ray marching.

Recall that volume samples are not given in the form of emission- and absorption values, but as simple scalar values, such as attenuation in X-ray computed tomography. The transfer function provides the mapping between those scalar values onto intensity and transparency. Use a simple lookup table that specifies a color C_k and opacity α_k value for each scalar value contained in the data set. Upload this lookup table as a 1D texture to your shader later.

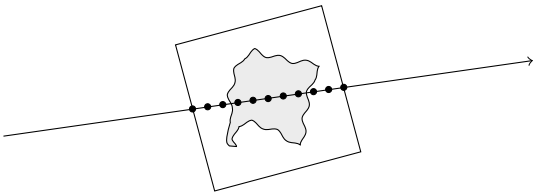
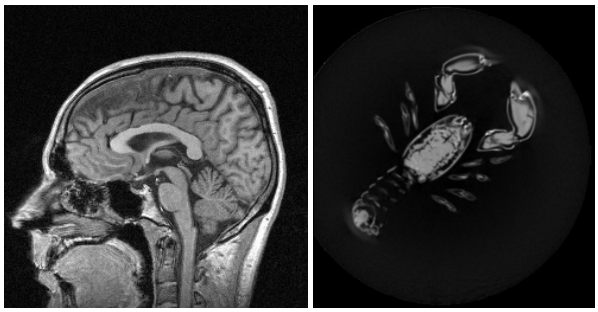


Figure 2: *Ray marching with constant step size through a participating volume.*

2 Loading Volumetric Data Sets

The starting point for our volume visualization is a three dimensional data set, which provides data given on a Cartesian grid. This means that a scalar value is given for each point of the grid, which is defined as a regular raster with a given X, Y, and Z resolution. The scalar value corresponds to the density of the volume at the associated 3D location. The scalar value of arbitrary points lying in between the samples of the volume are reconstructed by trilinear interpolation. (Your graphics hardware takes care of this for you.)

We provide you with two raw volumetric data sets: an MRI scan of a human head and a CT scan of a lobster contained in a block of resin [1]. Please make sure you open the files in the binary mode. The files are constructed as follows: in the first line, the resolution of the data set is specified, that is, the X, Y, and Z extent; in the second line, the aspect ratio of the Cartesian grid is specified, which defines the distance between to adjacent voxels in the X, Y and Z directions - and thus the size of the volume is the product of the extent and voxel distance in each direction. After the newline character, the raw data dump begins, which consists of 8 bit scalar values. There is also a tooth data set which has 16 bits (normalized unsigned shorts) per grid cell. Have a look at *slice planes* through these data sets to get a first impression of what they actually contain:



Implement a new scene graph node for volume visualization in your modelling application, which will be able to render volume data sets in a transparent way. Just like the other objects, the user should be able to pick, translate, and rotate the volume. When the user adds a volume visualization node to the scene, display a `QFileDialog` so he can select and open a raw volume data set.

In order to load a volume, parse the first two lines and then read the raw data into memory as unsigned chars. Loop over the Z resolution first, that is, as an outer loop, then over Y, and then over X as an inner loop. Create a 3D texture out of it with the correct resolution. Upload this 3D texture to your visualization shader later.

It is absolutely sufficient to support only one visualization node at the same time in your application; there is no need for you to support multiple volume nodes.

3 Volume Rendering

After loading a volume into memory, normalize its spatial extent in the following way: normalize the length of its longest side to 1 and also take the aspect ratio into account. (Recall that the aspect ratio is specified in the second line of a raw file).

There are no such things as rays in standard OpenGL. Rasterize the bounding box of a volume node, and then start ray marching in a visualization shader. Sample in the direction of your choice, starting either at the front or at the back faces, as previously described.

Before you have implemented an editor for transfer functions, use a hard-coded default transfer function first: map a scalar value $s \in$

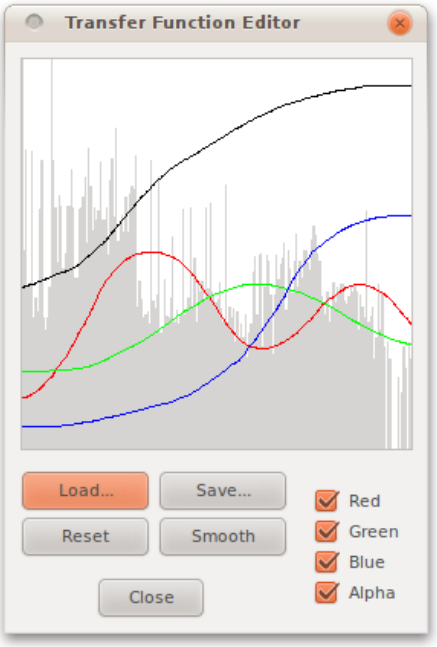


Figure 3: *Editor for transfer functions.*

$[0, 255]$ to the RGBA tuple $(s, s, s, s) \in [0, 255]^4$. In addition, evaluate the lighting model at each sampling step. (See Appendix.)

Implement an alternative visualization technique called *maximum intensity projection* (MIP). Provide a `QCheckBox` for enabling or disabling MIP. If MIP is enabled, sample the rays and keep track of the maximum scalar value encountered; evaluate the transfer function for these maximum values and color the fragments according to the RGBA results.

Implement your ray marching in one single render pass: use backface culling, intersect the cube with the ray in the pixel shader to get the distance to the intersection on the backface.

4 Transfer Function Editor

Create a new dialog (or `QDockWidget`) which provides the ability to customize the transfer function. See Figure 3 for an example. Create a new widget with a size of 256×256 and use a `QPainter` for customized drawing. Visualize the transfer function using a red, green, blue, and black curve for the RGBA co-domain. The scalar domain ranges from 0 on the very left to 255 on the very right.

Compute a *histogram* of the distribution of the scalar values of the volume. That is, create 256 bins and count how often a certain scalar value occurs within the data set. Display this histogram in the back of the new editor widget, as in Figure 3, and *normalize* it so that the maximum bin scales to the widget's height of 256.

Process mouse events on your new widget in order to allow editing the transfer function. Provide four `QCheckBox` widgets to specify which RGBA curves will be affected by the mouse event. If the left mouse button is pressed at position (x, y), set the values of the transfer function at x to y for all selected curves.

Upload the transfer function as a 1D texture to your shader, and access it like an ordinary lookup table. Use nearest filtering and don't offset the coordinate or use linear filtering and transform your coordinate to the $[\frac{1}{2n}, 1 - \frac{1}{2n}]$ interval.

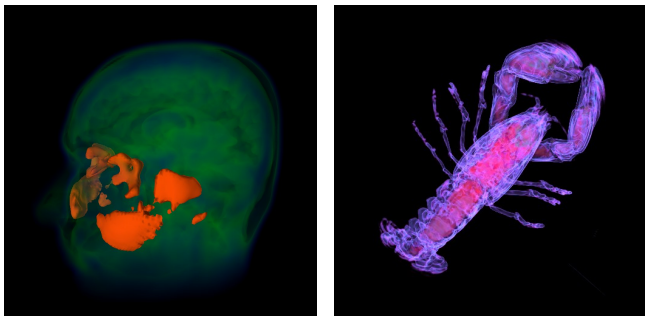


Figure 4: *Examples after applying custom transfer functions.*

Texture values are defined at texel centers. Thus, it is necessary to apply an offset of half a texel when accessing texture data.

After each mouse event, upload the current transfer function to your shader directly and render a frame; this way, you see changes reflected immediately. Note how the histogram in the back helps you to identify particular regions of the volume.

Provide ways to *save* and *load* transfer functions. Also offer a way to *smooth* the selected curves. (You could take a Gaussian weight of the local neighbourhood for all points into account.) Lastly, provide a way to *reset* the function to the default one described in Section 3. Again, only reset the curves that have been selected to be affected.

5 Wrap up Your Solution

Design two transfer functions, one for each data set, that yield nice looking results. See Figure 4 for an example. Make screenshots of both data sets visualized in your application, and save the corresponding transfer functions used.

Create a folder "results" and place your screenshots and the transfer functions there. Do not post processes your screenshots using image editing software. We should see the same results when we load your transfer functions into your application.

Appendix: Gradient Estimation

Lighting is crucial for getting a good impression of depth; that's why we would like to light the volume as well. In the previous assignment, you specified normals for the vertices or faces of your objects. Obviously, in this assignment it does not make any sense at all to use the normals of the faces of the bounding box for lighting. Instead, we will use the gradients inside the volume directly. At a voxel position (x, y, z) the gradient of the scalar field $\mathbf{V}$ can be approximated using *central differences*:

$$\nabla \mathbf{V}(x, y, z) \approx \begin{bmatrix} \mathbf{V}(x+h, y, z) & - & \mathbf{V}(x-h, y, z) \\ \mathbf{V}(x, y+h, z) & - & \mathbf{V}(x, y-h, z) \\ \mathbf{V}(x, y, z+h) & - & \mathbf{V}(x, y, z-h) \end{bmatrix}.$$

Observe that the step width h depends on the resolution of the data set—assign a meaningful variable to it, as it has a huge impact on shading. Moreover, normalize your estimated gradient afterwards.

6 Grading

You can achieve a total of 20 points in this assignment, based on the following criteria:

- 1 p You provide a `QFileDialog` to load volumetric data sets into a new scene graph node type.
- 1 p The new volume visualization node is displayed and can be translated and rotated like any other primitive. It can be correctly rendered in multiple viewports simultaneously.
- 1 p The user can pick the new nodes like all other primitives.
- 1 p You implemented a new dialog for editing transfer functions. A new custom widget sized 256×256 visualizes these functions.
- 1 p A histogram of the distribution of the scalar values within the volume data set is computed, normalized, and displayed at the back of the new widget.
- 1 p The user can select and edit the RGBA curves in the transfer function editor in a convenient way using the mouse.
- 1 p The user can save, load, smooth, and reset transfer functions.
- 9 p Volumes are ray marched and rendered using editable transfer functions. You observe texture bounds by applying correct offsets.
  - 1 p ray computation
  - 1 p ray marching is equidistant
  - 1 p ray marching correctly implemented
  - 1 p composition correct
  - 1 p - front-to-back: termination correct
  - 1 p - back-to-front: corretly weighted
  - 1 p 3D texture correctly used
  - 1 p texture offsets for 1D texture
  - 1 p RGBA values correctly read and applied
  - 1 p two nice looking editable transfer functions (for two different data sets)
- 3 p You account for lighting by evaluating gradients inside the volume using central differences and applying the lighting model at every sampling step during ray marching.
  - 1 p gradient computation implemented
  - 1 p good value for h
  - 1 p applying gradient correction at every step of ray marching for hong computation
- 1 p You implemented *maximum intensity projection* as an alternative visualization method and provide a `QCheckBox` to enable or disable it.

You get a *malus* if your code is not nicely written using meaningful variable names in proper English, or significantly lacks proper comments! Moreover, you get a malus if your solution produces errors or way too many warnings during compilation! Lastly, you lose points if your program crashes or freezes for weird reasons.

If we got the impression that you handed in copy-pasted code, and you are unable to explain your solution, we will expel you from the course!

References

[1] Computer Graphics Group, University of Erlangen. The Volume Library. `http://www9.informatik.uni-erlangen.de/External/vollib/`.

[2] HyperVis. Volume visualization and rendering. `http://www.siggraph.org/education/materials/HyperVis/vistech/volume/volume.htm`.

[3] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *GPU Gems II: Volume Rendering Techniques*, pages 667–692. Addison Wesley, 2004.