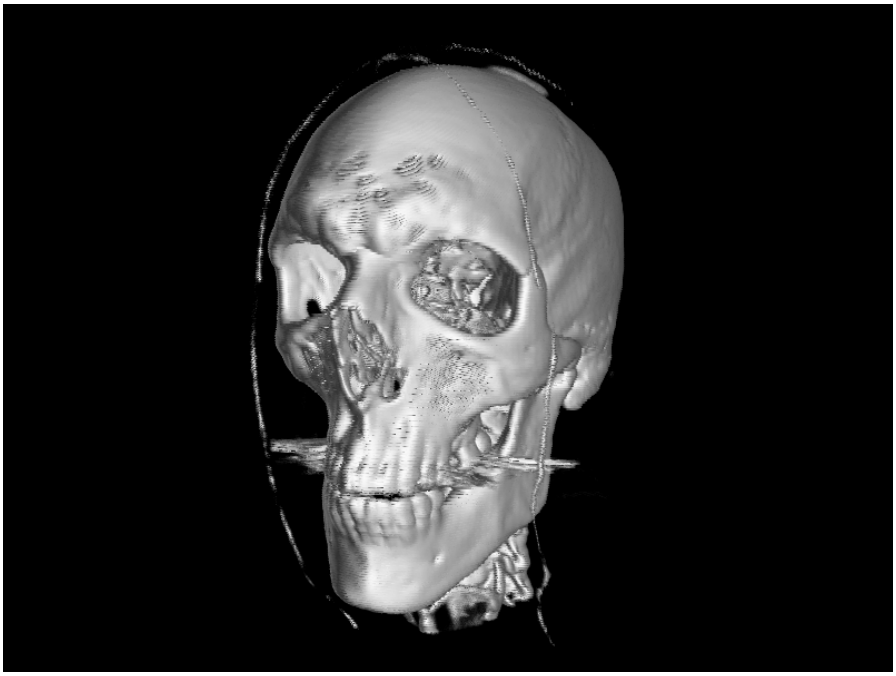


Figure 1: Example for Volume Raycasting. Visualization of a human skull from volumetric data.

CG/task2 — Volume Raycasting

Volume datasets contain information about the inside of the objects or volumes which they represent (see Fig. 1). In the simplest case, the dataset is a partition of the object via a regular grid in volume-elements, so called *voxels*. Every voxel has an assigned scalar value.

Volume rendering is then a technique to visualize volume datasets, which we can do in various different ways. The most well known is a technique called *ray casting* and is based on the idea of generating an image by sending rays from an observer (the virtual camera) through pixels in an image plane and then through the volume dataset. Depending on the setting and lighting model, the hit parts of the volume dataset contribute different color values to the final color of the pixel.

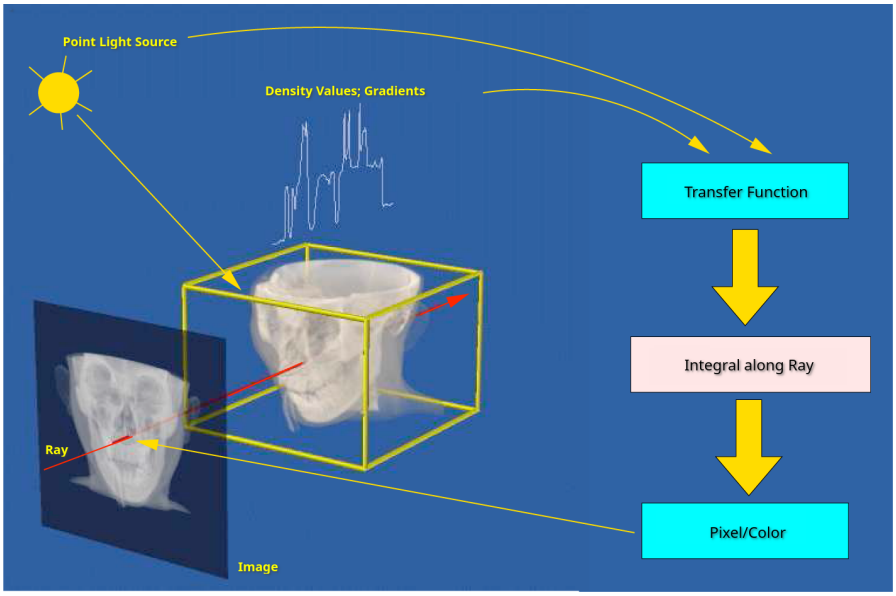


Figure 2: Schematic depiction of the raycasting-technique for a single pixel. A ray is sent from the observer through the volume. We then calculate color and transparency values from the sampled density and gradient values along the ray. The integration (accumulation) of those leads to the final color value of the pixel.

1 Basics

Volume data sets can be rendered very efficiently when one disregards physical laws such as lighting, shadow casting, refraction, reflection, and even scattering, and simply assumes that it is self-illuminating. Additionally, taking into account that the volume darkens further layers behind it, one arrives at the following volume rendering integral:

$$\mathbf{I} = \int_{t_n}^{t_f} T(t)\mathbf{c}(s(\mathbf{x}(t)))\,dt, \tag{1}$$

$$\text{where } T(t) = \exp\left(-\int_{t_n}^{t} \tau\left(s\left(\mathbf{x}(t')\right)\right)dt'\right). \tag{2}$$

$\mathbf{I}$ describes either the intensity or the color value perceived by the observer. The observer ray $\mathbf{x}(t)$ is parameterized by the distance from the observer t. $s(\mathbf{x})$ represents the density values in the dataset along $\mathbf{x}$. Through a so-called transfer function, each density value is assigned a color value $\mathbf{c}(s)$ and an absorption coefficient $\tau(s)$.

In other words, Eqn. (1) states that from every point in the volume, a color value $\mathbf{c}(s(\mathbf{x}(t)))$

is emitted, and this is dimmed by the integrated absorption coefficients $\tau(s(\mathbf{x}(t')))$ (from the observer's standpoint to the currently considered point in the volume). In the simplest case, $\mathbf{c}(s)$ depends only on s. The choice of the transfer function is critical because it determines how the individual layers are represented.

This method is often extended by a local lighting system, as it then becomes much easier for humans to recognize the structures of objects. To do this, a point light source is simply placed arbitrarily in the scene, and we define that it illuminates every point in the volume equally.

The volume rendering integral from Eqn. (1) is solved iteratively in practical raycasting implementations. The dataset is sampled at discrete locations along the ray. For each sample point, the color $\mathbf{c}_{act}$ and opacity α_{act} are computed. Mapping density values to color and opacity is done through the previously mentioned transfer function, referred to as *classification*.

Although it might seem easier to traverse the ray from back to front, as this allows for dimming the already accumulated light by the opacity of the current sample and adding the current emitted light, practical implementations usually choose the opposite approach. The ray is traversed from front to back, because this technique allows for terminating ray tracing when reaching a point where the remaining sample points can no longer influence the result significantly due to being heavily occluded. In our implementation, we keep track of the already accumulated opacity, resulting in the following relationships for the accumulated light/color $\mathbf{c}_i$ and opacity α_i:

$$\mathbf{c}_i = \mathbf{c}_{i-1} + (1-\alpha_{i-1})\alpha_{act}\mathbf{c}_{act}, \tag{3}$$

$$\alpha_i = \alpha_{i-1} + (1-\alpha_{i-1})\alpha_{act}. \tag{4}$$

Fig. 2 depicts the workings of the raycasting algorithm.

1.1 Raycasting

A ray is mathematically defined as

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d} \tag{5}$$

where $\mathbf{p}$ denotes the coordinates of the *ray origin* and $\mathbf{d}$ is the *ray direction*. The *ray parameter* $t \in \mathbb{R}_+$ is the distance along the ray, starting from the ray origin $\mathbf{p}$ towards the ray direction $\mathbf{d}$.

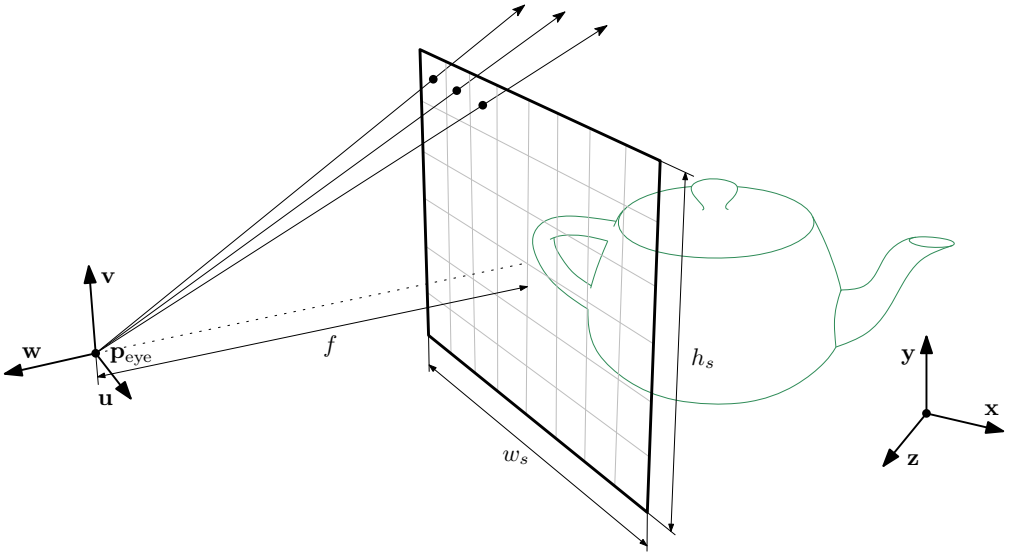


Figure 3: Primary rays are traced from the camera position towards the image plane. The rotation and position of the camera and image plane is given via the locally defined coordinate system $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{p}_{\text{eye}}$ as well as the distance to the image plane f. These axes (and their origin) can vary significantly from the scene coordinate system $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

The core of a ray tracing system is the generation of *primary rays* (also sometimes referred to as *camera rays* or *eye rays*), which are traced through the image plane. In order to generate a final raster image of the scene with resolution $r_x \times r_y$, we *sample* the scene through the image plane (see Fig. 3).

As can be seen in Fig. 4, the basis vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ as well as the position of the camera $\mathbf{p}_{eye}$ define the orientation and location of the camera in the scene. The image plane is positioned at an offset of f in the direction of $-\mathbf{w}$ in front of the camera position. The sampled region of the image plane is defined via its width w_s and height h_s. Each pixel in the output image corresponds to one sample in the image plane, which is determined by tracing a ray through it. Note that the pixels in an image are typically indexed row-wise, starting top left. This means that the pixel $(0,0)$ is located on the top left of the image, while the pixel $(r_x - 1, r_y - 1)$ is located on the bottom right. The sample positions on the image plane result from dividing the image plane into a regular grid consisting of $r_x \times r_y$ cells. The rays are then cast through the *center* of each cell/pixel.

1.1.1 Camera Parameters

The basis vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ are usually calculated via the eye point $\mathbf{p}_{eye}$, a `lookat` point $\mathbf{p}_{lookat}$ and the up vector $\mathbf{v}_{up}$. In our case we are already given the basis vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$.

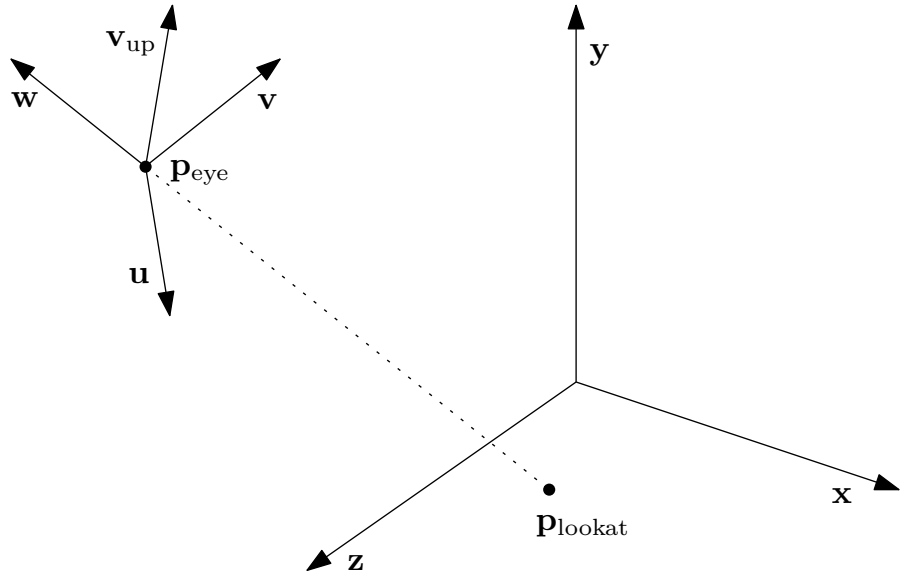


Figure 4: Orientation and position of a camera based on $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{p}_{\text{eye}}$

The distance to the image plane f corresponds to the focal length of the virtual camera. The dimensions of the image plane are usually defined via the field of view β and the aspect ratio a of the output image:

$$a = \frac{w_s}{h_s} \qquad\qquad h_s = 2 \cdot f \cdot \tan\left(\frac{\beta}{2}\right) \tag{6}$$

As we wish for uniform sampling from the image plane:

$$a = \frac{w_s}{h_s} = \frac{r_x}{r_y}. \tag{7}$$

1.2 Trilinear Interpolation

For good image quality, the data, which is presented as a voxel grid, must be interpolated while sampling along the ray. The simplest method is Nearest Neighbor Sampling, which takes the value of the nearest voxel. Significantly better results are achieved with higher-order interpolation schemes, such as *trilinear interpolation*, depicted in Fig. 5. Linear interpolation must be performed along each coordinate axis for this. Although it is conceivable to further increase the order of interpolation, this is uncommon in volume rendering, because the additional computational overhead usually results in only a marginal improvement in image quality.

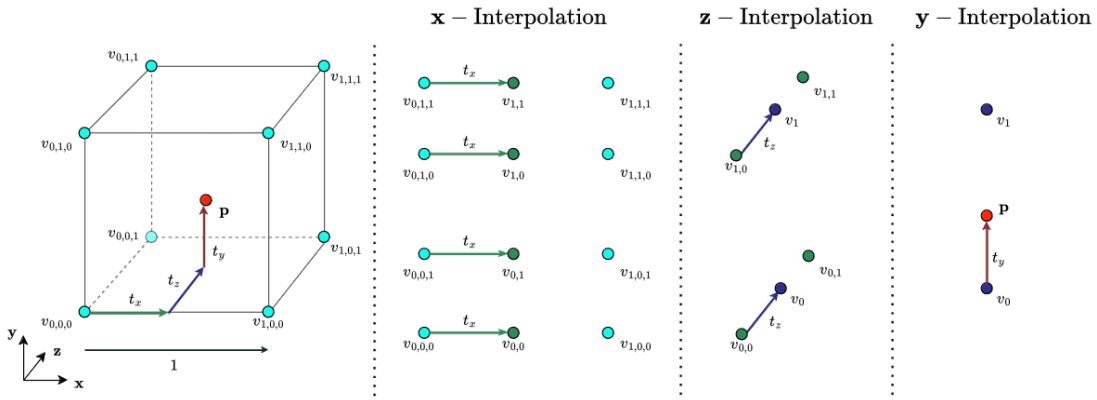


Figure 5: Sketch for trilinear interpolation in a uniform voxel grid. In our case, $v_{i,j,k}$ correspond to the surrounding 8 voxels. To get the value for $\mathbf{p}$, we first compute t_x, t_y, t_z. Next, we use these values to perform linear interpolation between the voxels. In this example, we first linearly interpolate along $\mathbf{x}$, then along $\mathbf{z}$ and finally along $\mathbf{y}$ — note that the order of axes is arbitrary.

To get an interpolated value $s(\mathbf{p})$ for a position $\mathbf{p}$ inside the voxel grid, we first determine the 8 surrounding voxels. Assuming a uniform voxel grid and the distance between the voxels is 1, the minimum corner point $\mathbf{p}_{\text{min}}$ is simply

$$\mathbf{p}_{\text{min}} = \lfloor \mathbf{p} \rfloor , \tag{8}$$

where $\lfloor \cdot \rfloor$ denotes the floor operator. The interpolation values $\mathbf{t} = [t_x, t_y, t_z]$ are now

$$\mathbf{t} = \mathbf{p} - \mathbf{p}_{\text{min}}. \tag{9}$$

Note that $\mathbf{t} \in [0,1]^3$. Next, we can perform linear interpolation along the canonical axes $\mathbf{x}, \mathbf{y}, \mathbf{z}$, as outlined in Fig. 5. Note that the order of the three linear interpolations may be chosen arbitrarily.

1.3 Classification

The mapping of density values to color and opacity occurs through the transfer function and is referred to as *classification*.

The time at which the transfer function is evaluated can vary:

- Pre-Classification: The available volume data is first classified and then interpolated.
- Post-Classification: The available volume data is first interpolated and then classified.

Although Pre-Classification is faster than Post-Classification, as the classification step can be performed as a pre-processing step, Post-Classification is usually preferred because the result is more accurate (see Fig. 6).

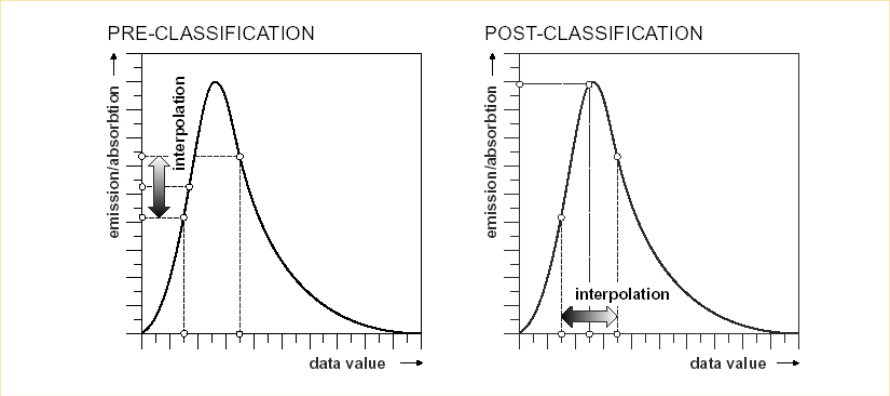


Figure 6: Pre-Classification interpolates the display values (y-axis) directly. Post-Classification first interpolates the data values (x-axis) and then looks into the transfer function to achieve a more accurate result.

For faster rendering, a quick evaluation of the transfer function is important. Therefore, the results of the transfer function are pre-computed for every possible density value and stored in a lookup table.

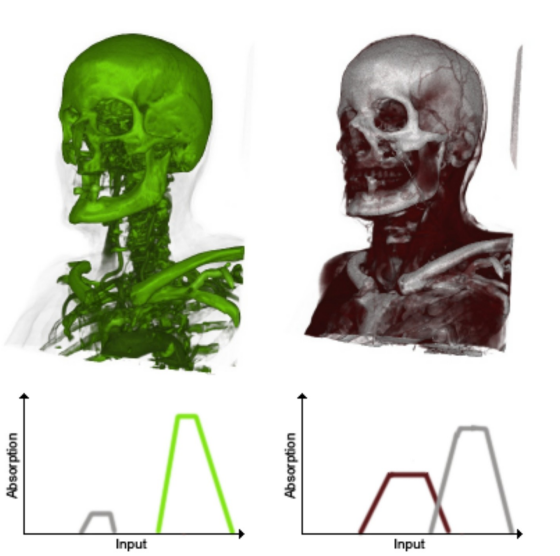
1.3.1 Transfer

The transfer function is the tool with which one can determine the representation of the volume dataset. The effect of two different transfer functions on the visualization of a dataset can be seen in Fig. 7a.

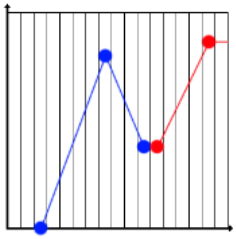
The specification of the transfer function allows for a large number of degrees of freedom. Any possible value in the volume dataset could be assigned any absorption coefficient as well as any color. It is sensible to assign similar values to a certain range. Thus, we specify individual control points between which linear interpolation is performed (see Fig. 7b).

1.4 Sampling Interval

An important parameter for rendering performance is the step size used during sampling. The larger the step size, the faster rendering can be completed. However, the quality of the rendered image also suffers if the step size is chosen too large.



(a) Two different transfer functions applied on the same dataset.



(b) Definition of a transfer function with control points, between which linear interpolation is performed.

Figure 7: Overview for transfer functions.

Caution is also necessary when the dimensions of the volume do not correspond to the number of voxels along all directions. For a physically accurate result, ray sampling must always occur in undistorted space.

1.5 Early Ray Termination

As mentioned earlier, we perform front-to-back rendering to enable early ray termination. When the accumulated opacity α_i reaches a specified threshold α_{thresh}: $\alpha_i \geq \alpha_{\text{thresh}}$, sampling for this ray can be terminated, as it is assumed that the remaining sample points will not bring significant changes to the color value anymore. Common values for α_{thresh} are typically in the range of 0.9.

It should be noted that despite the termination, the range $[0, \alpha_{\text{thresh}}]$ should be mapped to $[0,1]$ — otherwise, too much background color would be blended into the final result.

1.6 Local Shading Model

A local lighting model is the Phong model:

$$I = \min\left(k_{\text{amb}} + k_{\text{dif}}\langle \mathbf{n}, \mathbf{l}\rangle + k_{\text{spec}}\langle \mathbf{r}, \mathbf{v}\rangle^m, 1\right), \tag{10}$$

where $I \in [0,1]$ represents the resulting intensity, and k_{amb}, k_{dif}, k_{spec}, and m (the *shininess* coefficient) correspond to the associated material constants. Also important are the normal vector $\mathbf{n}$, light vector $\mathbf{l}$, reflection vector $\mathbf{r}$, and the view vector $\mathbf{v}$, as seen in Fig. 8.

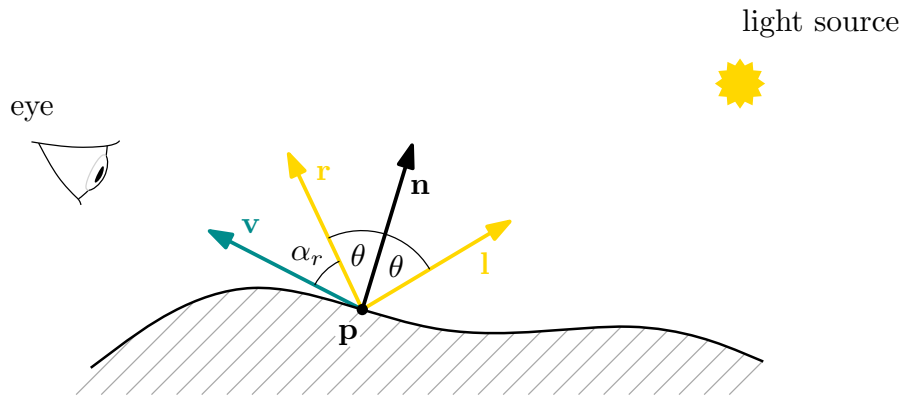


Figure 8: Phong Shading Model.

The normal vector $\mathbf{n}$ can be computed from the volume dataset. It corresponds to the gradient at the current sample point. One approach for computing said gradient is to approximate the partial derivatives using central differences:

$$\nabla s(x_i, y_j, z_k) \approx \begin{pmatrix} \frac{s(x_{i+1}, y_j, z_k) - s(x_{i-1}, y_j, z_k)}{2} \\ \frac{s(x_i, y_{j+1}, z_k) - s(x_i, y_{j-1}, z_k)}{2} \\ \frac{s(x_i, y_j, z_{k+1}) - s(x_i, y_j, z_{k-1})}{2}. \end{pmatrix} \tag{11}$$

The coordinate shifts by one unit correspond to a voxel step. If the extent of the volume is not proportional to the number of voxels along each coordinate direction, the resulting normal vector must be scaled accordingly to be transformed into object space.

With the position of the light source and the camera, as well as the current sample point, $\mathbf{v}$ and $\mathbf{l}$ can be computed (in our case, the light position is equal to the camera position). The view vector $\mathbf{v}$ is identical to the direction vector of the ray (make sure the direction is *towards* the viewer). The light vector $\mathbf{l}$ is calculated as a vector from the sample position to the light position and must then be normalized. To compute the reflection vector $\mathbf{r}$, the following formula is used:

$$\mathbf{r} = 2\mathbf{n}\langle \mathbf{n}, \mathbf{l} \rangle - \mathbf{l}. \tag{12}$$

In the dot product $\langle \mathbf{n}, \mathbf{v} \rangle$, it is important to note that the direction of the derivative ∇s is not known in advance. Since it is not meaningful for volume datasets to define backfaces

that should not be illuminated, if the normal vector points away from the camera, it must be flipped. It is important to ensure that $\mathbf{n}$, $\mathbf{l}$, $\mathbf{r}$, and $\mathbf{v}$ are always unit vectors.

The new color value $\mathbf{c}_i$, considering the lighting system, is then calculated as follows:

$$\mathbf{c}_i = \mathbf{c}_{i-1} + (1 - \alpha_{i-1})\alpha_{act}\mathbf{c}_{act} \cdot I \cdot \mathbf{c}_L, \tag{13}$$

where $\mathbf{c}_L$ represents the color of the light source. In contrast to ray tracing, it is important to note that neither a shadow ray is sent to the light source, which would result in darkening of the diffuse component, nor is the reflected ray traced (no consideration of the color value of the reflected ray).

2 Tasks

In this exercise, the Volume Raycasting Algorithm described in this document is to be implemented. All functions to be implemented can be found in the files «`task2.h/task2.cpp`». The framework already handles the following functions:

- Reading of all necessary data
- Graphical user interface
- Placement of the volume in space
- Creation of the axis-aligned bounding box (AABB)
- Camera

2.1 Creating the Transfer Function Lookup Table (4 points)

Implement the function

```
void createTransferfunctionLookupTable(
    std::vector<float4>& lut,
    const std::vector<ControlPointSet>& controlpoint_sets
)
```

which is intended to fill the lookup table. The function is already called by the framework, so you do not need to call the function yourself. The lookup table `lut` already has the correct size. Each entry represents a density value from 0 to the maximum density value. An entry itself consists of 4 floats, which correspond to the RGB colors and the opacity (alpha).

A control point set consists of an RGB color and a series of control points which are stored in a map. The map key corresponds to the density value, and the map value corresponds to the opacity. Since the control points are stored in a map, they are already sorted by their density value. Each control point set consists of at least 2 control points, which correspond to the minimum and maximum density values.

Your task is to combine the RGBA values from all control point sets to an RGBA value for each entry in the lookup table and store it. For the combination of the RGBA values, you should use the function provided by the framework:

```
float4 TransferFunction::mixColors(
    const float3& rgb1,
    const float3& rgb2,
    float alpha1,
    float alpha2
)
```

The RGBA values of a control point set should be determined by linear interpolation between 2 control points (except if a control point lies exactly at the sought density value, then the sought opacity corresponds exactly to the opacity of the control point). Note that the RGB value is the same for each density value, and only the opacity needs to be interpolated.

2.2 Transfer Function Lookup (2 points)

Implement the function

```
float4 lookupTransferFunction(
    const float density,
    const float opacity_correction,
    const std::vector<float4>& lut
)
```

which returns a linearly interpolated RGBA value from the lookup table based on a density value. The parameter `opacity_correction` is solely used for opacity scaling — multiply the computed α-value with this constant.

2.3 Ray-AABB Intersection (3 points)

Implement the function

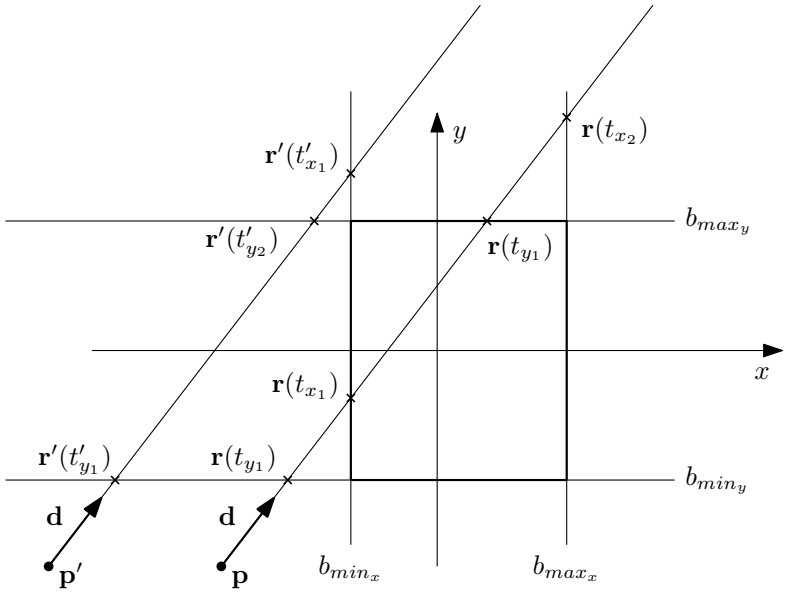


Figure 9: Intersection of a ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d}$, a parallel ray $\mathbf{r}'(t) = \mathbf{p}' + t \cdot \mathbf{d}$, and an AABB defined by the minimum and maximum corner points $\mathbf{b}_{min}$ and $\mathbf{b}_{max}$.

```
bool intersectRayWithAABB(
    const float3& ray_start,
    const float3& ray_dir,
    const float3& aabb_min,
    const float3& aabb_max,
    float& t_near,
    float& t_far
)
```

which returns `true` in case of an intersection with the AABB. In this case, the parameters `t_near` and `t_far` should be set to the distance to the entry and exit points of the ray. This allows for easily determining the entry and exit points $\mathbf{p}_{\text{in}}$, $\mathbf{p}_{\text{out}}$ of the ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d}$:

$$\begin{aligned} \mathbf{p}_{\text{in}} &= \mathbf{p} + t_{\text{near}} \cdot \mathbf{d}, \\ \mathbf{p}_{\text{out}} &= \mathbf{p} + t_{\text{far}} \cdot \mathbf{d}. \end{aligned} \tag{14}$$

Since the bounding box is aligned with the principal axes, the intersection points with a ray can be relatively easily determined via the 6 faces of the box. For each spatial dimension, there is a near and far intersection point along the ray. As shown in Fig. 9, the ray intersects the box if the maximum of all near distances is greater than or equal to the minimum of all far distances.

For a bounding box defined by its minimum corner point $\mathbf{b}_{min}$ and maximum corner point $\mathbf{b}_{max}$, and a ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d}$, the ray parameters t_{x_1} and t_{x_2} for the 2 intersection points along the X-axis can be calculated as follows:

$$t_{x_1} = \frac{b_{min_x} - p_x}{d_x}, \qquad t_{x_2} = \frac{b_{max_x} - p_x}{d_x}. \tag{15}$$

Similarly, the same calculations can be performed for the remaining axes Y and Z. Subsequently, the near and far distances are calculated across all 3 axes by finding the maximum of the near distances and the minimum of the far distances:

$$t_{near} = \max\left\{\min(t_{z_1}, t_{z_2}),\ \min(t_{y_1}, t_{y_2}),\ \min(t_{x_1}, t_{x_2})\right\} \tag{16}$$

$$t_{far} = \min\left\{\max(t_{z_1}, t_{z_2}),\ \max(t_{y_1}, t_{y_2}),\ \max(t_{x_1}, t_{x_2})\right\}. \tag{17}$$

The ray intersects the box with the ray parameters t_{near} and t_{far} if $t_{near} \leq t_{far}$.

2.4 Interpolation of density values (2 points)

Implement the function

```
float interpolatedDensity(
    const Volume& volume,
    const float3& sample_position
)
```

which returns a trilinearly interpolated density value based on a sample position. To perform trilinear interpolation, the 8 surrounding voxels must first be determined.

Assuming the voxel grid is an $11 \times 11 \times 11$ array, then the voxel at position (0,0,0) in the 3D array in the world coordinate system lies exactly at the minimum corner point of the AABB, and the voxel at position (10,10,10) lies at the maximum corner point of the AABB.

To determine the sample position in the uniform voxel grid, it is transformed into the coordinate system of the uniform grid. First, the origin of the coordinate system is shifted to the minimum bounding box corner point, as the indexing of voxels is done via positive integers including zero. Then the sample position can be transformed into the

coordinate system of the uniform voxel grid based on the scaling of the voxel grid. Now it is straightforward to determine the 8 surrounding voxels since the indices correspond to the nearest integers.

Further information regarding trilinear interpolation can be found in Sec. 1.2.

2.5 Volume Raycasting (1 points)

Nowadays, it is common to implement the ray casting method for the GPU as it can be highly parallelized. For simplification, this part of the exercise is to be implemented on the CPU.

Implement the function

```
void volumeRaycasting(
    std::vector<float3> &ray_directions,
    const int left,
    const int top,
    const int right,
    const int bottom,
    const Camera& camera);
```

which generates a ray for the center of each pixel in the output image region defined by `left` and `top` (both inclusive) as well as `right` and `bottom` (both exclusive).

It is important that your program precisely adheres to these specifications. For performance reasons, the framework may distribute the calculation across multiple processor cores. In this case, the function `volumeRaycasting()` will be called concurrently multiple times to process different parts of the image in parallel.

Store the ray directions in the provided vector, in row-major order.

2.6 Pixel Coloring (4 points)

After generating rays to shoot through the image plane, we need to color the pixels according to the intersections of the rays with the volume.

Implement the function

```
uchar4 colorPixel(
    const float3& camera_position,
    const float3& ray_direction,
    const Volume& volume,
    const float step_size,
    const float opacity_correction,
    const float opacity_threshold,
    const float3& background_color,
    const float3& light_color,
    const float k_ambient, const float k_diffuse,
    const float k_specular, const float shininess,
    const bool shading,
    const std::vector<float4>& tf_lut);
```

which takes a ray direction and intersects it with the AABB. If there is no intersection, the pixel color should be equal to the background color. Otherwise, perform front-to-back sampling including Post-Classification within the bounding box along the ray. The distances between the samples are specified by the provided step size. For sampling the density value of the volume dataset, you should use the previously implemented trilinear interpolation function. A lookup in the transfer function will then provide you with the values $\mathbf{c}_{act}$ and α_{act}.

Additionally, early ray termination should be implemented. Once front-to-back sampling for a ray is completed (ray leaves AABB or early ray termination), blend the accumulated color $\mathbf{c}_i$ with the background color based on α_i and α_{thresh}. Convert the resulting output color to discretized colors (`uchar`, thus $\in \{0,\ldots,255\}$) and return the result.

Ensure that the lighting model can be deactivated. If the parameter `shading` is set to `false`, pixel colors should be calculated without the lighting model.

2.7 Bonus: Shading (3 points)

Implement the function

```
float3 calcGradient(
   const Volume& volume,
   const float3& sample_position
)
```

which determines the gradient at the current sample position. Further information on calculating the gradient can be found in Sec. 1.6.

Adapt the color value calculation in the function «`colorPixel()`». The Phong model described in Sec. 1.6 should be implemented as the lighting model. In our case, the position of the light source is always the current position of the camera. Calculate the normal vector **n** using the «`calcGradient()`» function. If the gradient is a zero vector, this indicates a homogeneous density environment, i.e., the voxel neighborhood has the same density values, and normalizing the gradient would result in division by 0. In this case, the view vector **v** should be used as the surface normal.

Ensure that the lighting model can be deactivated; see Sec. 2.6.

Submission

Plagiarism. We explicitly emphasize that the exercise tasks must be solved *independently* by each participant. If source code is made accessible to other participants (intentionally or due to negligence in ensuring a certain minimum level of data security), the respective example will be evaluated with 0 points for all parties involved, regardless of who originally created the code. Similarly, it is not permissible to use code from the internet, books, or other sources. Both automatic and manual plagiarism checks will be conducted.

The use of LLM-based tools such as ChatGPT or Copilot is not particularly prohibited. However, it should be kept in mind that generated content with similar tools is very likely to be highly similar to the generated solution of others, which we also consider to be plagiarism. We therefore strongly encourage you to carry out the exercise independently and without LLM-based tools.

Build and Submission. You need to create your repository on `https://courseware.icg.tugraz.at/`, after which you can access it on `https://assignments.icg.tugraz.at/`. You will need to use git to pull and push code in your repository. Your repository comes configured with a `main` branch, which you can use to freely develop and experiment with changes.

For submission you *must* create a `submission` branch and push your code into this branch. This will also trigger an automated test, for which you can view build logs, output logs and rendered images in the CI/CD section of the gitlab webinterface. **We will only grade solutions that have been pushed to the submission branch of your repository!** To build the framework, please follow the build instructions in the `README.md` file within your repository.