Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Group 96
Postbus 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Molon Mattia (#1511866)
Remondini Leonardo (#1518550)

Responsible Lecturer
Dr. Michel Westenberg

Date
December 8, 2019

Volume rendering

Molon Mattia (#1511866)
m.molon@student.tue.nl

Remondini Leonardo (#1518550)
l.remondini@student.tue.nl

Technische Universiteit
Eindhoven
University of Technology

Table of contents

Title
Volume rendering

1 Ray function implementations

For the first part of the assignment, we had to extend a Python code that supported slice-rendering only, implementing different volume rendering techniques based on a ray-casting approach. For every pixel **p** in the final image, one ray **R** perpendicular to the view plane, is cast on a data volume **V**. In order to be visualized, the voxel values intersected by the ray are combined via a ray function **F**.

To test our rendering functions, we used the "orange dataset", which consists of a volumetric dataset containing scans of an orange. The data is given as a tridimensional matrix, where each triplet (x, y, z) of coordinates represents a voxel of the volume. We implemented three new features in `implementation.py`: trilinear interpolation, maximum energy projection (MIP), and composite volume rendering.

1.1 Trilinear interpolation

Since voxel values are represented in a tridimensional matrix, their positions are discrete. Therefore, the values of their coordinates (x, y, z) must be integers in order to be representable. As a consequence, whenever we want to render them through a ray-casting technique, we might try to visualize a triplet of coordinates that has no matrix representation. Hence, to best approximate the values of a non-integer triplet of coordinates, we implemented trilinear interpolation based on the following formulas:

$S_x = (1-\alpha)(1-\beta)(1-\gamma)S_{x_0} + \alpha(1-\beta)(1-\gamma)S_{x_1} + (1-\alpha)\beta(1-\gamma)S_{x_2} + \alpha\beta(1-\gamma)S_{x_3} + (1-\alpha)(1-\beta)\gamma S_{x_4} + \alpha(1-\beta)\gamma S_{x_5} + (1-\alpha)\beta\gamma S_{x_6} + \alpha\beta\gamma S_{x_7}$

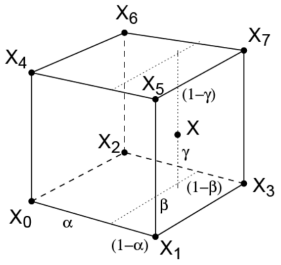

Figure 1.1: Trilinear interpolation cube.

S_{xi} represents the value of the voxel i in the cube that surronds the casted triplet, S_x represents the final interpolated value of the casted triplet, and the other parameters of the formula are the absolute distances between S_x and the S_{xi} of the cube of voxels. A good representation of the variables can be found in image 1.1. In our code, the implementation of the trilinear interpolation can be found in the `get_voxel()` function.

1.2 Maximum Energy Projection (MIP)

Maximum Energy Projection (MIP), is a technique that consists in projecting in the view plane the maximum voxel value intersected by each casted ray. In order to implement this rendering approach, we had first studied the already built-in slice rendering function that can be summaries in three main points:

1. Adjust the volume coordinates to be centered in the viewplane.
2. Compute the values of each voxel in the central slice of the volume
3. Assign to each voxel a color form a greyscale of colors

In terms of code implementation, the MIP technique is really similar to the `render_slicer()` function. We have taken its structure and have introduced a loop along the z axe. This loop lets us compute the maximum value that can be found along each casted ray. However, as the user changes the rendering viewpoint, the depth of the volume changes too together with the range of the z axe. Therefore, In order to always consider the correct depth of the volume, we have bounded the range of the loop

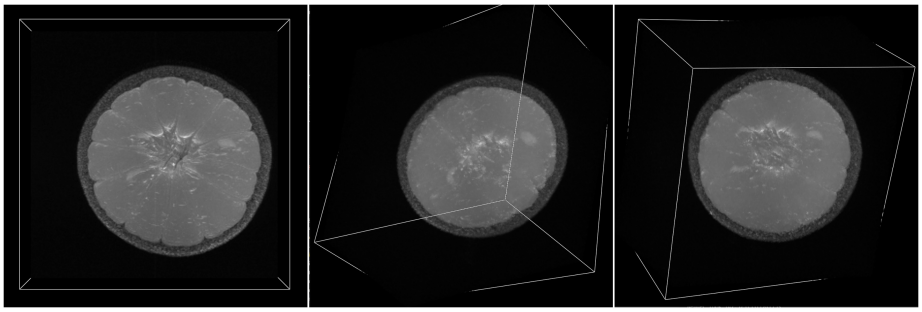

Figure 1.2: MIP renderings from 3 different rotations of the volume

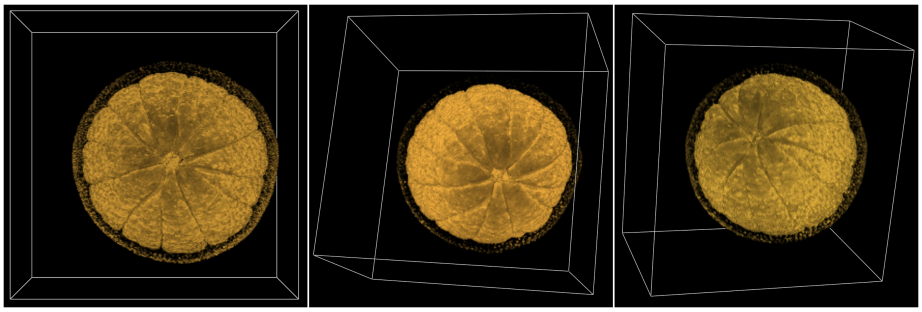

Figure 1.3: Composite rendering from 3 different orientations of the volume data

between `-half_diagonal` and `half_diagonal`, as the z axe cannot be longer than that value. Once we find the maximum intensity voxel, we map it in a greyscale according to its interpolated value and visualize it in the view plain.

This technique is computationally fast, but the 2D results do not provide a good sense of depth of the original data. For example, figure 1.2 shows how MIP, in the orange dataset, does not give a sense of rotation of the object.

1.3 Composite volume rendering

The composite volume rendering is part of direct volume rendering. It requires every sample value to be mapped to opacity and color, which can be done with a transfer function already implemented in the given framework. The algorithm works recursively: For each pixel of the final image, a ray perpendicular to the view plane is cast. Starting from the voxel located at the back of the volume, the final color and opacity of each voxel are calculated by the following function

$$C_i = \tau_i c_i + (1 - \tau_i) C_{i-1} \tag{1.1}$$

where C_i describes the final color of the voxel, the tuple (c_i, τ_i) describes the initial color and opacity setting of the voxel assigned by the transfer function, and C_{i-1} describes the final color of the last visited voxel. At the end of the recurrency, every pixel of the final image will take the value of the corresponding front voxel, as it describes a composition of the color and opacity of all voxels intersected by the ray. The implementation of composite volume rendering differs from the MIP and the slicer rending technique for the rendering criteria, but the general structure of the function is maintained. Moreover, when it comes to the final image, the composite technique emphasizes the shape of the object and provides it tridimensionality. Some examples of our implementation's outcomes are shown in Figure 1.3. The code can be found in `render_compositing()`

2 Real use case scenario

In the second part of the assignment we had to test our rendering functions on a real use case scenario. We analyzed a subset of the dataset of The Scientific Visualization's 2013 community challenge, with the goal of identifying spatial and temporal gene expression patterns in developing mouse brain. Specifically, the dataset set tracks on the level of gene expression for ~2000 genes in a 3D mouse brain from embryonic stages through adulthood, and it is divided in two groups :

1. A matrix of brain annotations: It describes a set of brain sections that are custom-labeled to emphasize structural boundaries. It has been provided a matrix annotation per temporal stage of the brain
2. A matrix of gene activation for each gene per temporal stage : They describe the intensity of the genes' expressions within the brain.

Our analysis is divided in three phases. First, we visualized brain annotations only; Second, gene activations only; and finally, both of them simultaneously. During our experimentation, we noticed that some brain sections have tiny ranges. Therefore in all tests, we introduced the possibility to visualize the firsts n desired biggest brain annotations, since it can be unclear and confusing to visualize them all together.

Each visualization technique is accessible through the function `render_mouse_brain()`. The selection of different brain annotations is customizable inside the `volume.py` file via the variables `n_structures`.

2.1 Visualization of brain annotations

We developed two different ways to visualize brain annotations; The first method consists in visualizing only the structural boundaries in grayscale; The second one, aims to color different sets of brain sections distinctively, using the color palette provided by the `structure.csv` file.

To retrieve the structural boundaries, we computed the gradient in respect to a derivation of the annotation volume since we don't want the id-s of the annotations to influence the gradient computation. Therefore, we first set every annotation that we don't want to visualize to 0 and to 10000 the ones we do want to visualize; then we compute the magnitude of the gradient for each voxel of the volume. The gradient is approximated via the following equation:

$$\nabla S \approx \frac{1}{2} \begin{pmatrix} s(x+1,y,z) - s(x-1,y,z) \\ s(x,y+1,z) - s(x,y-1,z) \\ s(x,y,z+1) - s(x,y,z-1) \end{pmatrix} \tag{2.1}$$

We chose the range $[0, 10000]$ since the biggest the range is, the higher the contrast within boundaries is going to be.

Once we obtained the magnitude volume, we applied greyscale composite rendering as final image. The results show a clear distinction between the different sections. Brain boundaries are emphasised and the overall image resolution is crisp. Some examples are represented in Figure 2.1.

We passed then at the second type of annotation visualization technique. We managed to visualize each brain annotation with the color association given by the `csv` file. Since we were not interested in boundaries anymore, we didn't compute the gradient nor used an annotation volume derivation. Indeed, we used the given id-s to link them to their associated color and we gave each of them an alpha channel value of 0.5. To visualize the final representation of the brain it has been used the composite rendering technique. As we can see in Figure 2.2, the results show that the boundaries within different sections are not as crisp as before, but if we look at different temporal stages, we can now clearly see the development

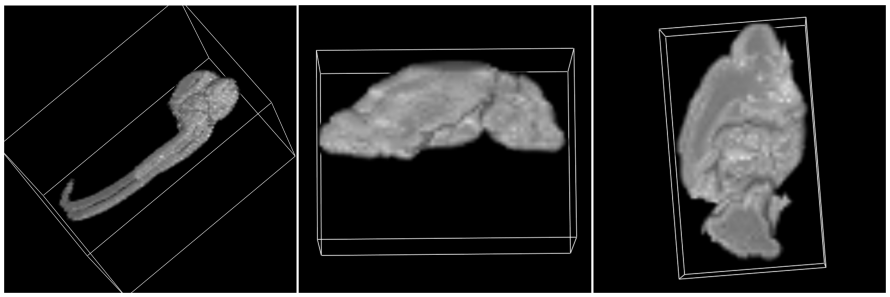

Figure 2.1: Grayscale composite rendering of brain annotations boundaries.

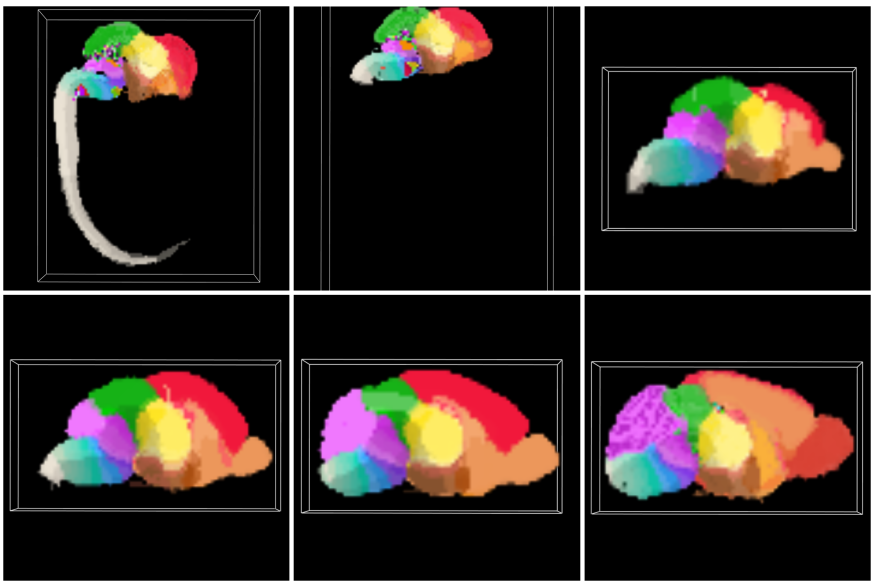

Figure 2.2: Colored brain annotations through different stages of the neurological development of the mouse brain. It goes from the youngest (upper left corner), to the oldest one (bottom right corner)

.

of the different parts of the mouse brain. It wouldn't have been easy to notice by using the previous technique.

The implementation of both techniques can be found in `visualize_annotations_only()`. The technique can be switched via the variable `csv_colors`.

2.2 Visualization of gene activations

In order to visualize different gene activation distributions within the same brain annotation, we decided to assign to each gene activation a different color as we performed in the previous section. Every time a new energy function is added to the image view, the rendering procedure is repeated and the selected gene activation is added in the energy volume dictionary. To determine the RGB color of each voxel, we performed a weighted average among all the colors of the gene activations present in that voxel. The weight of a gene activation is determined by the ratio between the intensity of that energy in that voxel and the maximum intensity of that energy among all brain's voxels. Furthermore, for every voxel we set its alpha channel value as the maximum intensity percentage we detect among all energy functions present in that voxel.

When all voxels' color are determined, we rendered the volume via composite rendering. Moreover, in our implementation that can be found in `visualize_energies_only()`, we give the possibility to

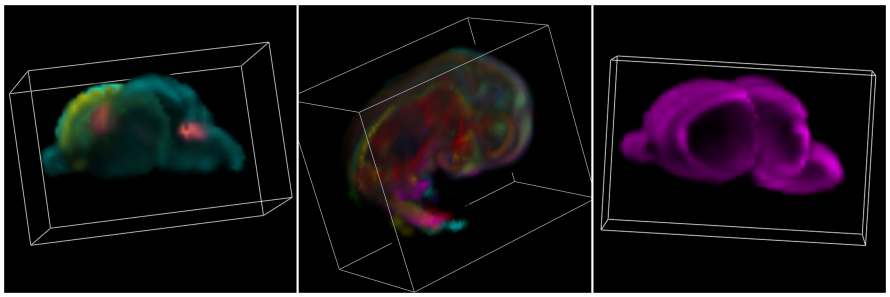

Figure 2.3: Colored composite rendering of various gene activations within the same brain annotation.

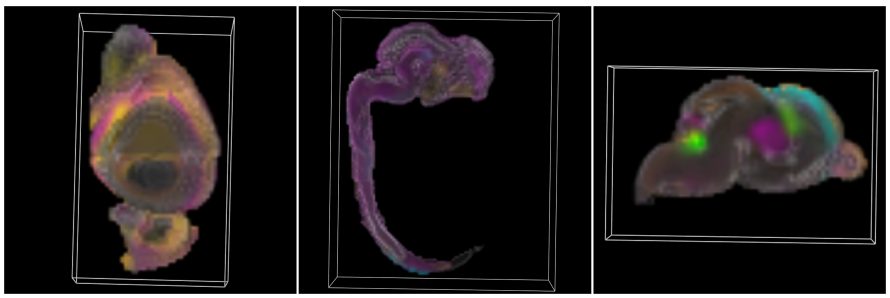

Figure 2.4: Combination of both techniques. It shows many gene activations within the structural boundaries of the selected brain annotation.

filter the energies by the selected brain structures.

Our techinque, seems to be accurate and clear even when multiple energies are rendered simultaneously, as we can see in Figure 2.3.

2.3 Combining brain annotations and gene activations

Concluding, we combined the rendering techniques we used in sections 2.1 and 2.2 to visualize multiple gene activations within the same brain annotations, while emphasizing the structure boundaries. Therefore the overall process is the following

1. We compute the approximate gradient on the derived annotation volume, as described in section 2.1, of the selected brain sections.
2. Separately, we merge the RGB colors of the selected gene activations by the weighted average computation described in section 2.2
3. Then, for each voxel we check if it is part of a boundary (e.g. the gradient is grater than 0) or not. If it is, we assign to the voxel a color equal to the average between the color's value of the boundary and the final RGB gene activation color in that point. If it isn't, we assign to it the RGB color of the gene activation only.
4. Finally, we compute the composite rendering technique on the RGB voxel values just calculated.

As we can see in Figure 2.4, it's now possible to locate more accurately every gene activation to a brain section without loosing image resolution. The implementation of this technique can be found in `visualize_both_energies_annotations()`.