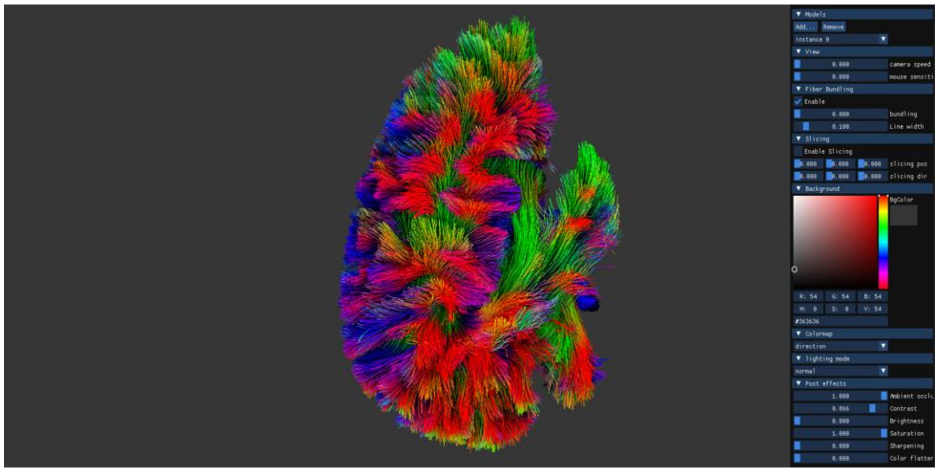
FiberVisualizer

Zhang Zichuan, Davey Vermeulen



```
▼ Models
Add...  Remove
instance 0
▼ View
0.000  camera speed
0.000  mouse sensiti
▼ Fiber Bundling
✓ Enable
0.000  bundling
0.100  Line width
▼ Slicing
Enable Slicing
0.000  0.000  0.000  slicing pos
0.000  0.000  0.000  slicing dir
▼ Background
BgColor
R: 54  G: 54  B: 54
H: 0   S: 0   V: 54
#363636
▼ Colormap
direction
▼ lighting mode
normal
▼ Post effects
1.000  Ambient occlu
0.066  Contrast
0.000  Brightness
1.000  Saturation
0.000  Sharpening
0.000  Color flatter
```

Fig. 1: Screenshot of the FiberVisualizer application rendering a model of human brain with user-defined effects appied.

Abstract—FiberVisualizer is a software application capable of loading and visualizing tractography data containing possibly large number of fibers in an efficient way. The application emphasizes the spatial and structural relation between geometric primitives within the model. It provides users with a set of interactive options for exploring the model in real-time.

Index Terms—Tractography, visualization, real-time rendering

1 Introduction

The context of the project is to efficiently visualize tractography dataset consisting of massive fibers (streamlines). An interactive visualizer for tractography data can be quite useful in medical scenarios. It offers a comprehensive depiction of neural pathways within human brain, enabling an intuitive understanding of the spatial and structural information crucial for both anatomy identification and medical education.

Appropriate rendering techniques should be applied to clearly reveal the geometric properties of the model. Meanwhile, all the computation involved is supposed to be performed in real-time to achieve a high level of interactivity. This criteria puts a great challenge on both the design and implementation of the visualizer.

An existing application for visualizing tractography data can be found in [1], which offers a variety of rendering options but experiences significant performance degradation when dealing with excessively large model sizes. Another example is [2], which employs a voxel-based rendering method with transparency enabled.

This project aims to deliver an application *FiberVisualizer* for visualizing large scale tractography dataset with both quality and performance guaranteed. Features of various aspects are implemented to collectively provides the user with an intuitive and interactive visualization. We focus on innovative features absent in existing applications, including the feature of *fiber bundling* used to reduce the complexity of the model, *ambient occlusion* used to emphasize spatial relationship between geometric structures, *post-processing effects* used to enhance the rendered image, and the all-on-GPU working pipeline. Meanwhile, we keep the application as light-weight as possiable with limited dependencies.

2 Requirement analysis

We identified a set of features the application should have to enable intuitive and interactive visualization, focusing on various aspects including interaction, geometry and rendering:

- TCK data loading and caching.
- Switching between loaded models for visualization.
- View controlling (rotation, moving forwards/backwards).
- Fiber bundling to reduce model complexity.
- Model slicing to inspect the internal structures of the model.
- Color mapping options for shading the model.
- Lighting options for improving the realism of the shaded model.
- Ambient occlusion to reveal spatial relationship between fibers/bundles.
- Post-processing effects to enhance the rendered image.
- A high-performance pipeline running on GPU.

3 Tools

The program is written in the C++ language and developed using Visual Studio. User interaction is handled by Dear ImGUI, a popular UI library for C++ applications. To keep our application lightweight, we will not use any existing rendering engine or gaming engine. Instead, we implement the entire rendering pipeline based on OpenGL APIs. Specifically, we use the version OpenGL 4.3, which allows us to use compute shader to accelerate computationally-intensive works on GPU.

4 Application overview

As shown in Fig. 2, the system is implemented in a modular fashion to separate concerns of various aspects. Specifically, the GUI component handles user input and interprets them as rendering parameters. The loaded models are managed by the module Scene, which is also responsible for extra functionalities (e.g., fiber bundling). Renderer is the core module that performs rendering given a model and parameter settings.

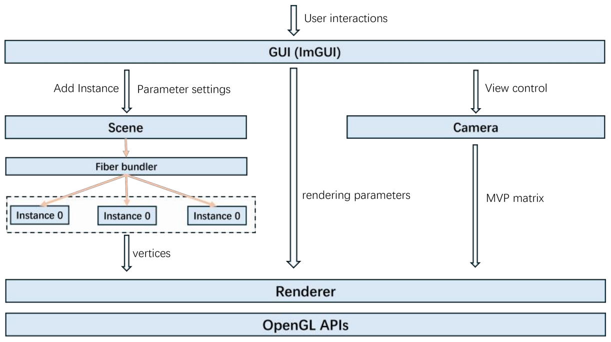


Fig. 2: Architecture of proposed system

The application runs a pipeline consists of three stages, namely the bundling stage, the rendering stage and posting-processing stage. Fiber bundling acts as a pre-processing step that would change the geometry of the original model, which is invoked only when the related parameter is updated by the user on GUI. A series of render passes, including geometry pass, lighting pass and AO pass, sequentially form a rendering pipeline that computes a color value to visible fragments on screen. The post-processing stage applies additional visual effects to the image produced by the rendering stage, which works completely on screen space instead of the entire model.

5 Fiber data encoding

5.1 Data loader

The application allows user to visualize a model in *.tck* format. A tck file loader is implemented to read a set of fibers, which are then stored as a set of points with each assigned a basic attribute *position*. Additional data will be created to mark the beginning and the ending of a fiber. This information is persistently stored in memory until the user switches to another model or removes that model on GUI.

5.2 Geometric attributes

In addition to the basic attribute *position*, another two geometric attributes required by subsequent operations on the model will be assigned to each point of the fibers at the initialization stage, namely *direction* and *normal*. For each line segment s in a fiber, its direction is calculated as the differential vector between the position of its endpoints:

$$Dir(s) = normalize(p_1(s) - p_2(s))$$

This direction is assigned to the two endpoints of s.

Attribute *normal* stores for each point its normal vector, which is required by both the algorithm of fiber bundling and the ambient occlusion. The normal vector of a point is determined as follows:

$$L(p) = Dir(s_2) \times Dir(s_1)$$
$$N(p) = normalize(Dir(s_1) \times L(p))$$

Where $Dir(s_1)$ and $Dir(s_2)$ denote the direction vectors of two consecutive segments that point p connects. $L(p)$ is an intermediate variable and $N(p)$ is the normal vector assigned to point p.

6 Slicing

Another feature of geometry we have in the application is the arbitrary directional slicing, which allows user to slice the model using a plane and have an inspection into the internal structures of the model. A slicing plane is defined by two parameters, the position of some point contained in the plane and the normal vector of the plane.
Given a slicing plane (defined on GUI), slicing is performed by cutting the line segments of the fibers according to their relative position to the plane, which would be in three cases:

- Lie above: If both of the endpoints of s lie above the plane, then discard the segment (by assigning an infinite position to both points).
- Lie below: If both of the endpoints of s lie above the plane, then preserve both points without doing anything.
- Intersect: If $s = (s_1, s_2)$ intersects the plane at some point p, then update s to a new segment s' with (s_1, p) as the endpoints if s_1 lies below the plane, or (p, s_2) if s_2 lies below the plane.

We use a compute shader to perform the cutting operation individually for all segments in parallel.

7 Fiber bundling

7.1 Motivation

Tractography dataset typically contain a large number of fibers. High density of geometric primitives prevent users from obtaining a clear view over the sub-level structures in the model. To this end, we apply a technique called fiber bundling to the dataset as a pre-processing step, which was originally proposed for visualizing 2D graphs with massive edges and has been extended to 3D streamlines. Fiber bundling would greatly reduce the complexity of a model while roughly preserving the geometric properties of the original model.

7.2 Algorithm

In the proposed application, we apply an interative algorithm from a literature [3] to perform fiber-bundling. Generally, the algorithm involves seven stages, namely voxelization, resampling, density estimation, gradient estimation, advection, smoothing and relaxation. Following sub-sections give a detailed description for the full pipeline step-by step.

Voxelization

To perform fiber bundling, the 3D space that contains the model needs to be voxelized. Specifically, we first compute the axis-aligned bounding box (AABB) for the model and divide the box into equal-size unit voxels (small cubes). The size of a voxel should be small enough such that the average number of points within in a single voxel is limited, which is necessary to obtain a high accuracy result of subsequent operations (e.g., density estimation).
By testing the quality of the result produced by various parameter settings, we set the size of a uint voxel to around 0.003 of the size of the bounding box (i.e., 300 - 400 voxels per axis), as a trade-off between the accuracy and the GPU memory occupation.
The next step is to count for each voxel the number of points it contains. This information is stored in a buffer *voxelCount*.

Fiber resampling

As noted in [4], the streamlines need to be resampled to have denser points with equal-size intervals, which is crucial for the bundling algorithm to work well.
For each line segment in the original fibers, we split it into multiple segments with equal length. As an empirical setting, we set the interval to be 0.01 of the length of the z-axis of the bounding box.

Density estimation

With the buffer *voxelCount* obtained from previous step, we then estimate the density for each voxel. This operation is formulated by the following equation:

$$\rho(v_i) = \sum_{v_j \in kernel(v_i)} K(v_i, v_j) voxelCount(v_j)$$

where $K(v_i, v_j)$ is a parabolic function formulated by:

$$K(v_i, v_j) = 1 - dis(v_i, v_j)^2 / R^2$$

That is, the density of a voxel is estimated by the weighted sum of the number of points within its surrounding voxels, and the voxels with larger distance from the target voxel contribute less to the estimation. The number of voxels used for estimation for a voxel is controlled by the kernel radius R, which corresponds to the degree of bundling Fig. 7. We provide the kernel size as a user-defined parameter on GUI.
The operation given above corresponds to performing a 3-dimensional convolution on the texture *voxelCount*, which is inefficient due to massive memory access on GPU and had been identified as a bottleneck of the performance of the bundling pipeline. Hence, we employ the method proposed in [3] to improve the efficiency of 3D convolution. The idea is to compose the 3D convolution to three separate 1D convolution passes, one pass on the X-axis, one pass on the Y-axis and one pass on the Z-axis.
By decomposition, we can obtain a mathematically identical result to the original 3D convolution, but with much less time consumption.

Gradient estimation

Based on the texture *densityMap* obtained from the previous step, we further estimate the gradient for each voxel $v_i(x,y,z)$ using the central difference method:

$$\frac{\partial \rho}{\partial x}(x,y,z) \approx \rho(x+1,y,z) - \rho(x-1,y,z)$$
$$\frac{\partial \rho}{\partial y}(x,y,z) \approx \rho(x,y+1,z) - \rho(x,y-1,z)$$
$$\frac{\partial \rho}{\partial z}(x,y,z) \approx \rho(x,y,z+1) - \rho(x,y,z-1)$$
$$\nabla \rho(v_i) = \left[\frac{\partial \rho}{\partial x}, \frac{\partial \rho}{\partial y}, \frac{\partial \rho}{\partial z}\right]$$

Advection

The next step is to update the position of the points in fibers, based on the estimated gradient map. Instead of updating the point by moving along its gradient, we adopt the method of constrained advection proposed in [3] which ensures that the shape of the original fiber is roughly preserved after advection.
Let d_i denote the direction vector of point p_i, $\nabla\rho(v_i)$ denote the gradient of p_i, and R denote the kernel radius, then the position of p_i is updated as:

$$p_i' = p_i + (R \frac{\nabla \rho(v_i)}{|\nabla \rho(v_i)|} \cdot u_i) u_i$$

Where u_i is a normalized vector orthogonal to d_i and is computed as:

$$u_i = d_i \times (\nabla \rho(v_i) \times d_i)$$

That is, the advection of a point is performed by first projecting the unconstrained moving vector $R\frac{\nabla\rho(v_i)}{|\nabla\rho(v_i)|}$ onto the orthogonal direction u_i, and then moving the point along the constrained vector $(R\frac{\nabla\rho(v_i)}{|\nabla\rho(v_i)|} \cdot u_i)u_i$. This is illustrated in Fig. 3.

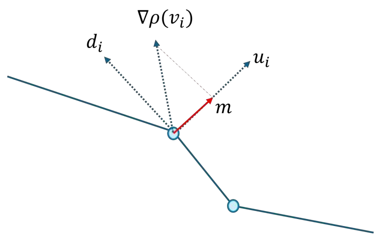


Fig. 3: Instead of directly move the point along its gradient $\nabla\rho(v_i)$, we update it along the contrained vector u_i orthogonal to its direction vector d_i.

Smoothing

We then smooth the updated fibers to have a better quality. This is done by updating a point p_i by averagely aggregating the position of its neighbouring points:

$$p_i' = (1-\phi)p_i + \frac{\phi}{2L+1} \sum_{j=i-L}^{i+L} p_j$$

where L and ϕ are parameters to control the smoothness of resulting fibers. We set ϕ to a constant value 0.90 and L to a value equal to the kernel radius R.

Relaxation

In the relaxation stage, we further set a strong constraint to the updated fiber to avoid a large deviation to its original shape. This is done by simply mixing the original fiber with the updated one, operating on the points:

$$p_i'' = (1-\alpha) \cdot p_i' + \alpha \cdot p_i$$

Where α is a parameter controlling the degree of relaxation. We set α to a constant value 0.85.

Iterative bundling

The sequential operations given above need to be repeated for several times to produce a significant effect of bundling. We set the number of iterations to 15 and slightly decrease the kernel radius (by a factor 0.95) at each iteration to adapt to the increasing density of fiber bundles. The entire pipeline for fiber bundling is shown Fig. 4.

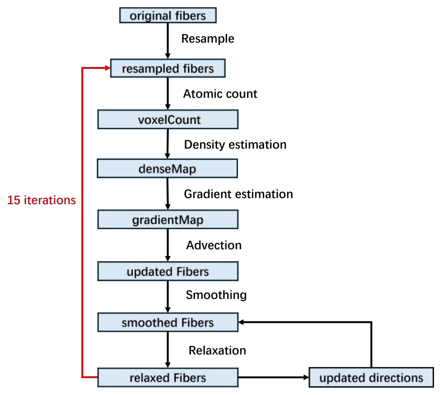


Fig. 4: Fiber bundling pipeline. Operations are performed repeatedly for 15 iterations.

GPU-based implementation

The complexity of the operaions given above is linear to either the total number of voxels or the total number of points in the fibers, which is impossible to be computed in real-time if running on CPU. Fortunately, the involved computations are logically independent and identical to

each other, therefore can be performed in parallel.
We utlize the compute shader, a feature supported by OpenGL 4.3+, to perform fiber bundling completely on GPU. We store all the required data in storage buffer objects (SSBO) or 3D textures, define the unit computation in a compute shader to be invoked in parallel and dispatch the computational task using a build-in command *glDispatchCompute*. Specifically, for the stage of voxelization, the compute shader is invoked for all points in parallel that will simultaneously write to the buffer *voxelCount*. For density estimation and gradient estimation, the shader is invoked for all voxels that will write to the 3D textures *densityMap* and *gradientMap* respectively. In the stage of advection, the shader is again invoked per point, which queries the original coordination of that point and updates it according to the gradient map. Similarly, in the stage of smoothing and relaxation, the compute shader is again invoked for all points to update the position of the points in parallel.

8 Rendering pipeline

As shown in Fig. 5, the rendering pipeline used in the proposed application is composed of four render passes, sequentially forming a deferred shading scheme. The pipeline is highly efficient since only the geometry pass operates on the entire model while all subsequent render passes operates completely on the screen space. Section 9 - 12 gives a detailed description of the four render passes.

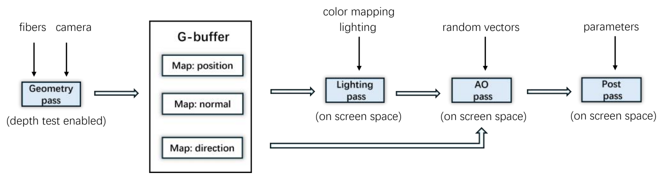


Fig. 5: Rendering pipeline integrated in the application. The white boxes represent the intermediate textures, the light blue boxes represent the render passes. Thin arrows denote the input of each render pass.

9 Geometry render pass

9.1 Concept

Geometry pass is often used as the first render pass in a deferred rendering pipeline. In geometry pass, only the fragments that pass depth testing will write to the G-buffer, which stores the geometric information of the rendered scene such as position, normal vector and so on. Using a dedicated geometry pass ensures that all invisible fragments (i.e., occluded by some other fragments) in the scene are discarded at an early stage of the rendering pipeline, so that successive render passes can work completely on the screen space instead of the entire model, which greatly reduces the complexity of computation.
In our implementation, the geometry pass produces three textures useful for subsequent passes, namely the texture *texPosition* storing world position of visible fragments, *texDirection* storing direction of fragments and *texNormal* storing normal vectors (in world space) of fragments.

9.2 Rendering primitives

Fibers naturally fit into the line rendering mode *GL_LINES*, a rendering primitive supported by OpenGL. At the rasterization stage, for each line segment, the attributes *position*, *direction* and *normal* of its two endpoints are interpolated and assigned to interior fragments, which will then be written into the three textures in G-buffer.
The line width is specified using the command *glLineWidth*. We provide this as an adjustable parameter on GUI.

10 Lighting render pass

10.1 Concept

The lighting pass acts as the second stage in a deferred rendering pipeline, which takes the textures produced by geometry pass as input and calculate a color to visible fragments on screen space. The color assigned to a fragment is jointly determined by option of color mappingSect. 10.2 and the option of rendering mode Sect. 10.3. Color mapping sets a base color for a fragment (or professionally speaking, the *albedo*), while a rendering mode applies some lighting model to the fragments to improve visual realism.

10.2 Color mapping

We offer three options on GUI for color mapping:

- Constant: A constant color selected by user on GUI.
- Direction: Color a fragment by the absolute value of the direction vector of the line segment it lies in.
- Normal: Color a fragment by the absolute value of its normal vector. The color can change by view since the normal vector of a point is defined with respect to the view direction of the camera.

10.3 Rendering mode

We offer two rendering modes to the user, the normal one without any lighting and the physically based one that allows for defining the material of the fibers.

Normal rendering

If user selects the normal rendering mode, then no lighting is applied to the fibers. The fragments are simply shaded by the albedo defined by color mapping.

Physically based rendering

The method of physically based rendering we applied in the application is mainly drawn from the OpenGL official tutorial [5]. The key idea of physical based rendering is to simulate the interaction between lighting rays and objects in a physically plausible way, can be formulated by:

$$L_o(p,\omega_{\mathbf{o}}) = \int_{\Omega} f_r(p,\omega_{\mathbf{i}},\omega_{\mathbf{o}}) \cdot L_i(p,\omega_{\mathbf{i}}) \cdot (\mathbf{n}\cdot\omega_{\mathbf{i}})\, d\omega_i$$

That is, the color of some shading point p seen by the viewer is the result of an integration over the contribution of lighting sources from all directions. The key component of this equation is the concept of *Bidirectional Reflectance Distribution Function (BRDF)*, which describes how the radiance from some direction w_i distributes over the space. The *BRDF* is composed of three components:

- Normal distribution function: defines how much the microfacets of some fragment are aligned to the halfway vector, influenced by the fragment's roughness.

$$D(\mathbf{n},\mathbf{h},\alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2}$$

- Geometry function: defines how much the microfacets of some fragment occlude each other, influenced by roughness.

$$G(\mathbf{n},\mathbf{v},k) = \frac{\mathbf{n}\cdot\mathbf{v}}{(\mathbf{n}\cdot\mathbf{v})(1-k)+k}$$

- Fresnel equation: defines the ratio of reflection at different angles, influenced by metalness.

$$F(\mathbf{h},\mathbf{v},F_0) = F_0 + (1-F_0)(1-(\mathbf{h}\cdot\mathbf{v}))^5$$

Where $\mathbf{n}$ is the normal vector of a shading point, $\mathbf{h}$ is the half vector between the light direction and view direction. α and F_0 define the roughness and metalness. Typically, physical based rendering is combined with *image based lighting (IBL)*, which is a technique that simulates the continuous and dense lighting sources in real world by sampling from a prefabricated *enviroment map*. For the purpose of simplicity, we set a fixed set of 30 directional lighting sources around the model, which is sufficient for showing the material properties of the fibers (i.e., the roughness and metalness) through the luminance distribution.

11 AO RENDER PASS

11.1 Motivation

The spatial and structural relation of fibers is crucial for understanding the tractography data. Such information is typically reflected by the color distribution over the shaded fragments. Displaying structural information of a model at a high accuracy corresponds to shading the model in a manner that captures global geometric information. However, the basic local illumination models (e.g., Phong model) color a fragment without accounting for the existence of occluders, making it hard to distinguish structures in dense regions.
To this end, we apply an efficient technique that works on screen space to roughly approximate the visual effect of global illumination, namely ambient occlusion.

11.2 A basic solution: SSAO

Screen space ambient occlusion (SSAO) is a commonly used technique to approximate the global illumination, usually used as a post-processing step following a local illumination shading process. The main idea of SSAO is to estimate for each visible fragment on screen that how much it is occluded by surrounding objects. For a shading fragment, SSAO randomly generates a set of sampling points around its normal vector, then projects the sampling points to the camera's view plane to check whether it is occluded by some other visible fragment on the screen space (this is done by comparing the shading fragment's z-value with the value stored in texture *position*). The occlusion for the fragment is then determined by the portion of sampling points being occluded. With the computed occlusion, the brightness of the shading fragment is accordingly decreased, resulting in a shadowing effect. The implementation of SSAO algorithm is given in algorithm 1.

Algorithm 1: SSAO

```
Data: Depth buffer $D$, Normal buffer $N$, Position buffer $Pos$
Data: Controling parameter $radius$, Camera parameters
Result: SSAO buffer
Initialize SSAO buffer $S$;
for each pixel $(x,y)$ do
    $d_{\text{current}} \leftarrow D(x,y)$;
    $n_{\text{current}} \leftarrow N(x,y)$;
    $pos_{\text{current}} \leftarrow Pos(x,y)$;
    $occlusion \leftarrow 0$;
    for $i \leftarrow 1$ to $numSamples$ do
        $sVec \leftarrow$ randomly sampled vector in world space;
        $sPoint \leftarrow pos_{\text{current}} + sVec \cdot \text{radius}$ ;
        $(x', y') \leftarrow$ project sampling point to screen space ;
        $d_{\text{sample}} \leftarrow D(x', y')$;
        $n_{\text{sample}} \leftarrow N(p_x, p_y)$;
        if $d_{sample} < d_{current}$ then
            $occlusion \leftarrow occlusion + \max(0, \vec{n}_{\text{current}} \cdot \vec{n}_{\text{sample}})$;
        end
    end
    $occlusion \leftarrow occlusion/\text{numSamples}$;
    $occlusion \leftarrow \max(0, occlusion - \text{bias})$;
    $S(x,y) \leftarrow occlusion$;
end
```

For purpose of efficiency, the random sampling vectors (64 in our implementation) are pre-generated in tangent space when initializing the rendering pipeline. When executing the AO pass, these random vectors are transformed to get the actual position of the sampling points in world space.

11.3 An Extended algorithm: Multi-scale ambient occlusion

Inspired by [6], we implemented a extended version of the classic SSAO, namely the multi-scale ambient occlusion (*MSAO*). The key idea of *MSAO* is to detect occluders at multiple scales in the scene, with decreased number of sampling points at large scales (see Fig. 6). In practice, we generate 64 sampling points at the smallest scale and halve the number at subsequent scales.

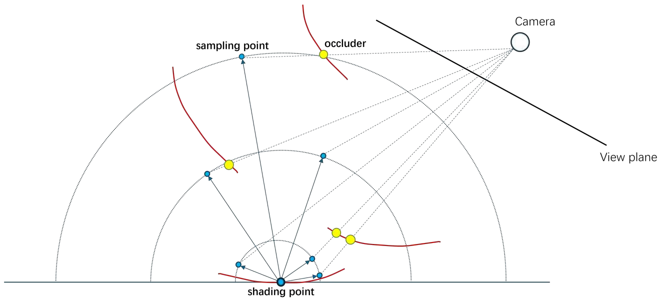


Fig. 6: Concept of multi-scale ambient occlusion. The red curves represent the fibers, blue circles with smaller size denote the sampling points at multiple scales, and the yellow circles indicate the detected occluders.

The multi-sampling scheme is capable of capturing the spatial relationship between both small structures and large structures since fragments at different distances may produce shadow on the shading fragment. (see the result in Fig. 9)

12 POST-PROCESSING EFFECTS

12.1 Motivation

A set of post-processing effects is provided as options to enhance the rendering image produced by the major render passes. Following gives a brief description of the included screen-space effects.

12.2 Color flattening

After the AO pass, fragments are shaded by continuous color. Color flattening maps the shaded color from continuous color space to discrete color space:

$$RGB' = \lfloor RGB/u \rfloor$$

where n denotes the granularity of the discrete color space, controlled by user.

12.3 Contrast

Applying contrast to the image highlights the difference between color of pixles by brightening pixels with high values and darkening the lows:

$$RGB' = c \cdot (RGB - 0.5) + 0.5$$

where c is a user-defined parameter for controlling the resulting contrast.

12.4 Brightness

Change the brightness of rendering image by adding an additional term:

$$RGB' = RGB + \delta$$

where δ is a user-defined parameter.

12.5 Saturation

Saturation is about highlighting difference by increasing color values. Higher saturation indicates more intense colors in the image. We do saturation/desaturation as follows:

$$L = RGB \cdot w$$
$$RGB' = vec3(L) \cdot (1 - s) + s \cdot RGB$$

Where s is a parameter determining saturation, w is a constant weight used to get the luminance of the original color. We set $w = (0.2125, 0.7154, 0.0721)$ as a common practice when converting RGB color to grayscale.

12.6 Sharpening

Sharpening works by computing the contrast between a pixel and its neighbouring pixels, and then update that pixel's color by adding the obtained contrast. This corresponds to performing a 2D convolution on the image using a 3 X 3 kernel:

$$c'_{ij} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix} \circ \begin{pmatrix} c_{i-1,j-1} & c_{i-1,j} & c_{i-1,j+1} \\ c_{i,j-1} & c_{ij} & c_{i,j+1} \\ c_{i+1,j-1} & c_{i+1,j} & c_{i+1,j+1} \end{pmatrix}$$

We then add c'_{ij} to the original color c_{ij} by a user-specified factor s that controls the degree of sharpening:

$$RGB'_{ij} = s \cdot c'_{ij} + c_{ij}$$

13 Results

13.1 Fiber bundling

Fig. 7 and Fig. 8 give the result of applying fiber bundling. Comparing Fig. 7 (a), (b) and (c), we can see that our algorithm for fiber bundling significantly reduce the complexity of the original model while roughly preserves the original geometric structures and the spatial relationship between the fibers.
However, the bundling algorithm is unable to preserve the geometric features of certain structures. As shown in Fig. 8 (b), the manifold-like structure of the original model is now broken into separate bundles, which is caused by isotropic bundling as noted in [3].
Meanwhile, the algorithm failed to separate fibers in high-density regions (see Fig. 8 (c)), preventing users from having a clear inspection into structures at micro level.

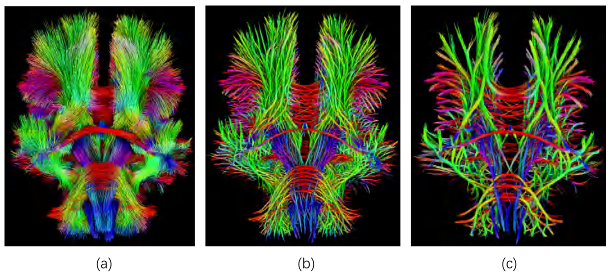


Fig. 7: Result of fiber bundling controlled by kernel width. (a) Unbundled. (b) Bundling with kernel width = 10 (c) Bundling with kernel width = 20. The bounding box of the model is discretized to 175 X 525 X 350 unit voxels. The fibers are resampled to have roughly three times the original number of points.

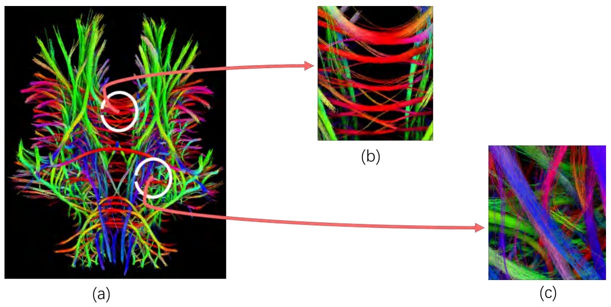


Fig. 8: A detailed view of bundled model. (a) Bundling with kernel width = 20. (b) Local details of the central region. (c) Local detail of the high-density region.

13.2 Muli-scale ambient occlusion

Fig. 9 shows the visual effect of ambient occlusion. Comparing Fig. 9 (b),(c) and (d), we can see that the proposed multi-scale sampling scheme effectively captures the occlusions between both micro-level structures (i.e., the individual fibers) and between large-scale structures (i.e., the bundles), thus providing a comprehensive visualization of the model's geometric features. The effectiveness of *MSAO* can also be seen from some local details shown in Fig. 10 (b) and (c).

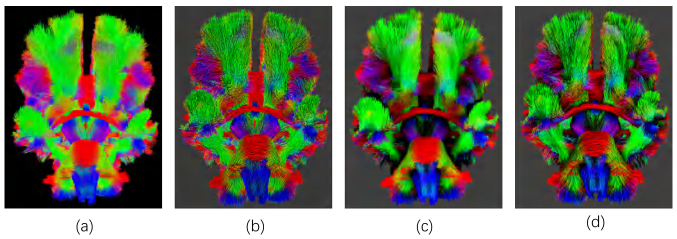


Fig. 9: Resulting images produced by detecting occlusions at different scales. (a) Flat shaded model without shadow. (b) Detect occlusions only at a small scale with sampling radius $r = 3$, (c) Detect occlusions only at a large scale with $r = 50$.(d) Detect occlusions at 5 scales with sampling radius ranging from 3 to 50.

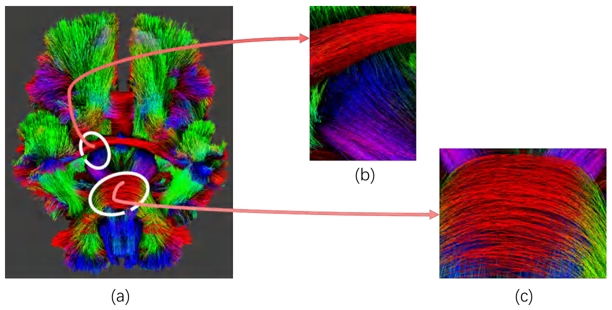


Fig. 10: Local details of the model with multi-scale AO applied. (a) Result of multi-scale sampling. (b) Local details 1. (c) Local detail 2.

13.3 Post-processing

The result of applying various post-processing effects is given in Fig. 11.

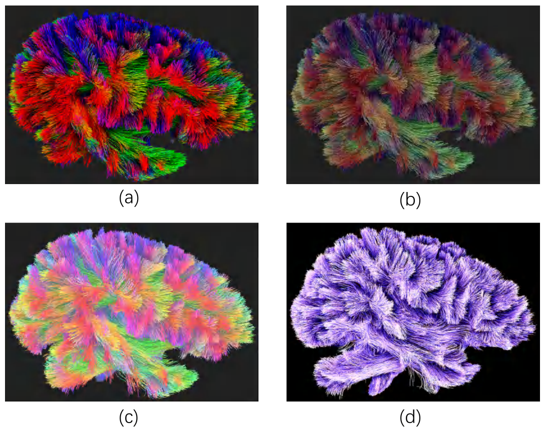


Fig. 11: Visual effects of post-processing. (a) High contrast. (b) Desaturation. (c) Lifted brightness. (d) Sharpness

14 Performance evaluation

In this section, we give the evaluation results of performance of the application, using the system configuration given in Appendix A. Table 1 shows that the proposed rendering pipeline achieves a high-level performance of over 80 fps even given a large dataset containing 10^5 fibers and 10^6 points. FPS drops for only 27.6% when the size of model is increased by a factor of 5, which indicates the scalability of the pipeline. The result meets our expectation since most render passes of the pipeline works on the screen space, which is only relevant to the number of pixels of textures while not the model size. Meanwhile, both the host memory usage and GPU memory usage remains within an acceptable level fitting in a PC-level system.
A significant flaw of our application can be seen from Table 2. When the pipeline for fiber bundling is enabled, the memory usage on both the host side and GPU side significantly increases, placing a high requirement on user's system hardware. This is an expected result of the resampling operation in the bundling pipeline (Sect. 7.2) and the large 3D textures created for voxels (Sect. 7.2)).

Table 1: Performance evaluation for the rendering pipeline on 5 datasets of increasing size. Rendering is performed on texture with 1920×1080 resolution. FPS is averagely measured over 100 seconds. Fiber bundling is disabled.

Model Size	FPS	Host mem	GPU mem
28.8k fibers, 0.63M points	113.6	0.5GB	0.4GB
57.6k fibers, 1.18M points	108.7	0.5GB	0.4GB
86.4k fibers, 1.96M points	90.5	0.5GB	0.5GB
115.2k fibers, 2.78M points	92.5	0.6GB	0.6GB
144k fibers, 3.62M points	82.3	0.6GB	0.7GB

Table 2: Performance evaluation for the rendering pipeline on 5 datasets of increasing size when fiber bundling is enabled.

Model Size	FPS	Host mem	GPU mem
28.8k fibers, 2.19M points	110.8	1.1GB	2.6GB
57.6k fibers, 4.18M points	87.0	1.5GB	2.8GB
86.4k fibers, 5.62M points	77.5	1.6GB	2.9GB
115.2k fibers, 7.64M points	69.0	1.9GB	3.1GB
144k fibers, 9.87M points	61.7	2.4GB	3.3GB

Another computationally-intensive part of our application is performing fiber bundling. We fix the total number of voxels to 175 × 525 × 350 and evaluate the influence of two parameters on the running performance, namely the model size and the degree of bundling. The result is given in Table 3 and Table 4. We can see that the executing time of the fiber bundling pipeline is mainly determined by the model size while weakly relevant to the degree of bundling. The executing time of fiber bundling is limited to around 1 second even given a large model, which ensures a quick response to user's operation.

Table 3: Performance of fiber bundling on model size. Number of voxels fixed to 175 × 525 × 350, kernel width fixed to 15 voxels.

Model Size	Execution time (seconds)
28.8k fibers, 2.19M points	0.27
57.6k fibers, 4.18M points	0.34
86.4k fibers, 5.62M points	0.63
115.2k fibers, 7.64M points	0.90
144k fibers, 9.87M points	1.10

15 Conclusion

In conclusion, the implemented application *FiberVisualizer* is capable of providing an intuitive visualization of large scale tractography dataset with spatial and structural information emphasized, by applying integrated algorithms of fiber bundling, multi-scale screen space ambient occlusion and a set of post-processing effects. Meanwhile, the proposed all-on-GPU pipeline guarantees that all involved computationally-intensive works can be performed in real-time, which meets the requirement of highly interactive.
However, there is still room for improvement in terms of the quality of the result produced by fiber bundling. In addition, the GPU memory footprint of fiber bundling remains to be optimized to reduce the hardware requirement ro run the application.

Table 4: Performance of fiber bundling on the degree of bundling (controlled by the kernel radius). Number of voxels fixed to 175 × 525 × 350, model size fixed to [144k fibers, 9.87M points].

Kernel radius	Execution time (seconds)
6	0.96
8	0.99
10	1.00
12	1.09
14	1.06

16 Future work

Given limited time for this project, we omitted some details when implementing the algorithm of fiber bundling and multi-scale ambient occlusion, including the anisotrophy-constrained bundling proposed in [3] and a carefully designed sampling method proposed in [6]. Our prior future work would be further improving the quality of fiber bundling and ambient occlusion by encompassing all algorithmic details included in the literature, as well as testing some innovative ideas for specific scenarios.
The second part of the future work is to integrate more advanced rendering techniques. The rendering features we have implemented so far fall under the diagram of surface-based rendering, where the intersection between light rays and geometric primitives all take place on the surface of an object. However, such model is not able to simulate the phenomenon of light penetrating the interior of an object, and thus cannot show the effect of transparent materials. One direction of future work would be implementing the voxel-based rendering.
Moveover, we have the plan to do exploration on photorealistic rendering of fiber data as in [7], which involves combining physical-based rendering, voxel-based rendering and path tracing.

Acknowledgments

We would like to thank the professors for their guidance during this project and the opportunity to gain a deep understanding of knowledge in computer graphics.

References

[1] Surf Ice. `https://www.nitrc.org/projects/surfice/`.
[2] NeuroTrace. `https://gitlab.com/Besm/neurotrace`.
[3] Steven Bouma, Christophe Hurter, and Alexandru Telea. Structure-aware trail bundling for large dti datasets. *Algorithms*, 13(12), 2020.
[4] Matthew van der Zwan, Valeriu Codreanu, and Alexandru Telea. Cubu: Universal real-time bundling for large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 22(12):2550–2563, 2016.
[5] LearnOpenGL. `https://learnopengl.com/PBR/TheoryLearn` OpenGL PBR Theory.
[6] Sebastian Eichelbaum, Mario Hlawitschka, and Gerik Scheuermann. Lineao—improved three-dimensional line rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):433–445, 2013.
[7] Dorin Comaniciu, Klaus Engel, Bogdan Georgescu, and Tommaso Mansi. Shaping the future through innovations: From medical imaging to precision medicine. *Medical Image Analysis*, 33:19–26, 2016. 20th anniversary of the Medical Image Analysis journal (MedIA).

A Hardware

Evaluation is conducted using a PC-level system with the following configuration:

Evaluation system specifications	
CPU	Intel Core i7-12700H 2.30 GHz
GPU	GeForce RTX3050
Resolution	1920×1080
Memory	32 GB
OS	Windows 11

B Contributions

We divide the work according to our skills and experience in specific aspect. Generally, Zhang was responsible for OpengGL related coding, while Davey was responsible for interaction-related work and the implementation of post-processing effects. We each worked about 10 hours a week on this project.

Davey's contributions

- TCK file loader
- GUI design and implementation (using Dear ImGUI)
- User interactions
- View controlling (camera)
- Implementation of post-processing effects (shaders)
- Color mapping options
- Evaluation of the post-processing effects
- Part of the report writing

Zhang' contributions

- Rendering pipeline
- Implementation of the fiber-bundling pipeline using compute shaders
- Implementation of model slicing
- SSAO implementation (shaders)
- Physically based material (study and implementation)
- Evaluation of the effects of fiber bundling, ssao, msao
- Performance evaluation
- Part of the report writing