NeuroTrace

Besm Osman and Mestiez Pereira

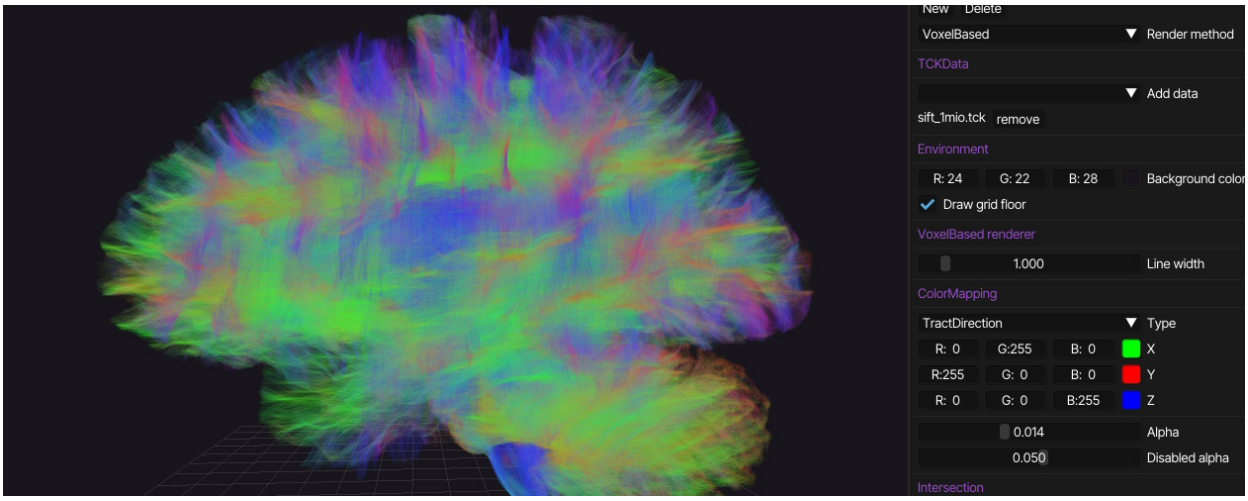

Fig. 1. Screenshot of the NeuroTrace application visualizing one million semi-transparent *streamlines* using the VoxelBased rendering method.

Abstract—NeuroTrace is a software application capable of reading, interpreting, and visualizing MRI tractography data. The application will render one or more visually appealing three-dimensional models consisting of *streamlines* representing white matter tracts within the human brain. It will provide tools for exploring and analyzing the visualization in *real-time*. NeuroTrace will provide a graphical user interface as the main interface between the user and the application, in addition to keyboard shortcuts and mouse controls. A short video teaser be seen at: `https://www.youtube.com/watch?v=BaQmGj3dmsU`

1 Introduction

Medical imaging is an important process in the medical field as it provides great insight into the human body, and it has become an essential tool in clinical analysis and treatment. The datasets that are generated during these processes are often extremely large as they attempt to map out the complex structures of the body. The human brain is the most intricate organ and, as such, requires very precise apparatus to accurately map. Brain scanning is done using magnetic resonance imaging (MRI). The resulting data can be processed to generate tractography data, mapping the white matter tracts that partially make up the brain. These datasets consist of connected points that form separate lines of variable length and amount.

Tractography datasets can be very large and are challenging to interactively visualize in *real-time* on a modern computer. A dataset may consist of over a million lines that an application would have to display while maintaining interactivity and supporting a number of desirable tools that can interact with the data. Existing solutions are hard to use, not performant, and/or visually unappealing.

We present NeuroTrace, an interactive MRI tractography data visualizer application. The software was developed to provide a performant, more interactive, and more intuitive solution to tractography visualization. Our objectives with this application are mainly focused on visualization rather than scientific analysis, however it is intended to support some essential analysis tools nonetheless. It aims to increase the accessibility to, and enjoyment of, the exploration of tractography data by improving ease-of-use and performance.

2 Background and Related Work

The context of the problem is the field of medical imaging, specifically the visualization of MRI tractography data or streamlines in general. At the time of writing, numerous applications are being developed that aim to deal with these datasets whether it be visualization, analysis, or both. They are being used in the medical field but are still in the early stages, as *real-time* software for datasets this large have only relatively recently become feasible due to the advances in hardware capabilities.

Existing visualizers

There are several tools capable of visualizing and analyzing tracktography data such as MRtrix3 [8], TrackVis [9] and FiberNavigator [2] [1]. Some more capabable of rendering large datasets then others. Each providing different interaction tools. These tools are used as visual references for our project.

3 Problem definition

MRI tractography datasets are often large, potentially containing hundreds of thousands to over a million lines. Due to their size it can be difficult to render interactively.

We defined with our professor certain criteria we want to accomplish during the project:

- Visualize large amount of diffusion MRI tractography data (.tck data) in *real-time*.
- Provide interactivity such as camera movement (moving, rotating, dragging and zooming)
- Visualizing the streamlines intersecting with an arbitrary sphere.
- Domain specific filters and tooling for analysis.

- Multiple render methods (primitive, tubes, etc)

4 TOOLS

The software is written in the C# language and uses the .NET 7 runtime. Window management, input, and rendering is handled using OpenTK, which provides a thin wrapper around OpenGL. For the user interface we decided on Dear ImGui, a widely used immediate-mode UI library. The input data consists only of a set of streamlines, which are series of three-dimensional points. This data is provided as TCK files, a format that is commonly used for this purpose. The input data is interpreted as described in Sect. 5.1, after which it is left open for further processing by the numerous render methods described.

5 METHODS

In this section we describe the implementation regarding significant parts of the application. Rendering methods are omitted because they are described in their own section (Sect. 6). Before any visualization is rendered, the data has to be interpreted and processed to achieve the optimal performance for rendering and interactivity.

5.1 Reading tractography data

NeuroTrace currently only supports TCK data. We wrote our own TCK reader to extract the metadata and *streamline* data. We support Float32LE format as this was the only format we have encountered in the B.A.T.M.A.N. [7] files and the assignment provided TCK data, but the code can be easily extended to support other formats and other file types. The raw data is processed into a set of streamlines, stored contiguously in memory. After this processing, it is trivial to enumerate the set and perform other processing steps depending on the selected render method.

5.2 Data encoding

There are several existing solutions that encode *streamlines* to optimize render performance allowing rendering of much larger datasets [6]. While this does increase render performance, it also increases the preprocessing time by several magnitudes. In our experiments with Fiblets [6], loading a new TCK file takes approximately three seconds in NeuroTrace/MRtrix [8], while encoding can take multiple hours with Fiblets. This would hinder workflow of users who wants to load a TCK file and see a visualization to such a large degree that it would not be worth any performance gains in rendering. For this reason we decided against encoding tract data in NeuroTrace.

5.3 Voxel representation

The voxel service assigns each point in each *streamline* to a voxel of arbitrary size. Every voxel contains information about the points that intersect with it and the *streamlines* that they belong to. This data structure is especially useful for spatial queries and the implementation of many spatial optimizations. The intersection service uses this for fast distance queries and VoxelBased rendering uses it in the mesh generation algorithm.

Certain files do not have a set step size indicating the maximum distance between *streamline* points. For these files we add interpolated points to make sure voxels are not skipped. In practice we add very few interpolated points as the voxels are much larger than distance between streamline points.

5.4 Shape intersection filter

One core functionality we provide is a sphere intersection filter. This allows the user to specify a 3D point and a radius indicating a sphere. The software then renders the *streamlines* with different alpha values allowing for transparency or completely hiding of intersecting or non-intersecting *streamlines*. Filtering has great performance that works in *real-time* on both small and large datasets (up to 1 million *streamlines*).

The implementations uses our voxel service and OpenGL 'Shader Storage Buffers Objects' [4]. Our voxel service allows us to quickly query the relevant voxels for intersecting *streamlines*, drastically reducing the amount of *streamlines* that need to be checked. The resulting intersection lines are passed by *id* to a Shader Storage Buffer. The fragment shader has access to this 'flags' buffer, and uses it give each fragment the appropriate transparency value.

5.5 Bounds filter

We added a functionality to visualize a subsection of the brain using two vectors indicating cuboid center position and size. Bounds work on a point level, meaning it will mark all *streamlines*) points as either enabled or disabled using the same 'flag' system described in Sect. 5.4.

5.6 Multiple visualizations

NeuroTrace uses two central concepts, namely 'view' and 'visualization'. We define a visualization as an 3D environment. Each visualization has its own loaded TCK data, color mapping, and rendering method. Views render a visualization with a specific camera. Using these two concepts, users can easily render multiple tractography datasets in the same visualization, render the environment from different angles, side by side, etc. We provide camera tools such as "camera sync" to synchronize different cameras to allow for certain analysis.

5.7 Color mapping

NeuroTrace provides several color mapping options:

- Absolute streamline direction: Color vertices based on the absolute line tangent direction.
- MatCap: Color mapping using material capture in the line tangent direction or normal direction for tubes.
- Constant color: User selected color.
- Gradient: Random color per *streamline* based on user selected colors.
- View based: Transparency per *streamline* point based on the similarity of streamline tangent and the view direction, allowing for viewing streamline segments going certain directions.

Color mapping works for all render methods and in combination with intersection and bound filters.

6 RENDERING METHODS

The application is capable of visualizing the *streamlines* in different ways, each offering its own advantages and disadvantages. These *rendering methods* are useful in different contexts (analysis, photorealism, etc.) and can be used simultaneously.

6.1 Primitive lines

The first rendering method we implemented is primitive line rendering. This involved batching *streamlines* in batches using the OpenGL 'Lines' primitive. Each batch consists of one mesh that is rendered in one draw call. A batch can hold around sixteen million vertices therefore drastically reducing the draw calls needed to render the entire set. E.g. our reference file of one million lines with approximately one hundred points per line can be rendered using seventy meshes. Utilizing the depth buffer and OpenGL line thickness, this render method is capable of accurately rendering an appealing visualization of any set of *streamlines*. Drawbacks include the limited amount of *streamlines* that can be drawn in real-time, and that accurate transparency is not feasible because the *streamlines* need to be sorted.

6.2 Tube meshes

Rendering *streamlines* using tube meshes is not uncommon and quite naïve. This method involves generating a "skin" mesh around the given *streamline* given a profile shape. A profile shape is a 2D curve defined by a series of points. This shape is extruded over the length of any given 3D curve. In our implementation, the profile mesh is arbitrary but it is set to an octagon.

The algorithm follows the given *streamline* and uses the precomputed tangent vector to generate the tube mesh. To minimise twist, a smoothed tangent t_s is used instead:

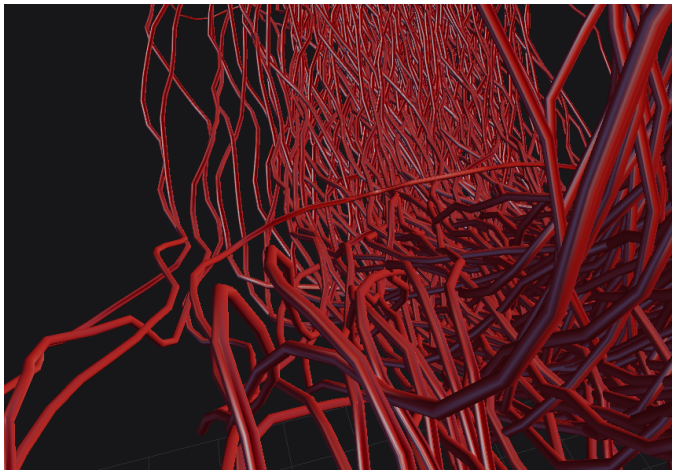

Fig. 2. A bundle of *streamlines* rendered using the tube mesh method using a MatCap material

$$t_s = \frac{p_t \cdot w_p + n_t \cdot w_n}{w_p + w_n} \tag{1}$$

Where p_t, w_p are *previous tangent* and *previous weight* respectively, and where n_t, n_w are *next tangent* and *next weight* respectively. The tangent weight of a point corresponds to its distance to the current point.

Next, for every point omitting the first and last, a set of vectors is calculated that correspond to points of an octagon relative to the point. The cross product of the smooth tangent and its rotation is taken to get the normal vector. The normal vector is rotated and plotted eight times (determined by the desired profile mesh resolution) to form an octagon. Then, edges are formed between the recently created profile and the previous one. This results in one continuous tube with an unnoticeable amount of twist (Fig. 2). Notably, this algorithm does not address start and end caps. This was purposefully omitted as it seemed insignificant when rendering hundreds of thousands of streamlines.

There are few advantages to this render method, one of which is versatility. Since this is a traditional mesh without any fancy tricks, it is open to most traditional rendering techniques including lighting, normal mapping, PBR, MatCaps, etc. Another advantage is that it is the most convincing visualisation in the application because it's a volumetric shape as opposed to infinitely thin lines, or thick lines without surface normals.

Disadvantages are plentiful. To start with, this method uses more video memory and processing power than other methods as the vertex count per point is greatly increased. This results in lower overall performance when compared to primitives. Secondly, a similar effect can be achieved with imposter tube rendering – as seen in [5] – which devalues the naïve implementation further.

6.3 VoxelBased

6.3.1 Motivation

The primitive render method has great performance and was easy to implement; however, it has one major issue: transparency. We were not able to support transparency while using the depth buffer. Since transparency is necessary for other functionality of NeuroTrace, we made it a priority to find a solution.

6.3.2 Existing solutions

We researched existing solutions to solve transparency. The main solution for transparency is rendering transparent objects from back to front. This involves sorting all transparent objects based on their distance to the camera. However, this is not feasible in our context for two reasons. Firstly, streamlines have varied distances from the camera, which makes them difficult to sort. Secondly, as we have an interactive camera and large datasets (1 million lines / 150 million points), ordering would not be performant enough to achieve real-time rendering.

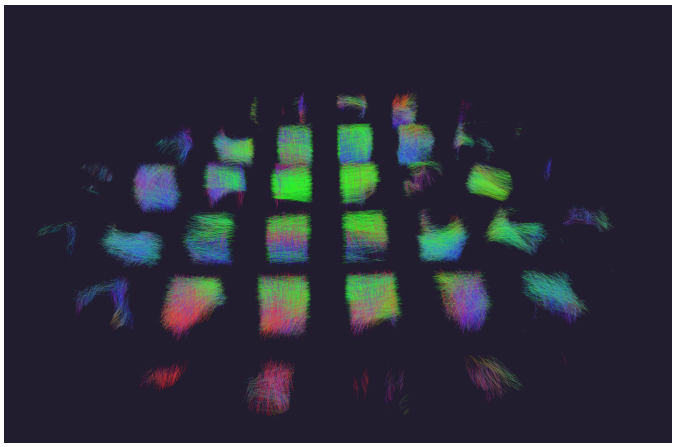

Fig. 3. A slice of voxels, each group of line segments is rendered as one mesh. Voxels are rendered from back to front for correct transparency.

There are order-independent transparency (OIT) solutions. After researching, we concluded that these solutions do not fit our context. Certain solutions required multiple render passes, which would not work in this context. Rendering large datasets with millions of points using the primitive render method was already difficult to do in real-time, and we could not afford to render them multiple times. Other solutions required a very specific, complex render pipeline that precluded us from using certain OpenGL functionality we used for other functionalities and did not want to give up.

6.3.3 Algorithm

We came up with an rendering algorithm we call VoxelBased rendering. Algorithm 1 shows a simplified pseudocode version of the algorithm. The idea is to divide the streamline data into voxels. Each voxel is a 3D cube with a fixed position and size. Per voxel, we generate one mesh that displays all the streamline segments that pass through the voxel. We do this once per TCK file as a preprocessing step. After preprocessing, we simply sort the voxels back to front every frame and render them accordingly. This way we achieve semi-accurate transparency with good performance without any complicated render pipelines. While the idea behind this algorithm is quite simple; however, generating the voxel meshes has some complications that need to be handled properly.

One issue is that using only the points inside the voxel will result in broken streamlines. As a streamline passing through multiple voxels will have a missing line connecting the individual line segments together, we resolve this by adding the next point in the streamline to each line segment that does not hold the final streamline point. This ensures that each line segment connects to the next line segment.

A different issue occurs when a streamline passes through multiple voxels. Initially, we simply rendered line segments based on the sorted points, but it caused incorrect lines to be rendered. Let's say a line has points 1,2,5,6 in voxel A and points 3,4 in voxel B. Rendering lines based on sorted points will cause an incorrect line to be rendered between points 2 and 5. To solve this issue, we added an additional step that checks if the line segment points are always increasing by one. If they are not, we either add a new point if the distance is exactly 2 (e.g., adding point 8 between 7 and 9), or we split the line segment into multiple line segments if the distance is larger than 1.

Result

The result is a relatively accurate transparent rendering of streamlines as seen in Fig. 4. The major issue is that the render order within voxels is not safeguarded. This causes visual discrepancies when a high (> 0.9) alpha value is used. In practice VoxelBased is used for transparent rendering. With transparency the render order within voxels quickly

become irrelevant by the properties of transparency and blending. We believe this is one of the best ways to render transparent objects for the given context.

Algorithm 1 Voxel mesh generation algorithm

```
for each streamline in dataset do
    for each point in streamline do
        add point to the voxel the point is within
    end for
end for

for each voxel do
    linesegments ← voxel points grouped by line
    for each linesegment in linesegments do
        points ← linesegment points sorted by ascending
        for k = 0 to linesegment point count - 1 do
            curPointIndex ← line relative point index of k
            nextPointIndex ← line relative point index of k+1
            if curPointIndex+1 is not nextPointIndex then
                if curPointIndex+2 is nextPointIndex then
                    add point curPointIndex+2 in
                    between points k and k+1.
                else
                    Replace linesegment entry in linesegment
                    with two new linesegments.
                    One with points 0..k and other
                    with points k+1..linesegment length
                end if
            end if
        end for
    end for
end for
for each segment in linesegments do
    if segment does not contains the final streamline point then
        nextPoint ← max line segment point index + 1
        Append nextPoint to segment
    end if
end for

vertices ← new vertex list
indicies ← new integer list
for each linesegment in linesegments do
    for each point in linesegment do
        Add point to vertices
    end for
end for

for each linesegment in linesegments do
    for k = 0 to linesegment point count - 1 do
        Add k to indicies
        Add k+1 to indicies
    end for
end for
```

7 Results

Visual comparison between different applications with screenshots of same camera. Performance comparison using the hardware described in Appendix C.

8 Discussion

We have met our initial objectives of rendering TCK data in *real-time* and providing interaction tools. We believe our software is aesthetically pleasing and it provides plenty of options for the user to configure the visualisation to their desires. Our filtering and colouring tools, while simple, are easy to use and very performant. Our user interface is intuitive and simple, making the exploration of the MRI tractography data more accessible. We have received great feedback from both our colleges and professor.

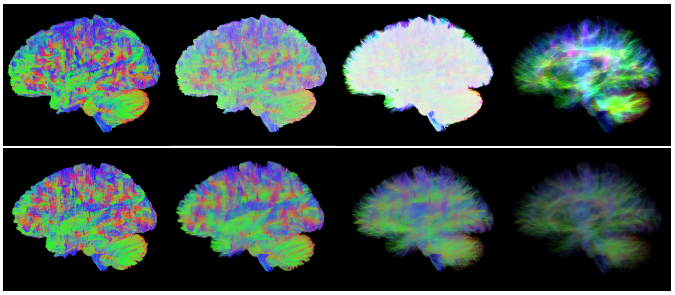

Fig. 4. Visual comparison between different transparency values of the *sift_1mio.tck* B.A.T.M.A.N. [7] file. On the top is MRView [8] rendering with lines render method. Bottom is our novel VoxelBased render method. *Note: MRview does not display the alpha value, so no direct comparison could be made with same transparency values.*

Table 1. TCK File Information

TCK File	Lines	Points
smallerSIFT_200k.tck	200k	4,671,775
sift_1mio.tck	1 million	23,290,802

Our solution does fall short in performance. In most cases, rendering takes longer than the existing solutions using similar render methods (MRView Lines vs NeuroTrace Primitives). Additionally, loading data takes a more time than other software. This is caused by the voxelization service which is necessary to speed up filtering tools and VoxelBased rendering and our focus on adding new functionality with limited the time we spent on optimization. However, we did optimize the intersection service. We managed to make it work in real-time with large datasets (sift_1mio). We are confident that if we spent more time on render performance we could improved it considerably.

We made the conscious decision to focus on novel render techniques and exploration instead of existing solutions for rendering massive amounts of streamlines. Using more existing research would have improved performance, but we focused on our own methods and exploration instead. This is not necessarily a negative, as it allowed us to try creative techniques, but it has evidently impacted the final product in some negative ways mainly having worse render performance than existing visualization software.

9 Future Work

Our main area of focus for future work would be visual quality and performance. There are many optimization techniques that can be implemented. Most functionality have been implemented at a bare bones level and can be expanded upon. For instance the intersection service currently only has spheres, other shapes or inclusion/minus combinations could be implemented.

There is no shortage of additional features can be implemented. Examples include raymarched rendering using SDFs, lighting with shadows, ambient occlusion – as described in [3] –, anti-aliasing, subsurface scattering (considering the visualisation subject matter), and volumetric MRI scan rendering for improved contextual awareness. Some of these are partially finished or at least work-in-progress. Unfor-

Table 2. Render performance compared to MRtrix3 [8] MRView visualizer

TCK File	Renderer	Load time	Render time
smallerSIFT_200k	MRView Lines	$<$ 1 sec	8.5 ms
smallerSIFT_200k	Primitive (ours)	1.8 sec	7.3 ms
smallerSIFT_200k	VoxelBased (ours)	3.4 sec	10.6 ms
sift_1mio	MRView Lines	1.4 sec	15.6 ms
sift_1mio	Primitive (ours)	9.8 sec	21.3 ms
sift_1mio	VoxelBased (ours)	14.6 sec	47.6 ms

tunately, we could not implement these given the time constraint and workload of other courses.

10 Conclusion

In conclusion, NeuroTrace is a promising solution that has achieved what it was designed for, providing both impressive visual results and interaction tools. There is room for improvement regarding performance and ample opportunity for expansion.

References

[1] M. Chamberland, M. Bernier, D. Fortin, K. Whittinstall, and M. Descoteaux. 3d interactive tractography-informed resting-state fmri connectivity. *Frontiers in Neuroscience*, 9:275, 2015. doi: 10.3389/fnins.2015.00275

[2] M. Chamberland, K. Whittinstall, D. Fortin, D. Mathieu, and M. Descoteaux. Real-time multi-peak tractography for instantaneous connectivity display. *Frontiers in Neuroinformatics*, 8(59):1–15, 2014. doi: 10.3389/fninf.2014.00059

[3] S. Eichelbaum, M. Hlawitschka, and G. Scheuermann. Lineao—improved three-dimensional line rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):433–445, 2013. doi: 10.1109/TVCG.2012.142

[4] Khronos. Shader storage buffer object.

[5] L. L. Nesi, C. Rorden, and B. C. Munsell. A simple and efficient cylinder imposter approach to visualize dti fiber tracts. In G. Wu, P. Laurienti, L. Bonilha, and B. C. Munsell, eds., *Connectomics in NeuroImaging*, pp. 89–97. Springer International Publishing, Cham, 2017.

[6] J. Schertzer, C. Mercier, S. Rousseau, and T. Boubekeur. Fiblets for real-time rendering of massive brain tractograms. *Computer Graphics Forum (Proc. EUROGRAPHICS 2022)*, 41(2):447–460, 2022.

[7] M. Tahedl. B.a.t.m.a.n. - basic and advanced tractography with mrtrix for all neurophiles, 2017.

[8] J.-D. Tournier, R. Smith, D. Raffelt, R. Tabbara, T. Dhollander, M. Pietsch, D. Christiaens, B. Jeurissen, C.-H. Yeh, and A. Connelly. Mrtrix3: A fast, flexible and open software framework for medical image processing and visualisation. *NeuroImage*, 202:116137, 2019. doi: 10.1016/j.neuroimage.2019.116137

[9] R. Wang, V. J. Wedeen, TrackVis.org, and M. G. H. Martinos Center for Biomedical Imaging. Trackvis: visualization and analysis of diffusion mri data. *Proceedings of the International Society for Magnetic Resonance in Medicine*, 15:3720, 2007.

A Acknowledgements

We would like to thank the school for this opportunity and our professor for their support and guidance during this project. Both of us have gained valuable knowledge and experience throughout this assignment. We are also grateful to other students for their feedback and criticism.

B Contributions

We have worked together before and are aware of each others abilities. Therefore we divided the project in two main goals in the first week. Mestiez was responsible for the OpenGL context and general rendering, while Besm was responsible for UI and interaction. After the fourth week we changed responsibilities to both focus on more of the rendering work. This division was not absolute as our contributions overlapped and we helped each other where needed. We each worked about 16 hours a week on this project.

Besm's contributions

- TCK file loader (10%)
- Dear ImGui integration (10%)
- Most of the user interface (10%)
- Camera options such as sync and lock (10%)
- General code architecture (service based with visualization services and global services) (10%)
- Voxel service (5%)
- VoxelBased render method (25%)
- Intersection functionality (10%)
- Bounds functionality (5%)
- Colormapping options (5%)

Mestiez' contributions

- OpenGL context including window, vertex attributes, meshes (20%)
- Camera movement (5%)
- Shaders, materials, textures, rendertextures (20%)
- Primitive render method (10%)
- Tube render method (25%)
- MatCap (5%)
- Raymarching rendermethod (unfinished) (15%)

C Hardware

All performance evaluations were performed on a laptop with the following system:

Evaluation system specifications	
CPU	Intel Core i7-9750H CPU @ 2.60GHz
GPU	GeForce GTX 1650
Display	1920 × 1080
Memory	16 GB, 2400 MHz

GLOSSARY

real-time defined by a consistent update rate of at least 30Hz. 1, 2, 4, 6

streamline a set of connected points provided by an input file. 1, 2, 3, 6