Volume Rendering Assignment
2IMV20
Technische Universiteit Eindhoven

Patrick Strookappe 0896169 Jakob Serlier 1645757 Wouter van Oorschot 0906557

Group 67

-

December 13, 2020

1 Introduction

Visualisation methods are an increasingly important topic of research due to their importance in fields like medical imaging. In this report we we will discuss multiple classical and state of the art visualisation techniques. We start by discussing the theoretical foundation of each technique. Then we discuss our implementation and show how our results compare to the examples in the assignment and demo video. Finally in the Analysis we discuss the advantages of each technique and compare them with each other using interesting visualisations.

2 RayCasting

2.1 Tri-linear interpolation

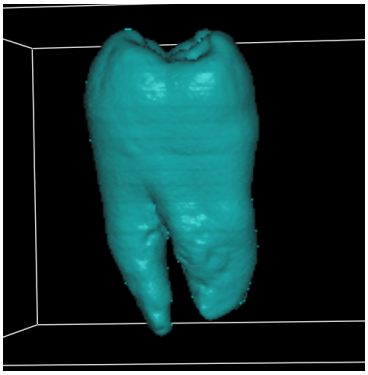
Instead of the standard nearest neighbour approach to retrieve voxel values, we use tri-linear interpolation for all visualisation techniques.
Similarly to `getVoxel()`, the first step of the interpolation checks whether or not the coordinates are inside the volume. However, because we now also use `ceil()` values of the coordinates, an index out of bounds exception might occur. This coordinate-inside-volume check was slightly adjusted (by [-1,-1,-1]), which prevents index out of bounds at the extremes.
Next, the 8 vertices of integer coordinates, representing a surrounding cube, are calculated by taking the`floor` and `ceil` values of the original 3 coordinate points. These points of the cubic lattice are labeled v_{000} to v_{111}.
Using the tri-linear formula described in the lectures we can now calculate the desired interpolated value:

$$\begin{aligned} val = &(1-\alpha)(1-\beta)(1-\gamma)V_{000} \\ &+(\alpha)(1-\beta)(1-\gamma)V_{100} \\ &+(1-\alpha)(\beta)(1-\gamma)V_{010} \\ &+(\alpha)(\beta)(1-\gamma)V_{001} \\ &+(1-\alpha)(1-\beta)(\gamma)V_{110} \\ &+(\alpha)(1-\beta)(\gamma)V_{101} \\ &+(1-\alpha)(\beta)(\gamma)V_{011} \\ &+(1-\alpha)(1-\beta)(\gamma)V_{111}. \end{aligned} \tag{1}$$

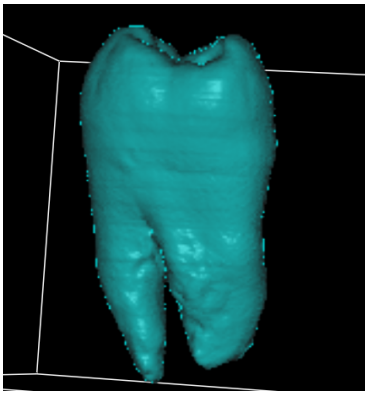
For the formula above, α, β, and γ represent the distance between the coordinate values and the rounded floor values. Figure 1a shows a subtly sharper image, with less artifacts, as opposed to 1b, confirming that our implementation is correct. Further methods and images shown below use tri-linear interpolation where possible. Tri-linear interpolation for gradients is discussed in section 3.2.1.

2.2 Compositing ray functions

The compositing ray function is created using the same basis as the MIP-raycasting function. For each step on the rayvector the voxel value is calculated using tri-linear interpolation. The transfer function then maps this value to a final value, based on the parameters we set in the transfer function. All these transfer function colors along the ray are added using the front-to-back version of the compositing function from the 1995 optical model paper [5],



(a) Tooth dataset using tri-linear interpolation, isosurface raycasting (ISO 500) and Phong shading: little artifacts.



(b) Tooth dataset using nearest neighbour, isosurface raycasting (ISO 500) and Phong shading: increased artifacts.

Figure 1: Difference between trilinear interpolation and nearest neigbour, with isosurface raycasting and phong shading.

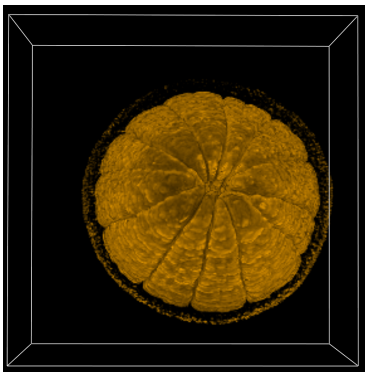
which is simplified in the lecture to equation 2. In the implementation (equation 3) we calculate each new color C_i in the ray using a color c_i, based on the voxel color and transfer function, the current opacity and the previous color C_{i-1}. Additionally, α_i is the opacity of the current voxel and calculated using equation 4, which we get a from the transfer function. Equation 4 uses a similar approach to equation 3 where it iteratively calculated the new α based on the previous value.

$$I(p) = \sum_{i=0}^{n-1} c_i \prod_{j=i+1}^{n-1} (1-\alpha_j). \tag{2}$$

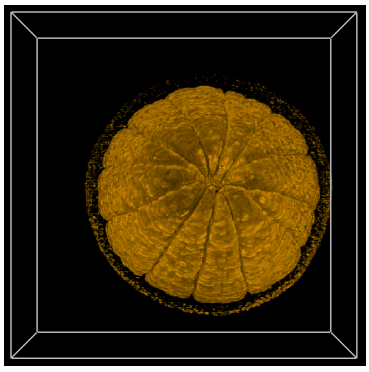
$$C_i = \alpha_i * c_i + (1-\alpha_i) * C_{i-1} \tag{3}$$

$$\alpha_i = \alpha_{i-1} + a_i * (1-\alpha_i) \tag{4}$$

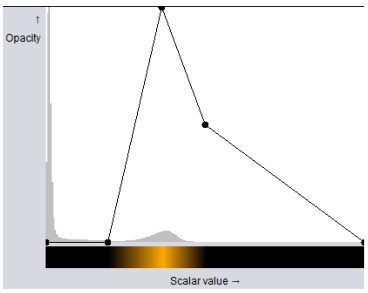
The composite function is compared to the examples given in the assignment. In this case the Orange8 dataset. As shown in image 2 the results are visually nearly identical.



(a) Drawn composite image of the orange



(b) Composite image from the assignment



(c) Transfer function for figure 1b

Figure 2: Orange comparison

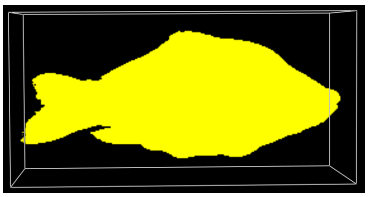
2.3 Resolution optimizations

The raycasting calculations are too heavy to allow for a responsive user experience when moving or rotating the object by default. Because of this a global boolean `InteractiveMode` indicates if the user is manipulating the view area. If this is the case, both the `sampleStep` and the `increment` value are changed from 1 to 3. The `sampleStep` value lowers the amount of steps calculated over a casted ray. The `increment` value determines the amount of rays casted per pixel on screen. When this is increased, the output resolution of the image is lower. This results in a fluent user experience, but still clear enough to be able to see how the object is rotated or translated.

3 Isosurface Raycasting and Shading

3.1 Isosurface Raycasting

The implementation of `traceRayIso` largely follows the existing implementation of `traceRayMIP`. Following the ray which is cast, we determine the increments and number of samples and initialize a current position. This current position changes when we move along the raycast. Next, as the current position starts moving along the ray in steps, the voxel-value is compared to the `isoValue` (`isoValueFront` or `isoValueBack`). When the voxel-value is greater than the user-determined `isoValue`, the voxel needs to be drawn and the opacity is set to 1, and iterating over the position stops. Then the isocolor is pulled from the UI. Allowing the isovalue to change is therefore what allows for interactive rendering of different sections of the volume.
The above method follows isoSurface rendering using raycasting, in contrast to Marching Cubes described in the lectures. Figure 3 shows a comparison between our results and the result from the assignment, confirming the correctness of isosurface raycasting, in addition, figure 7 shows the correctness of isosurface raycasting with Phong Shading enabled.



(a) Our drawn composite image of the rendered carp

(b) Composite image from the assignment

Figure 3: Iso surface raycasting carp comparison

3.2 Computing gradient

We use the gradient approximation from Levoy [2] as seen in equation 5. We do not calculate these values for the outermost edge voxels, as that prevents out of bound exceptions. These voxels are set to 0. This generally does not affect of the visualisation as these edges generally are empty anyway. This formula was implemented in the `Compute()` function in `GradientVolume.java`.

$$\begin{aligned} &\nabla f(\mathbf{x}_1) = \nabla f(x_i, y_j, z_k) \approx \\ &\tfrac{1}{2}\left\{f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)\right] \\ &\tfrac{1}{2}\left\{f\left(x_i, y_{j+1}, z_k\right) - f\left(x_i, y_{j-1}, z_k\right)\right] \\ &\tfrac{1}{2}\left\{f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})\right]\Big]. \end{aligned} \tag{5}$$

3.2.1 Tri-linear application to gradient

For tri-linear interpolation of gradients the approach is very similar to that of voxels. In tri-linear interpolation, a single final value is interpolated from 8 surrounding voxels. With gradients each of these voxels is a vector of size three. The only difference in these values is that they can be negative, which is not a problem in the formula. We apply the formula three times, separately for the gradient value of the coordinates of x, y, and z. This new coordinate value is returned as a `VoxelGradient` vector. This implementation is done in `getGradientTrilinear`.

3.3 Phong shading

Shading is the use of light reflection to improve the look of images. More than that, it gives both a sense of depth, and, when used on different materials, gives us clearer boundaries and helps differentiate between them. We implement a simplified version of Phong shading based on using a halfway vector as an approximation of the reflection vector. Examples of the implementation can be seen in figures 4, 5, and 7. The halfway vector (equation 6) avoids the expensive vector multiplications and dot products of the reflection vector (equation 7) by using simple vector additions. The usage of the `phongShading` is simplified further by assuming that the lightvector (L) originates in the same point as the Viewvector (V), such that $L = V$. The `phongShading` function does allow for a different light- and viewvector, but the visualization will always call it with $L = V$.

$$H = \frac{L+V}{|L+V|} \tag{6}$$

$$R = (2N \cdot L)N - L \tag{7}$$

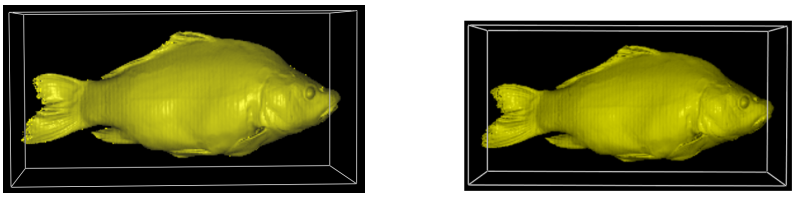
The complete implementation of the implementation is seen in equation 8. This calculation is done separately for the three colors (I_d).

$$I = I_a * k_{ambient} + I_d k_{diff}(L \cdot N) + k_{spec}(N \cdot H)^{\alpha} \tag{8}$$

- I_a = Opacity of voxel.
- I_d = R, G, B value of voxel.
- k_{diff} = spectral diffusion
- $L = V$ = Viewvector
- N = normal vector = $\frac{\nabla(f(x_1)}{|\nabla(f(x_1)|}$
- k_{spec} = spectral diffusion
- α = n = exponent used to approximate highlight.

Which is a small alteration to the simplified model in the lecture slides. We can directly implement shading in Isosurface Raycasting by using $I_a = 1$. For Composite ray functions it is a bit more difficult as we need to shade each step in the ray and combine these values into one final RGB value. We use an adapted version of equation 3, where instead of color we keep track of a phongcolor and instead of a voxelcolor we calculate a phongcolor for every step, this is written down in equation 9. The α_{pix} value is the opacity value of the color of the previous step. For 2D transfer function we also need to update the Phong value every step. For this equation 9 is also used. In each step the the Phong color of the previous step (PhongColorPrevious) is used. In `2DTransferFunction` the α_{pix} value is the alpha value from the previous step, times the opacity based weighting, as explained in the next chapter. [4].

$$PhongColor = PhongColor * \alpha_{pix} + PhongColorPrevious * (1 - \alpha_{pix}) \tag{9}$$



(a) Our drawn composite image of the rendered carp with Phong Shading

(b) Phong shading image from the assignment

Figure 4: Phong shading with Isosurface raycasting

4 2D transfer function: Gradient-based opacity weighting

Gradient based opacity weighing is an example of a separable 2D Transfer function. It uses the voxel value and it's gradient magnitude to create 2 1D functions, which can be combined to create a picture in which it is easier to classify different materials or features [3]. Gradient magnitude gives an idea of how quick changes are happening between different voxels. Such regions with rapid changes are useful to visualise as they may indicate a change in materials [1]. Our approach to combining the voxel values and gradients is presented in [2]. We use the same ray vector as in MIP and compositing ray functions. For each sample in this ray we want to compute an opacity. The intuition behind deciding on an opacity value is that we want to visualise a voxel if it's value $f(x_i)$ is the same as a value we have chosen: f_v in the formula and intensity in the transfer function. In this case we make it fully opaque. When the intensity is close to the voxel value we also want the opacity to be close, that's why we measure the distance between them $|f_v - f(x_1)|$. When gradients change edges become visible. We smoothen these by dividing by the gradient magnitude $|\nabla f(x_1)|$. Finally we also only calculate these opacities for values that are within some radius r, such that we can decide which voxels we want to display. In the implementation this r is scaled by the maximum gradient. If voxels are not in this radius they are set to completely transparent. Similarly to composite ray functions, we calculate the final opacity by iteratively combining the opacity of the previous voxel in the rayvector and the newly retrieved opacity from the transfer function. This approach is described in Levoy [2] (equation 11). In the Implementation this value α_{pix} is calculated iteratively each step of the rayvector.

$$\alpha(x_1) = \alpha_v \begin{cases} 1 & \text{for } |\nabla f(x_1)| = 0 \text{ and } f(x_1) = f_v \\ 1 - \frac{1}{r}\frac{|f_v - f(x_1)|}{|\nabla f(x_1)|} \mid & \text{for } |\nabla f(x_1)| > 0 \text{ and } f(x_1) - r|\nabla f(x_1)| \leq f_v \leq f(x_1) + r|\nabla f(x_1)| \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

$$\alpha_{\text{pix}}(x_1) = 1 - \prod_{n=1}^{N} (1 - \alpha_n(x_1)) \tag{11}$$

5 Cutting plane

The first challenge with the cutting plane assignment is determining the side of which a voxel is in regards to the cutting plane. In order to do so, we created a helper function `leftOfPlane` which uses an altered version of the point-normal formula of a plane in order to return a Boolean that determines whether the voxel is left of the plane or not. In this function, the `planeNorm` and `planePoint` are used to determined the side of the voxel:

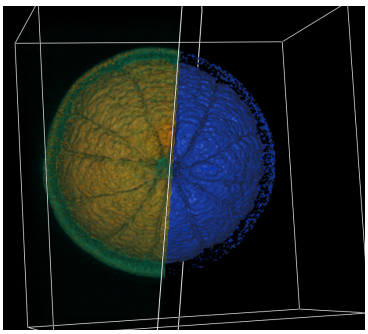
$$a(x-x_c)+b(y-y_c)+c(y-y_c)+d>0. \tag{12}$$

The values a, b, c and x, y, z, are taken from the already implemented variables `planeNorm` and `planePoint` respectively. x_c, y_c and z_c are the coordinates of the point-coordinate to be checked. This allows us to check if a point is left or right of a plane, representing the front and back of the image.
Next, in the raycast function, the cutting plane logic is implemented by calling the `leftOfPlane` helper function on the current pixelcoordinate. Instead of a single switch statement to calculate the color values, two are now used, for the front and back modes respectively. A helper variable `isFront` is created and used which is used in the various modes to determine if we should use the values from the front or back modes (for instance, `isoValueFront` or `isoValueBack`). This helper variable is then passed as an argument in the method functions.

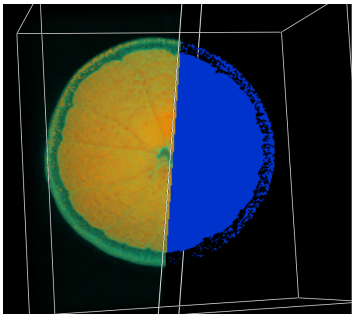
Specific mode implentations:
In the `traceRayIso` function, the helper variable `isFront` is used to use either `isoValueFront` or `isoValueBack`, and to use the rgb values of the `isoColorFront` and `isoColorBack`. Since it already returns a value which is either positioned left or right of the plane, this concludes the implementation of cutting plane for `traceRayIso`.
The modes `COMPOSITING` and `TRANSFER2D` are a bit more complex. They are contained in the `traceRayComposite` function. Here, `isFront` is used to take both the correct voxel values and material values (where required) depending if the voxels are left or right of the plane.
The switch statement for the modes within `traceRayComposite` needed to be updated to contain the truth value of the specific `isFront` value for the voxels that are being calculated. This was done by creating a new `Raycast Mode` instance which is changes depending on the `isFront` variable. Therefore, both the modes in the switch statements can now correctly use the color and material values for the front and back modes respectively, which completes the implementation for cutting plane of each of the rendering modes.

6 Analysis

Comparison of modes: The modes MIP and slicing were pre-implemented, and will therefore not be a major part of this analysis. The current implementation of slicer is useful when taking a look at a single slice of an object. MIP allows for a quick identification of high density objects within objects.
Identifying use-cases and comparison of composite and 2d transfer function modes: In image 7a we can see a comparison between the 2d transfer function and the 1d transfer function (compositing mode). When attempting to find the coins hidden inside the pig dataset, from a practical perspective, the 2d transfer function allowed us to pinpoint the coins a lot easier and quicker. However, the 1d transfer function allows for the use of more colors, making us able to better distinguish the coins. When attempting to distinguish multiple objects within the dataset with varying densities ('values'), the composite function is especially useful by giving them different colors to better distinguish them as seen in figure 7b.
Comparing compositing raycasting to the MIP function, we see in figure 8 it gives more detail and more control over what can be seen. Using a transfer function, different intensity values of voxels on the ray can be given different colors and intensities. Using this, internal objects (such as organs) within other objects can become more clearly visible, whereas MIP only returns the most intense (highest voxel value) value throughout the ray.
Figure 5 clearly illustrates the added value Phong shading gives. It allows us to see depth in an image giving new information which was not seen before. Another image which clearly illustrates the advantages gained is the example from the isosurface mode on the pig: images 6a and 6b. Without Phong shading, the flower pattern could not be distinguished.
Finally, figure 9 showcases how the cutting mode allows two distinctly different modes to work well together. In the tooth example, the isosurface mode allows the viewer to get a clear understanding of the depth and shape of the tooth, where the 2d transfer function gives a solid impression of the internals of the tooth. All in all, different modes allow for different methods of analysis. Each of the modes can find added value for different datasets, and combining techniques (especially using the cutting plane) gives for the most broad analysis.

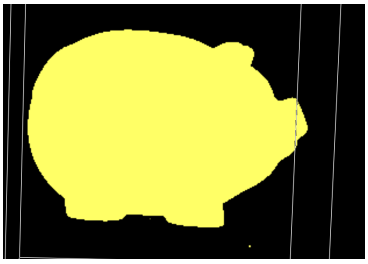


(a) Orange dataset: Phong enabled, composite and 2d transfer function

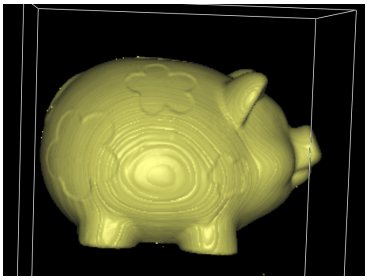


(b) Orange dataset: Phong disabled, composite and 2d transfer function

Figure 5: Showcasing composite and 2d transfer modes with Phong shading enabled and disabled

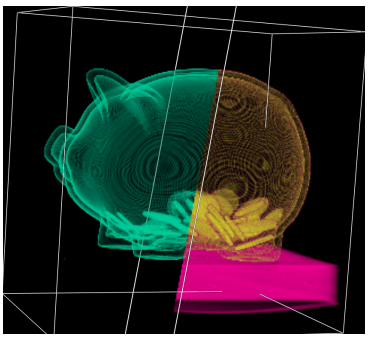


(a) Pig dataset with default isosurface raycasting settings

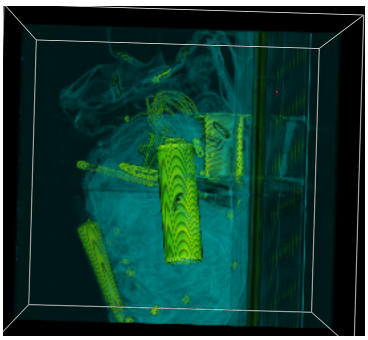


(b) Pig dataset with default isosurface raycasting settings and phong shading enabled

Figure 6: Finding hidden features using phong shading

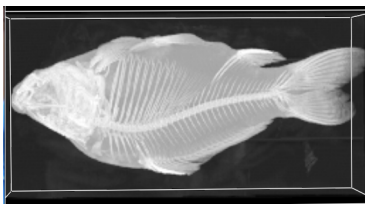


(a) Pig dataset. Left slice: 2d transfer function. Right slice: compositing mode

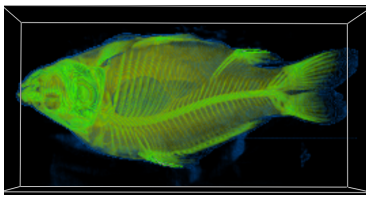


(b) Backpack dataset: use-case for multiple colors in composite function

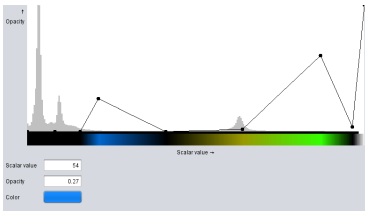
Figure 7: Practical difference in analysis 2d transfer function and compositing (1d transfer function) modes allow



(a) Carp dataset with MIP

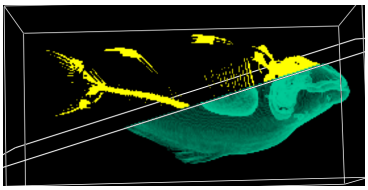


(b) Carp dataset with composite mode

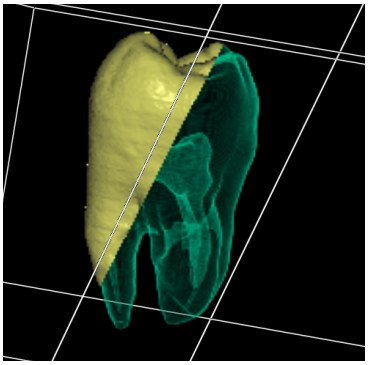


(c) Transfer function for figure 8b

Figure 8: MIP versus composite mode



(a) Carp dataset with isosurface raycasting (ISO 250) and 2d tranfer function



(b) Tooth dataset. Left: isosufrace (ISO 500). Right: 2d transfer function (DEFAULT). Phong shading enabled

Figure 9: Isosurface raycasting versus 2d transfer function: depth and shape versus internals

References

[1] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[2] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[3] Patric Ljung, Jens Krüger, Eduard Gröller, Markus Hadwiger, Charles Hansen, and Anders Ynnerman. State of the art in transfer functions for direct volume rendering. *Computer Graphics Forum*, 35:669–691, 06 2016.

[4] E.B. Lum and kun ling Ma. Lighting transfer functions using gradient aligned sampling. volume 2004, pages 289– 296, 11 2004.

[5] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

```java
/**
 * All edited functions for 2IMV20 assignment 1
 * Made by:
 * Jakob Serlier        1645757
 * Patrick Strookappe   0896169
 * Wouter van Oorschot  0906557
 */

/**
 * Set the IsoValue for the back mode, a small error in the given code,
 * as mentioned by email.
 */
private void IsovalueTextBox1ActionPerformed(java.awt.event.ActionEvent evt) {
    // get the isovalue from IsovalueTextBox1, as this is the back mode isovalue box
    String s = this.IsovalueTextBox1.getText();
    renderer.setIsoValueBack(Float.parseFloat(s));//set value
    renderer.changed();
}


/**
* Determines if the coordinates of a coordinate pixel is on the left or
* right side of the cutting plane.
*
* @param coord Pixel coordinate in 3D space
* @return boolean if coordinate is left of plane (true = left)
*/
private boolean leftOfPlane(double[] coord) {
    // Coordinates to check
    double cx = coord[0], cy = coord[1], cz = coord[2];
    // Point of the plane
    double nx = planePoint[0], ny = planePoint[1], nz = planePoint[2];
    // Normal vector of plane
    double a = planeNorm[0], b = planeNorm[1], c = planeNorm[2];
    //Point-normal plane formula used to determine left or right of plane
    return a*(cx - nx) + b*(cy - ny) + c*(cz - nz) < 0;
}


/**
 * Gets the corresponding voxel using Tri-linear Interpolation.
 *
 * @param coord Pixel coordinate in 3D space of the voxel we want to get.
 * @return The voxel value.
 */
private short getVoxelTrilinear(double[] coord) {
    // TODO 1: Implement Tri-Linear interpolation and use it in your code
    // instead of getVoxel().

    // Initialize dx, dy and dz with coordinate values
    double dx = coord[0], dy = coord[1], dz = coord[2];

    // Verify the coordinates inside the volume
    if (dx < 0 || dx >= volume.getDimX()-1 || dy < 0 || dy >= volume.getDimY()-1
            || dz < 0 || dz >= volume.getDimZ()-1) {

        // If not, just return 0
        return 0;
    }
    //https://en.wikipedia.org/wiki/Trilinear_interpolation
    // get floor and ceil of all coordinate points
    int x_floor = (int) Math.floor(dx);
    int y_floor = (int) Math.floor(dy);
    int z_floor = (int) Math.floor(dz);
    int x_ceil = (int) Math.ceil(dx);
    int y_ceil = (int) Math.ceil(dy);
    int z_ceil = (int) Math.ceil(dz);
    // find integers above and below the coordinate(cubic latice around point)
    short v_000 = volume.getVoxel(x_floor, y_floor, z_floor);
    short v_100 = volume.getVoxel(x_ceil, y_floor, z_floor);
    short v_010 = volume.getVoxel(x_floor, y_ceil, z_floor);
    short v_001 = volume.getVoxel(x_floor, y_floor, z_ceil);
    short v_110 = volume.getVoxel(x_ceil, y_ceil, z_floor);
    short v_101 = volume.getVoxel(x_ceil, y_floor, z_ceil);
    short v_011 = volume.getVoxel(x_floor, y_ceil, z_ceil);
    short v_111 = volume.getVoxel(x_ceil, y_ceil, z_ceil);

    // Distance between coordinate and cubic latice
    double alpha = dx - x_floor;
    double beta = dy - y_floor;
    double gamma = dz - z_floor;

    // return trilinear interpolation of the value, code from the lecture
    // notes.
    return (short) (((1-alpha) * (1-beta) * (1-gamma)*v_000) +
            (alpha * (1-beta) * (1-gamma) * v_100) +
            (beta * (1-alpha) * (1-gamma) * v_010) +
            (alpha * beta * (1-gamma) * v_110) +
            (gamma * (1-alpha) * (1-beta) * v_001) +
            (alpha*(1-beta)*gamma*v_101) +
            ((1-alpha) *beta*gamma*v_011) +
            (alpha*beta*gamma*v_111));

}

/**
 * Gets the corresponding VoxelGradient using Tri-linear interpolation.
 *
 * @param coord Pixel coordinate in 3D space of the voxel we want to get.
 * @return The voxel gradient.
 */
private VoxelGradient getGradientTrilinear(double[] coord) {
    // TODO 6: Implement Tri-linear interpolation for gradients

    // initialize coordinate in dx dy and dz
    double dx = coord[0], dy = coord[1], dz = coord[2];

    // check if coordinate is within volume
    if (dx < 0 || dx > (gradients.getDimX() - 2) || dy < 0 || dy > (gradients.getDimY() - 2)
            || dz < 0 || dz > (gradients.getDimZ() - 2)) {

        // if outside return a zero gradient
        return ZERO_GRADIENT;
    }

    // get floor and ceil of dx dy and dz
    int x_floor = (int) Math.floor(dx);
    int y_floor = (int) Math.floor(dy);
    int z_floor = (int) Math.floor(dz);
    int x_ceil = (int) Math.ceil(dx);
    int y_ceil = (int) Math.ceil(dy);
    int z_ceil = (int) Math.ceil(dz);

    // get gradients above and below the coordinate
    double v_000x = gradients.getGradient(x_floor, y_floor, z_floor).x;
    double v_100x = gradients.getGradient(x_ceil, y_floor, z_floor).x;
    double v_010x = gradients.getGradient(x_floor, y_ceil, z_floor).x;
    double v_001x = gradients.getGradient(x_floor, y_floor, z_ceil).x;
    double v_110x = gradients.getGradient(x_ceil, y_ceil, z_floor).x;
    double v_101x = gradients.getGradient(x_ceil, y_floor, z_ceil).x;
    double v_011x = gradients.getGradient(x_floor, y_ceil, z_ceil).x;
    double v_111x = gradients.getGradient(x_ceil, y_ceil, z_ceil).x;

    double v_000y = gradients.getGradient(x_floor, y_floor, z_floor).y;
    double v_100y = gradients.getGradient(x_ceil, y_floor, z_floor).y;
    double v_010y = gradients.getGradient(x_floor, y_ceil, z_floor).y;
    double v_001y = gradients.getGradient(x_floor, y_floor, z_ceil).y;
    double v_110y = gradients.getGradient(x_ceil, y_ceil, z_floor).y;
    double v_101y = gradients.getGradient(x_ceil, y_floor, z_ceil).y;
    double v_011y = gradients.getGradient(x_floor, y_ceil, z_ceil).y;
    double v_111y = gradients.getGradient(x_ceil, y_ceil, z_ceil).y;

    double v_000z = gradients.getGradient(x_floor, y_floor, z_floor).z;
    double v_100z = gradients.getGradient(x_ceil, y_floor, z_floor).z;
    double v_010z = gradients.getGradient(x_floor, y_ceil, z_floor).z;
    double v_001z = gradients.getGradient(x_floor, y_floor, z_ceil).z;
    double v_110z = gradients.getGradient(x_ceil, y_ceil, z_floor).z;
    double v_101z = gradients.getGradient(x_ceil, y_floor, z_ceil).z;
    double v_011z = gradients.getGradient(x_floor, y_ceil, z_ceil).z;
    double v_111z = gradients.getGradient(x_ceil, y_ceil, z_ceil).z;

    // get distance between coordinate and cubic latice
    double alpha = dx - x_floor;
    double beta = dy - y_floor;
    double gamma = dz - z_floor;

    // calculate the trilinear interpolated gradient for the x, y and z
    // direction.
    double gx = (((1 - alpha) * (1 - beta) * (1 - gamma) * v_000x)
            + (alpha * (1 - beta) * (1 - gamma) * v_100x)
            + (beta * (1 - alpha) * (1 - gamma) * v_010x)
            + (alpha * beta * (1 - gamma) * v_110x)
            + (gamma * (1 - alpha) * (1 - beta) * v_001x)
            + (alpha * (1 - beta) * gamma * v_101x)
            + ((1 - alpha) * beta * gamma * v_011x)
            + (alpha * beta * gamma * v_111x));

    double gy = (((1 - alpha) * (1 - beta) * (1 - gamma) * v_000y)
            + (alpha * (1 - beta) * (1 - gamma) * v_100y)
            + (beta * (1 - alpha) * (1 - gamma) * v_010y)
            + (alpha * beta * (1 - gamma) * v_110y)
            + (gamma * (1 - alpha) * (1 - beta) * v_001y)
            + (alpha * (1 - beta) * gamma * v_101y)
            + ((1 - alpha) * beta * gamma * v_011y)
            + (alpha * beta * gamma * v_111y));

    double gz = (((1 - alpha) * (1 - beta) * (1 - gamma) * v_000z)
            + (alpha * (1 - beta) * (1 - gamma) * v_100z)
            + (beta * (1 - alpha) * (1 - gamma) * v_010z)
            + (alpha * beta * (1 - gamma) * v_110z)
            + (gamma * (1 - alpha) * (1 - beta) * v_001z)
            + (alpha * (1 - beta) * gamma * v_101z)
            + ((1 - alpha) * beta * gamma * v_011z)
            + (alpha * beta * gamma * v_111z));

    // save the gradient in a VoxelGradient object
    VoxelGradient newGrad = new VoxelGradient((float) gx, (float) gy,(float) gz);

    // return the gradien
    return newGrad;
}

/**
 *
 * Updates {@link #image} attribute (result of rendering) using the
 * Isosurface raycasting. It returns the color assigned to a ray/pixel given
 * its starting and ending points, and the direction of the ray.
 *
 * @param entryPoint Starting point of the ray.
 * @param exitPoint Last point of the ray.
 * @param rayVector Direction of the ray.
 * @param sampleStep Sample step of the ray.
 * @return Color assigned to a ray/pixel.
 */
private int traceRayIso(double[] entryPoint, double[] exitPoint, double[] rayVector, double sampleStep, boolean isFront) {
    //implemented traceRayIso by altering the traceRaceMIP algo.

    // cutting plane logic
    // isoColorFront contains the isosurface color from the GUI
    // Initialization of the colors as floating point values
    double r, g, b, isoValue;
    if(isFront){
        isoValue = isoValueFront;
        r = isoColorFront.r;
        g = isoColorFront.g;
        b = isoColorFront.b;
    } else {
        isoValue = isoValueBack;
        r= isoColorBack.r;
        g = isoColorBack.g;
        b = isoColorBack.b;
    }

    // same as in composite
    double[] lightVector = new double[3];
    VectorMath.setVector(lightVector, rayVector[0], rayVector[1], rayVector[2]);

    //initializing alpha, for opacity
    double alpha = 0.0;
    //compute the increment and the number of samples
    double[] increments = new double[3];
    VectorMath.setVector(increments, rayVector[0] * sampleStep, rayVector[1] * sampleStep, rayVector[2] * sampleStep);

    // Compute the number of times we need to sample
    int nrSamples = 1 + (int) Math.floor(VectorMath.distance(entryPoint, exitPoint) / sampleStep);

    //the current position is initialized as the entry point
    double[] currentPos = new double[3];
    VectorMath.setVector(currentPos, entryPoint[0], entryPoint[1], entryPoint[2]);


    do {
        double value = getVoxelTrilinear(currentPos);
        //when it encounters a value larger than the isovalue boundary, set opacity to 100% and break
        if (value > isoValue){
            // set alpha is one, the color is determined by the r g b values
            // set at the beginning of this function
            alpha = 1;
            if(shadingMode){
                // if shading is on, calculate the phongshading according
                // to the color of the isosurface renderer (in the UI)
                TFColor current = new TFColor(r,g,b,alpha);
                TFColor phong = computePhongShading(current, getGradientTrilinear(currentPos), lightVector, rayVector);
                r = phong.r;
                g = phong.g;
                b = phong.b;
                // THIS BREAK IS VERY IMPORTANT
                // break when phong is calculated instead of summing all shading
                // functions of all voxels which exceed the isoValue
                break;
            }
        }
        // update currentpos
        for (int i = 0; i < 3; i++) {
            currentPos[i] += increments[i];
        }
        nrSamples--;
    } while (nrSamples > 0);

    //compute color
    int color = computePackedPixelColor(r, g, b, alpha);
    return color;
}

/**
 *
 * Updates {@link #image} attribute (result of rendering) using the
 * compositing/accumulated raycasting. It returns the color assigned to a
 * ray/pixel given its starting and ending points, and the direction of the
 * ray.
 *
 * Ray must be sampled with a distance defined by sampleStep.
 *
 * @param entryPoint Starting point of the ray.
 * @param exitPoint Last point of the ray.
 * @param rayVector Direction of the ray.
 * @param sampleStep Sample step of the ray.
 * @return Color assigned to a ray/pixel.
 */
private int traceRayComposite(double[] entryPoint, double[] exitPoint, double[] rayVector, double sampleStep, boolean isFront) {
    double[] lightVector = new double[3];

    // the light vector is the same as the viewvector, as the light comes
    // from the viewer in this application
    VectorMath.setVector(lightVector, rayVector[0], rayVector[1], rayVector[2]);

    // initialize r g b alpha
    double r, g, b;
    r = g = b = 0.0;
    double alpha = 0.0;

    TFColor final_color = new TFColor();
    TFColor phong = new TFColor();
    // 1D transfer function
    // do something with samples
    // calculate all samples same as in traceRayMIP()
    double[] increments = new double[3];
    VectorMath.setVector(increments, rayVector[0] * sampleStep, rayVector[1] * sampleStep, rayVector[2] * sampleStep);

    // Compute the number of times we need to sample
    int nrSamples = 1 + (int) Math.floor(VectorMath.distance(entryPoint, exitPoint) / sampleStep);

    //the current position is initialized as the entry point
    double[] currentPos = new double[3];
    VectorMath.setVector(currentPos, entryPoint[0], entryPoint[1], entryPoint[2]);

    RaycastMode mode;
    if(isFront){
        mode = modeFront;
    }
    else{
        mode = modeBack;
    }
    switch (mode) {
    case COMPOSITING:
        // loop through all the samples
        // slide 31
        // final_color = t_i c_i + (1  - t_i )final_color_i-1
        // initialize the final_color for the final_color_i-1
        // initialize tfcolor phong to allow phong shading
        phong.r = final_color.r = r;
        phong.g = final_color.g = g;
        phong.b = final_color.b = b;
        phong.a = final_color.a = alpha;


        do {
            // calculate for every sample
            // get current voxel using new trilinear function
            int value = getVoxelTrilinear(currentPos);
            // get the color associated with the value of this voxel
            TFColor color = isFront? tFuncFront.getColor(value) : tFuncBack.getColor(value);

            if (shadingMode) {
                // calculate phong shading values
                TFColor phongNew = computePhongShading(color, getGradientTrilinear(currentPos), lightVector, rayVector);
                // update the final phong color with the calculated phong color of this voxel
                phong.r = phong.r + phongNew.r * color.a * (1 - final_color.a);
                phong.g = phong.g + phongNew.g * color.a * (1 - final_color.a);
                phong.b = phong.b + phongNew.b * color.a * (1 - final_color.a);
                // the alpha is only changed for the final_color, this is combined at the end of this function
                final_color.a = final_color.a + color.a * (1 - final_color.a);
            }
            else{
                //calculate non shaded composite values
                // make voxel color from TFColor using front-to-back compositing
                final_color.r = final_color.r + color.r * color.a * (1 - final_color.a);
                final_color.g = final_color.g + color.g * color.a * (1 - final_color.a);
                final_color.b = final_color.b + color.b * color.a * (1 - final_color.a);
                //recalculate alpha value iteratively
                final_color.a = final_color.a + color.a * (1 - final_color.a);
            }

            nrSamples--;
            // update position
            for (int i = 0; i < 3; i++) {
                currentPos[i] += increments[i];
            }
        } while (nrSamples > 0);
        // final color is opaque
        final_color.a = 1;
        alpha = 1;
        break;
        case TRANSFER2D:
            // 2D transfer function
            // loop through all the samples
            // slide 31
            // final_color = t_i c_i + (1  - t_i )final_color_i-1
            // initialize the final_color for the final_color_i-1
            final_color.a = 1;
            phong.a = 1;
            do {
                // calculate for every sample
                // get current voxel using new trilinear function
                double voxelValue = getVoxelTrilinear(currentPos);
                double material_value;
                double material_r;
                TFColor color;
                // cutting plane logic for back or front items
                if(isFront){
                    material_value = tFunc2DFront.baseIntensity;
                    material_r = tFunc2DFront.radius;
                    color = tFunc2DFront.color;
                }
                else{
                    material_value = tFunc2DBack.baseIntensity;
                    material_r = tFunc2DBack.radius;
                    color = tFunc2DBack.color;
                }

                double gradMagnitude = getGradientTrilinear(currentPos).mag;
                double tempAlpha = computeOpacity2DTF(material_value,  material_r,
                    voxelValue,  gradMagnitude);
                // do transferfunction with the value of the voxel
                final_color.r = color.r;
                final_color.g = color.g;
                final_color.b = color.b;
                // make voxel color from TFColor
                alpha = tempAlpha*color.a;

                final_color.a = (final_color.a)*(1-alpha);
                if(shadingMode){
                    // get shaded color
                    TFColor phongNew = computePhongShading(final_color, getGradientTrilinear(currentPos), lightVector, rayVector);
                    // Source: https://escholarship.org/content/qt71g7h4kx/qt71g7h4kx_noSplash_79bd87b32ef2ee453600d36ddd930e58.pdf
                    // update phong color with latest shading color
                    phong.r = phongNew.r * final_color.a + (1-final_color.a) * phong.r;
                    phong.g = phongNew.g * final_color.a + (1-final_color.a) * phong.g;
                    phong.b = phongNew.b * final_color.a + (1-final_color.a) * phong.b;
                }

                nrSamples--;
                // update currentpos
                for (int i = 0; i < 3; i++) {
                    currentPos[i] += increments[i];
                }
            } while (nrSamples > 0);

            // calculate final alpha for packed color
            alpha = 1-final_color.a;
            break;
    }

    // colors, or shaded colors to the r g b of this ray
    if (shadingMode) {
        // Shading mode on
        r = phong.r;
        g = phong.g;
        b = phong.b;
    }
    else{
        r = final_color.r;
        g = final_color.g;
        b = final_color.b;
    }

    //computes the color of the ray
    int color = computePackedPixelColor(r, g, b, alpha);
    return color;
}


/**
 * Compute Phong Shading given the voxel color (material color), gradient,
 * light vector and view vector.
 *
 * @param voxel_color Voxel color (material color).
 * @param gradient Gradient voxel.
 * @param lightVector Light vector.
 * @param rayVector View vector.
 * @return Computed color for Phong Shading.
 */
private TFColor computePhongShading(TFColor voxel_color, VoxelGradient gradient, double[] lightVector,
        double[] rayVector) {

    // TODO 7: Implement Phong Shading.
        TFColor color = new TFColor(voxel_color.r, voxel_color.g, voxel_color.b, voxel_color.a);

        //Make sure that maginute of the gradient is non 0 to avoid division by 0.
        // make sure the sure that the voxel is no transparent
    if(gradient.mag != 0.0 && voxel_color.a != 0.0) {

            // in our code the light vector and view vector are always the same
            // however in case it is not, computePhonShading can handle a different light vector
            double[] LVec = lightVector;
            double[] viewVec = rayVector;
            double[] NVec = new double[3]; // normal vector
            double[] HVec = new double[3]; // halway vector

            // Calculate normal vector N, normalize the gradient
            double[] gradientVec = new double[3];
            VectorMath.setVector(gradientVec, gradient.x, gradient.y, gradient.z);
            VectorMath.normalize(gradientVec, NVec);

            // set the sum of the light and view vector
            double[] sum_lv = new double[3];
            VectorMath.setVector(sum_lv,
                    LVec[0] + viewVec[0],
                    LVec[1] + viewVec[1],
                    LVec[2] + viewVec[2]);

            // normalize the sum to get the halway vector
            VectorMath.normalize(sum_lv, HVec);

            // dot product as explained in lecture 2
            double ln = VectorMath.dotproduct(LVec, NVec);
            double nh = VectorMath.dotproduct(NVec,HVec);

            if (ln > 0 && nh > 0) {
                // only apply shading if the dot products are greater than 0
                // parameters given from assignment
                double k_d = 0.7;
                double k_s = 0.2;
                double alphas = 100;
                double i_a = 0.1;

                // Simplyfied phong calculation
                color.r = i_a * voxel_color.a + voxel_color.r * k_d * ln + k_s * Math.pow(nh, alphas);
                color.g = i_a * voxel_color.a +  voxel_color.g * k_d * ln + k_s * Math.pow(nh, alphas);
                color.b = i_a * voxel_color.a +  voxel_color.b* k_d * ln  + k_s * Math.pow(nh, alphas);
            }
    }
    return color;
}

/**
 * Implements the basic tracing of rays through the image given the camera
 * transformation. It calls the functions depending on the raycasting mode.
 *
 * @param viewMatrix
 */
void raycast(double[] viewMatrix) {
    //data allocation
    double[] viewVec = new double[3];
    double[] uVec = new double[3];
    double[] vVec = new double[3];
    double[] pixelCoord = new double[3];
    double[] entryPoint = new double[3];
    double[] exitPoint = new double[3];

    // TODO 5: Limited modification is needed
    int unit = 1;
    if(interactiveMode){
        // when interactive mode is on, i.e. the object is being moved
        // lower calculation resolution for both increment in pixels in the
        // image and amount of steps.
        unit = 4;
    }
    // increment in the pixel domain in pixel units
    int increment = unit;
    // sample step in voxel units
    int sampleStep = unit;


    // reset the image to black
    resetImage();

    // vector uVec and vVec define a plane through the origin,
    // perpendicular to the view vector viewVec which is going from the view point towards the object
    // uVec contains the up vector of the camera in world coordinates (image vertical)
    // vVec contains the horizontal vector in world coordinates (image horizontal)
    VectorMath.setVector(viewVec, viewMatrix[2], viewMatrix[6], viewMatrix[10]);
    VectorMath.setVector(uVec, viewMatrix[0], viewMatrix[4], viewMatrix[8]);
    VectorMath.setVector(vVec, viewMatrix[1], viewMatrix[5], viewMatrix[9]);

    // We get the size of the image/texture we will be puting the result of the
    // volume rendering operation.
    int imageW = image.getWidth();
    int imageH = image.getHeight();

    int[] imageCenter = new int[2];
    // Center of the image/texture
    imageCenter[0] = imageW / 2;
    imageCenter[1] = imageH / 2;

    //The rayVector is pointing towards the scene
    double[] rayVector = new double[3];
    rayVector[0] = -viewVec[0];
    rayVector[1] = -viewVec[1];
    rayVector[2] = -viewVec[2];

    // compute the volume center
    double[] volumeCenter = new double[3];
    VectorMath.setVector(volumeCenter, volume.getDimX() / 2, volume.getDimY() / 2, volume.getDimZ() / 2);

    // ray computation for each pixel
    for (int j = imageCenter[1] - imageH / 2; j < imageCenter[1] + imageH / 2; j += increment) {
        for (int i = imageCenter[0] - imageW / 2; i < imageCenter[0] + imageW / 2; i += increment) {
            // compute starting points of rays in a plane shifted backwards to a position behind the data set
            computePixelCoordinatesBehindFloat(pixelCoord, viewVec, uVec, vVec, i, j);
            // compute the entry and exit point of the ray
            computeEntryAndExit(pixelCoord, rayVector, entryPoint, exitPoint);

            // TODO 9: Implement logic for cutting plane.
            // TODO: implement isfront in other modes to change variable from front/back values.
            // TODO: fix index out of bounds
            if ((entryPoint[0] > -1.0) && (exitPoint[0] > -1.0)) {
                int val = 0;
                //helper variable to determine if we should use front or back values of variables
                boolean isFront = true;
                // if left of plane, or if cutting plane is off (default)
                // use the modeFront calculation
                if (leftOfPlane(pixelCoord) || !cuttingPlaneMode) {
                    switch (modeFront) {
                        case COMPOSITING:
                            //obsolete
                            //val = traceRayComposite(entryPoint, exitPoint, rayVector, sampleStep);
                        case TRANSFER2D:
                            val = traceRayComposite(entryPoint, exitPoint, rayVector, sampleStep, isFront);
                            break;
                        case MIP:
                            val = traceRayMIP(entryPoint, exitPoint, rayVector, sampleStep);
                            break;
                        case ISO_SURFACE:
                            val = traceRayIso(entryPoint, exitPoint, rayVector, sampleStep, isFront);
                            break;
                    }
                } else {
                    // set boolean isFront to false to pass on to traceRay functions
                    isFront = false;
                    switch (modeBack) {
                        case COMPOSITING:
                            //obsolete
                            //val = traceRayComposite(entryPoint, exitPoint, rayVector, sampleStep);
                        case TRANSFER2D:
                            val = traceRayComposite(entryPoint, exitPoint, rayVector, sampleStep, isFront);
                            break;
                        case MIP:
                            val = traceRayMIP(entryPoint, exitPoint, rayVector, sampleStep);
                            break;
                        case ISO_SURFACE:
                            val = traceRayIso(entryPoint, exitPoint, rayVector, sampleStep, isFront);
                            break;
                    }
                }
                for (int ii = i; ii < i + increment; ii++) {
                    for (int jj = j; jj < j + increment; jj++) {
                        image.setRGB(ii, jj, val);
                    }
                }
            }

        }
    }
}

/**
 * Computes the opacity based on the value of the pixel and values of the
 * triangle widget. {@link #tFunc2DFront} contains the values of the base
 * intensity and radius. {@link TransferFunction2D#baseIntensity} and
 * {@link TransferFunction2D#radius} are in image intensity units.
 *
 * @param material_value Value of the material.
 * @param material_r Radius of the material.
 * @param voxelValue Voxel value.
 * @param gradMagnitude Gradient magnitude.
 * @return
 */
public double computeOpacity2DTF(double material_value, double material_r,
        double voxelValue, double gradMagnitude) {

    // TODO 8: Implement weight based opacity.
    // normalize radius as told in announcement
    double radius = material_r / gradients.getMaxGradientMagnitude();
    double opacity;

    if (gradMagnitude == 0 & voxelValue == material_value) {
        // if gradient magnitude is zero and the voxel value is the same as
        // the material value, opacity is 1. This is the 'bottom' point of
        // the triangle in the 2DTF UI
        opacity = 1;
    }
    else if (gradMagnitude > 0 &
            voxelValue - (radius * gradMagnitude) <= material_value &
            material_value <= voxelValue + (radius * gradMagnitude)) {
        // if the value somewhere within the triangle, calculate the opacity
        // based on the location in the triangle
        opacity = 1 - Math.abs((1/radius)* ((material_value - voxelValue)/ gradMagnitude));
    }
    else {
        // if not on, or inside triangle, thus outside triangle, set opacity
        // zero, black.
        opacity = 0;
    }
    return opacity;
}

/**
 * Computes the gradient information of the volume according to Levoy's
 * paper.
 */
private void compute() {
    // TODO 4: Implement gradient computation.
    // calculate gradient on everything but the edges of the volume
    for(int x = 1; x < dimX-1; x++){
        for(int y = 1; y < dimY-1; y++){
            for(int z = 1; z < dimZ-1; z++){
                float gradX = 0.5F*(volume.getVoxel(x+1, y, z) - volume.getVoxel(x-1, y, z));
                float gradY = 0.5F*(volume.getVoxel(x, y+1, z) - volume.getVoxel(x, y-1, z));
                float gradZ = 0.5F*(volume.getVoxel(x, y, z+1) - volume.getVoxel(x, y, z-1));
                // save gradient in a VoxelGradient object
                VoxelGradient tmp = new VoxelGradient(gradX, gradY, gradZ);
                // set data[i] with x y z
                setGradient(x,y,z, tmp);
            }
        }
    }
    for(int i = 0; i <data.length; i++){
        if(data[i] == null){
            // set the edge cases to zero, as the gradient is zero on the
            // edges
            data[i] = zero;
        }
    }
}
```