Prof. Dr. Michael Krone
Marco Schäfer
Marcel Bok
Big Data Visual Analytics in Life Sciences
Department of Computer Science & IBMI
Faculty of Science

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Scientific Visualization **SS/2021**
Assignment 9 **Due: July 22th, 2021, 9:00 am**

Code skeleton:

We supply you with a basic code template to use for this assignment but you are free to use the code you wrote for assignment one as a basis for this task. Our supplied code will contain code snippets assisting you in solving the tasks. So be sure to look at the supplied code if you choose to use your own code as a basis for this assignment.

This is a two week assignment (with 30 points!): We strongly recommend you starting to work on this assignment early, as the concept of volume rendering is nontrivial and so is its implementation! If you run into any dead ends you can ask us your questions and still have enough time to successfully apply it to your solution!

1 Volume Renderer (30 pts)

In the last assignment you were tasked to read in a two dimensional data set (an image) to apply vertical scaling to a mesh of vertices. In this two week assignment you are tasked to implement a volume renderer. A volume renderer uses data from a three-dimensional data set (usually scalar values stored as a rectangular or uniform 3D grid) to calculate projections of volumetric objects. The data sets consist of density values of these objects, for example as generated by a computed tomography. The difficulty of volumetric rendering is that for the value of a given pixel, the densities associated with every voxel of the volume along the camera ray must be considered (Figure 1).

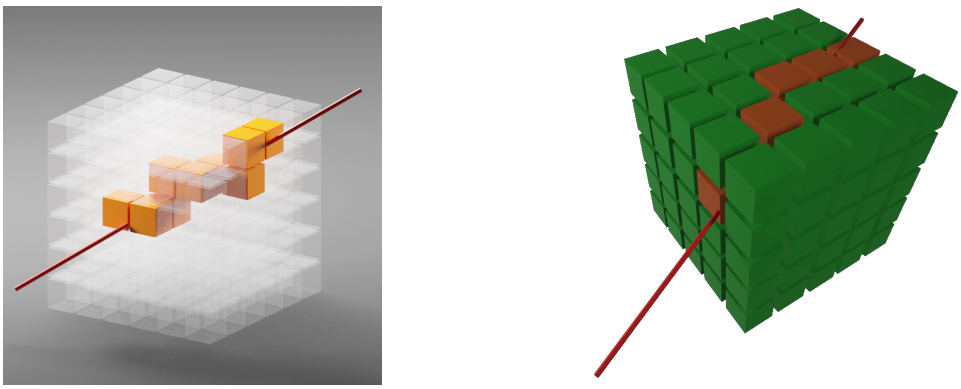

Figure 1: Ray through a cubic volume that intersects the emphasized voxels. The orange voxels indicate intersection of ray and voxels and thus must be taken into account for the shading.

For this assignment, the code skeleton contains a file reader, that reads and parses the binary

volume data. To load the data, use the file loading button on the HTML page and load one of the `.dat` files found in the `assets` folder. These `.dat` files contain the name of the corresponding binary-data file (*.raw), as well as the resolution (i.e., the number of voxels in each direction). Both the binary data and the data info `let datFile` are generated after loading and should be used going forward. The raw binary data, for example, must be converted into a `THREE.DataTexture3D`.

The assignment is partitioned into the following tasks:

- Generate a Cuboid to project the volume into. (3 Pts)
- Writing the Vertex Shader (2 pts)
- Writing the Maximum Intensity Projection Fragment Shader (10 pts)
- Writing the DVR Fragment Shader (5 pts)
- Applying a Transfer Function in DVR (2 pts)
- Adding Lighting to the DVR (3 pts)
- Writing the Isosurface Shader (5 pts)
- BONUS: Volume Rendering with Non-Uniform Voxels (2 pts)

1.1 Generate a Cuboid to project the volume into. (3 pts)

In order to perform volumetric rendering, a geometry has to be generated on which to project the volume. Generate a buffer geometry with eight vertices and twelve triangular faces with their normals pointing outwards. The dimension of the cuboid should fit to the resolution and aspect ratio of the 3D texture (volume), i.e. the longest edge should have a length of 1 with the smaller edges being scaled accordingly. The cube should be centered around the origin of the coordinate system. When using the default code skeleton, you should see a red cuboid (Figure 2).

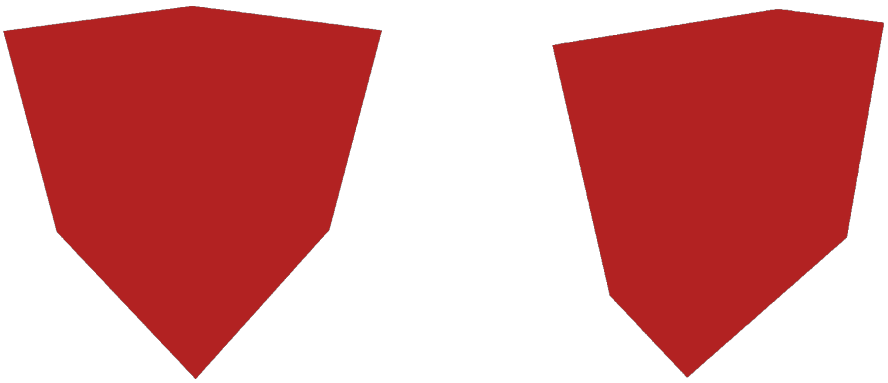

Figure 2: Default Setup: Static color value mapped to fragment position

Analogous the UV coordinates used in assignment 8, the shader does need additional per vertex information in order to correctly map the volume texture onto the cube. Add a custom attribute (texCoords), mapping values from 0.0 to 1.0 in X, Y and Z direction, i.e. the left-bottom-front corner should be (0.0,0.0,0.0) and the right-top-back corner should be (1.0,1.0,1.0). When mapping these texture coordinates onto an RGB color, with X, Y and Z mapping to R, G and B respectively, you should see a correctly interpolated "Rainbow Cube" (Figure 3).

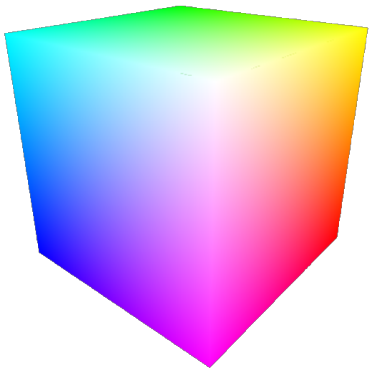

Figure 3: For buckyball data set generated geometry (box). The texture coordinates are mapped to color.

1.2 Writing the Vertex Shader (2 pts)

The basic idea of volumetric rendering is to have a camera-ray which intersects with the voxels of the 3D texture and sample values along this ray in the fragment shader. In order to calculate this ray we need two positions, the camera position, in world coordinates, and the interpolated vertex positions – also in world space. While the world camera position is available as a uniform (`vec3` cameraPosition) in `THREE.js` the interpolated world vertex position is not.

In order to write a functioning vertex shader (code skeleton: `/src/vertexShader.vert.js`):

- Add a variable to be interpolated for the fragment shader containing this vertex position.
- Add a variable to be interpolated for the fragment shader containing the texture position.

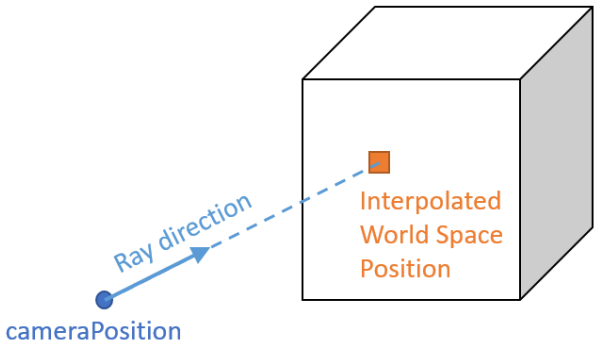

Figure 4: Calculation of ray direction with world cameraPos and worldSpace position

1.3 Writing the Maximum Intensity Projection Fragment Shader (10 pts)

The easiest volumetric shader utilizes maximum intensity projection to assign the pixel of the highest intensity sampled along the camera ray. Write such a maximum intensity projection fragment shader utilizing a set step size to sample the 3D texture along the camera ray finding the voxel of maximum density.

For this you will have to complete the MIP fragment shader (code skeleton: `/src/mipShader.frag.js`) by:

- Calculate the ray direction (Figure 4). If your calculation of the ray is correct you should see the cuboid in Figure 5 on the left, when mapping the ray components to a color. By using the absolute values of the ray direction you will see the right cuboid(Figure 5).
- Choosing a reasonable starting point for sampling
- Choosing a reasonable step size and amount of steps
- Sample the density values along the ray and save the maximum value
- Set this maximum value (grey scale) as fragment color
- Make screenshots of the Buckyball and the engine (Figure 6)

Tip: think about the space in which the steps/sampling occurs: you might need the `uniform vec3 u_volumeTexSize;` variable for this.

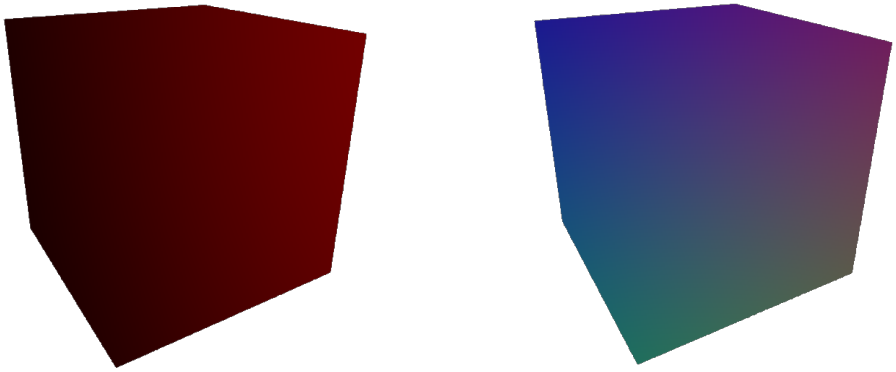

Figure 5: ray direction mapped to color (i.e. R = Ray.x, G = Ray.y, B = Ray.z, Alpha = 1.0) (left) and to abs(Ray)(right).

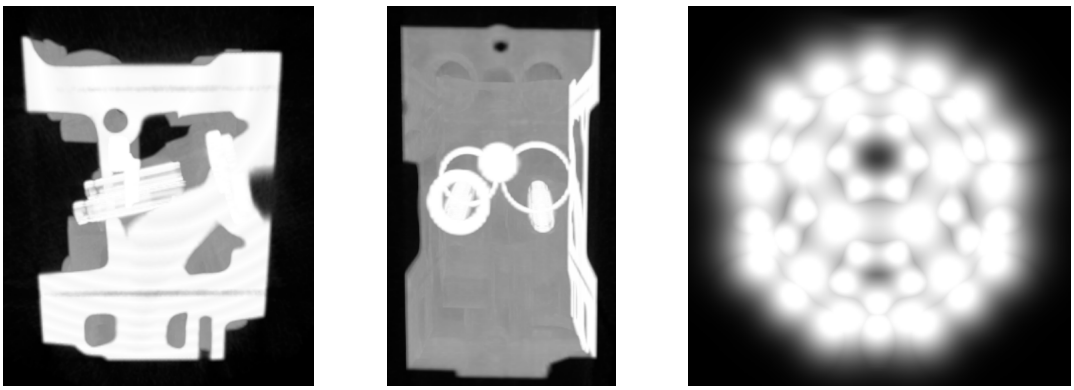

Figure 6: Engine (unrotated and rotated by 90 degree along y-axis) and Buckeyball in MIP

1.4 Writing the DVR Fragment Shader (5 pts)

In a next step this basic volume renderer will be adapted to perform direct volume rendering (DVR). In contrast to the maximum intensity projection, the DVR shader uses all information sampled along the ray and composites them.

For this you have to:

- Copy your code from the MIP into the DVR shader
  (code skeleton: `/src/dvrShader.frag.js`)
- Sample along the ray and accumulate values (Front-to-Back-Compositing)
- Use these accumulated values as fragment color
- Make a screenshot of the result for the Buckyball, the Engine and the Aneurysm (Figure 7)

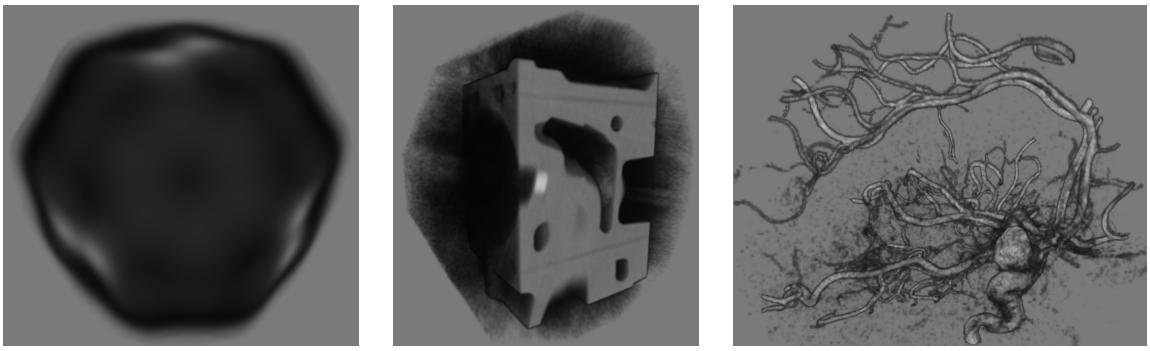

Figure 7: DVR: Raw density values in R,G,B and A channels accumulated

1.5 Applying a Transfer Function in DVR (2 pts)

In a next step you will adapt the DVR shader to utilize a transfer function mapping color values to the densities

For this you will:

- Use the values sampled along the ray to read from a RGBA transfer function
- Accumulate the RGBA colors and use them as fragment colors using Front-to-Back-Compositing
- Make screenshots of the three models (Figure 8)

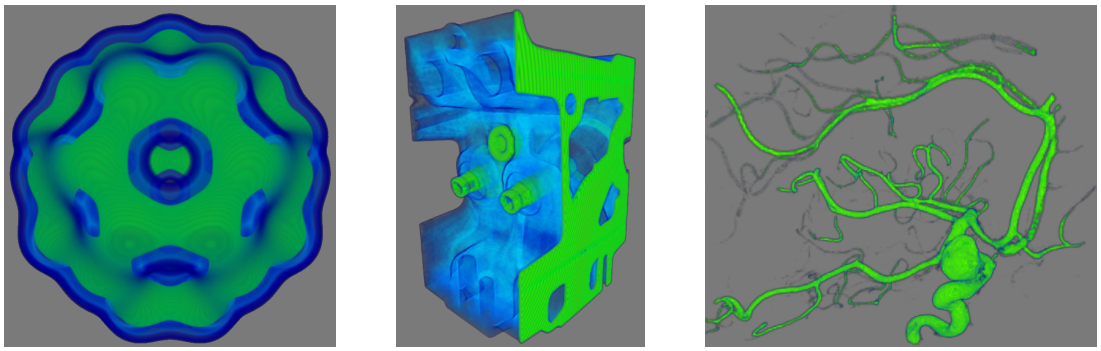

Figure 8: DVR: R,G,B and A channels as mapped from a transfer function, accumulated (buckyball, engine, aneurysm)

1.6 Adding Lighting to the DVR (3 pts)

Analogous to regular surface rendering, lighting can be used for the DVR as well. In this case you are tasked to implement a basic lighting without specular highlights (i.e. only ambient and

diffuse terms) and with a fixed ambient intensity. Furthermore we assume that light direction is equivalent to the camera ray direction.

To implement this:

- calculate the normals at each fragment position using the central differences method
- use a fixed ambient light intensity of 0.25.
- add a toggle to turn the lighting on and off.
- Make screenshots of the three models (Figure 9)

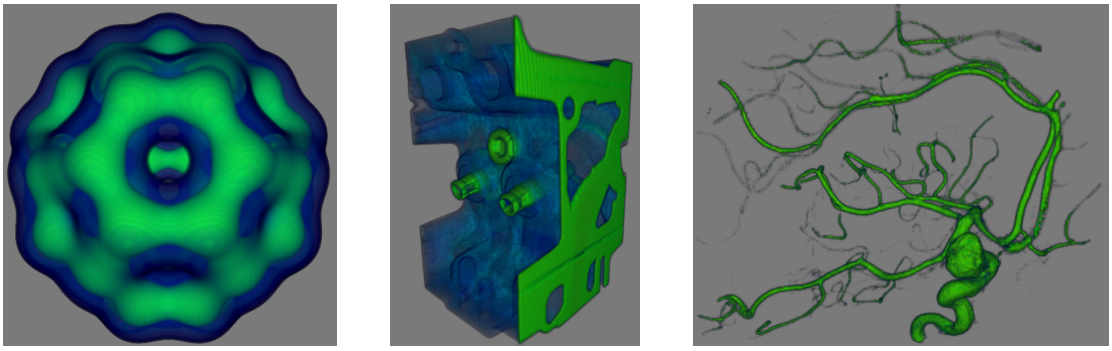

Figure 9: DVR: R,G,B and A channels as mapped from a transfer function, accumulated – ambient and diffuse lighting added

Tip: Think about the allowed range of values for the lighting calculation (for the edge cases for the normal/camera-ray interaction).

1.7 Writing the Isosurface Shader (5 pts)

For the third and final shader you have to adapt the DVR-shader to render an isosurface with a supplied iso value. An isosurface is, analogous to the 2 dimensional isoline, a surface in which a specific metric is the same. In our case the isosurface should visualize in which a specific density value – the isovalue – is exceeded. To implement the isosurface shader you have to perform the following tasks:

- Copy your code from the DVR shader into the isosurface shader
  (code skeleton: `/src/isoShader.frag.js`)
- Introduce the `uniform u_isoValue` variable controlling the desired isovalue
- Detect the isosurface and assign it the static color (1.0,0.5,0.0)
- Apply the lighting used in the DVR to the isosurface
- make screenshots of the engine and the buckyball for an isovalue of 0.5 (Figure 10)
- adapt the shader to allow semi-transparent isosurfaces with their transparency defined by the `uniform float u_isoAlphaValue;` variable.
- make screenshots of the engine and the buckyball for an isovalue of 0.5 and an alpha value of 0.5 (Figure 11)

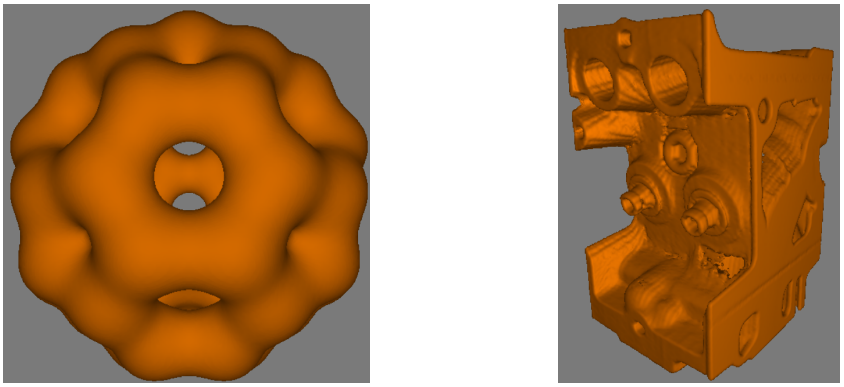

Figure 10: Isosurfaces of the buckyball and engine with an isovalue of 0.5, opaque

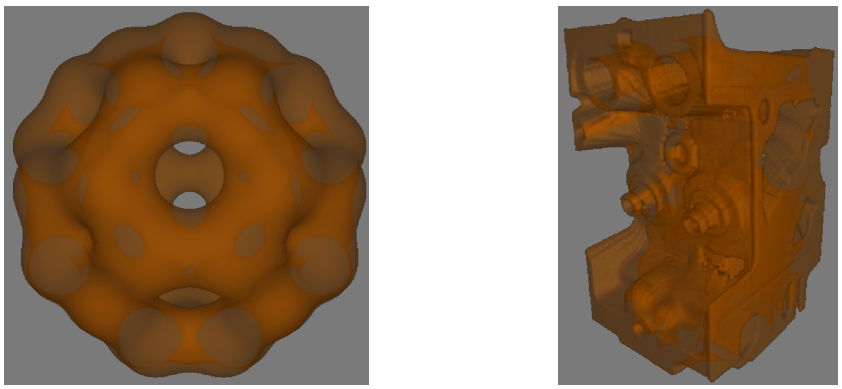

Figure 11: Isosurfaces of the buckyball and engine with a isovalue of 0.5, alpha of 0.5

1.8 BONUS: Volume Rendering with Non-Uniform Voxels (2 pts)

While the Engine, Aneurysm, and Buckyball data sets have uniform, cubic voxels, the Bunny data set has rectangular voxels (i.e., cuboids). While this is not an issue for the texture fetches in the shader, you have to take this into account when generating the cuboid that represents your volume. Hint: the slice thickness (i.e., the extent of the voxels) is also stored in the *.dat* file and is already read in by the provided file loader. Apply the necessary changes to your cuboid rendering and your shaders to display the Bunny data set.

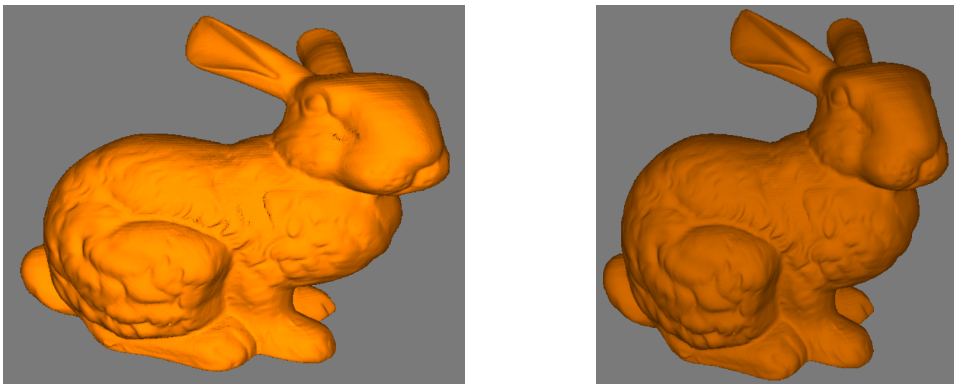

Figure 12: Bunny without compensation (left), with compensation (right)

Handing in

For the Hand-in, write the names of both group members at the top of all code files and PDF documents. Create a ZIP archive of your project folder and **PLEASE EXCLUDE** the folder **NODE_MODULES**, the binary volume textures as well as the **main.js** file from your archive. Name it with the last and first name of each group member, and the assignment number (e.g., *schaefer_marco___bok_marcel_assignment9.zip*) and upload it to `Assignment_9`.