EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), Number 2

Fiblets for Real-Time Rendering of Massive Brain Tractograms

Jérémie Schertzer[1], Corentin Mercier[1], Sylvain Rousseau[1] and Tamy Boubekeur[2]

[1]LTCI, Télécom Paris, Institut Polytechnique de Paris
[2]Adobe Research

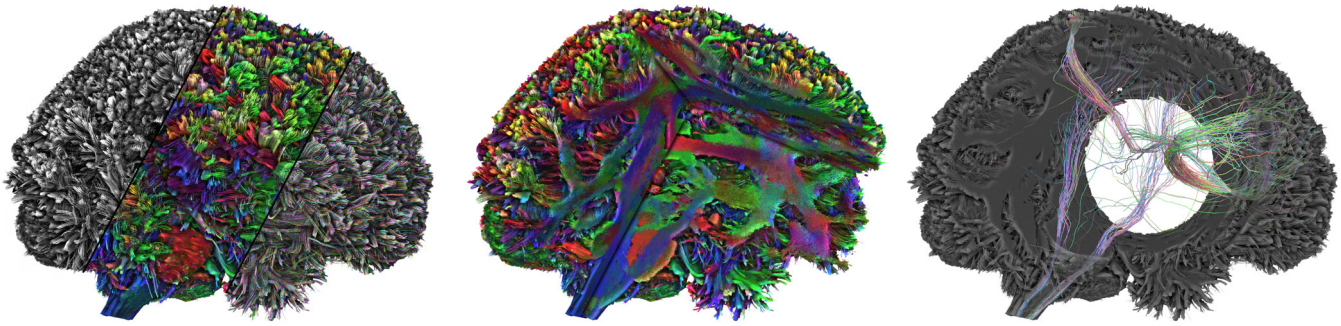

Figure 1: *Real-time rendering (40ms/frame) of a 64GB brain tractogram containing 3 million of individual fibers (5.37 billion segments) compressed down to 7GB. From left to right, the figure shows the rendering using different shadings (solid color, fiber orientation, per-fiber color), the ability for our pipeline to perform interactions with occlusion meshes (here with a cube mesh), and the per-fiber interaction capabilities with a selection (combined here with an occlusion mesh).*

Abstract

We present a method to render massive brain tractograms in real time. Tractograms model the white matter architecture of the human brain using millions of 3D polylines (fibers), summing up to billions of segments. They are used by neurosurgeons before surgery as well as by researchers to better understand the brain. A typical raw dataset for a single brain represents dozens of gigabytes of data, preventing their interactive rendering. We address this challenge with a new GPU mesh shader pipeline based on a decomposition of the fiber set into compressed local representations that we call fiblets. Their spatial coherence is used at runtime to efficiently cull hidden geometry at the task shader stage while synthesizing the visible ones as polyline meshlets in a warp-scale parallel fashion at the mesh shader stage. As a result, our pipeline can feed a standard deferred shading engine to visualize the mesostructures of the brain with various classical rendering techniques, as well as simple interaction primitives. We demonstrate that our algorithm provides real-time framerates on very large tractograms that were out of reach for previous methods while offering a fiber-level granularity in both rendering and interaction.

CCS Concepts

• ***Hardware*** → ***GPUs and Graphics Hardware;*** • ***Rendering*** → ***Real-Time Rendering;*** • ***Visualization*** → ***Medical Imaging;***

1. Introduction

Brain tractograms are medical imaging data obtained using diffusion magnetic resonance imaging (dMRI). They represent the path of the axons that connect neurons through the white matter. Tractograms are used by neurosurgeons for operation planning as well as to predict the possible post-operation consequences. They are also used by researchers to better understand the brain. Brain tractograms – sometimes referred to as *fiber tracking datasets* – are typically composed of millions of 3D polylines, called streamlines or fibers [TML11]. A dMRI gives the diffusion flux of the water molecules in the brain with a resolution that is typically between 1.25 and 2 mm^3 sampled as a regular 3D grid. The fibers can be obtained from those fluxes using the FACT (fiber assignment by continuous tracking)-algorithms [MCCV99; MV02; TML11]. Seeds are randomly placed in the white matter and are propagated following the gradient in the two directions. An explanation of FACT algorithms is provided in the additional materials. This propagation can be done with either deterministic or probabilistic algorithms.

Depending on applications, two strategies coexist to compute and display tractograms. On-the-fly fiber generation from the

Figure 2: *Mesh/Task Shader pipeline*

lightweight dMRI voxel grid [CWF*14] and the use of a precomputed file indexing fibers data. At the cost of consequent file size, visualization from precomputed fiber files allows the visualization of full tractograms with a sufficient number of fibers. Besides, the result is invariant and may display precomputed attributes per fiber that on-the-fly generation would not be able to compute. In this paper, we target the visualization of offline-computed tractograms, following most tractograms visualization tools. Still, visualization from real-time generation is also likely to benefit from our fiblet formulation and pipeline.

To generate a complete high-definition tractogram, millions of fibers are usually tracked. Each fiber typically contains hundreds to thousands of ordered vertices and the whole tractogram can contain several billion segments. Consequently, most processing, visualization, and interaction software are not able to handle such large tractograms. They often sub-sample the fibers, introducing approximation and loss of information. In this paper, we introduce a pipeline able to visualize and interact in real-time with tractograms, without any sub-sampling.

Nvidia's Turing architecture introduced the Mesh Shading Pipeline [Kub18; Mou18] which starts with an optional task shader allowing for high control on the workflow and invocations to the mesh shader (Fig. 2). The mesh shader is responsible for computing vertices' location and attributes, as well as a primitive index buffer. Each mesh shader invocation processes a few dozen vertices, generating a *meshlet*. The two stages have a compute shaders-like syntax. A major consequence is that each task or mesh shader invocation happens at the scale of a warp (32 threads), so the computation of meshlets should be collaborative within the warp for good performances as described in experiments [Kra18; Kub20]. The mesh shader pipeline has only received little attention so far even though it offers a powerful framework to handle emerging data representation such as brain tractograms. Beyond level-of-detail control and culling strategies, this new pipeline has been recently used for custom tessellation [Rah19] and skeletal animation optimization [Tor19]. We propose an original use case with a new rendering algorithm exploiting the task shader to alleviate the processing workload down the pipeline.

Contributions. Our main contributions are:

- a new representation model, called *Fiblets*, designed for efficient GPU decompression of massive tractogram data
- a fine-grained, parallel scalable fiber synthesis algorithm exploiting the new *mesh shading* pipeline to achieve from one to two orders of magnitude rendering speed-up compared to previous methods.

We also provide all the algorithms and implementation details of our method.

2. Previous Work

2.1. Medical imaging visualization

A few general medical data visualization tools provide tractogram rendering capabilities, such as *3DSlicer* [PLSK06], *OpenWalnut* [EHS13], or *Paraview* [AGL05; Aya15] but they are all extremely limited in terms of scalability and cannot render large tractograms. More specialized tools exist [WBSW07; GBA*14], with better performances for small tractograms. However, they suffer from severe limitations when it comes to displaying large tractograms at full resolution, even when the raw data fit into the VRAM. For instance, *Trackvis* [WBSW07], only displays a small percentage of the fibers.

Fiber Navigator [CWF*14; CBF*15] does not only rely on a precomputed tractogram but can also generate fibers on the fly depending on a user-selected region of interest. These software packages enable simple rendering using lines, as well as transparency [CBF*15]. Some geometric primitives such as cylinders [WBSW07; PLSK06; EHS13] can also be used at the cost of lower rendering performances, with different coloring options, including solid colors or coloring following the local orientation of the fibers [PLSK06; WBSW07; EHS13; GBA*14; CBF*15]. We compare our work to all of these tools.

Alternative methods use approximations to render large tractograms, such as clustering [GBC*12; KAC15; DÇ15], level of details with cylinders [PFK07] or decimation relying on a linearization process [RHD17]. The latter can have a negative impact on frameworks relying on point-based interaction and processing [SMAS13].

Other methods target data understanding, such as *LineAO* [EHS12] which introduces ambient occlusion for lines, while not entirely darkening far away fibers that could still be visible, providing a better perception of the depth, without losing too many details in occluded regions. *Illuminated lines* [MPSS05] enriches lines with smooth normals depending on the camera position, allowing shading using standard reflectance models. *Depth halo* [EBRI09] improves depth perception by creating halos around fibers that are visually close to each other. All of these visualization techniques can be used with our rendering pipeline as post processes. In particular, we use *Illuminated lines* [MPSS05] as well as screen space ambient occlusion in our final renderings. More details on algorithms that help to better understand tractograms can be found in the recent survey by Isenberg et al. [Ise15].

2.2. Computer graphics

Brain tractograms are represented using 3D polylines. Such data are commonly used in real-time computer graphics to model hairs or fur [WBK*07; JCLR19]. Fur typically contains a few hundred thousand strands with 3 to 4 vertices [JCLR19] while tractograms can contain several millions of fibers with up to a thousand vertices per fiber leading to billions of segments. Fabric rendering with fiber-level details uses detailed fibers [ZLB16; WY17] but relies on procedural generation directly on the GPU as well as on Level of Details (LoDs). *Fiber Navigator* [CBF*15; CBF*15] can deal, on the fly, with parts of a large tractogram but not with an entire model.

Brain tractograms LoDs methods are based on medical considerations that require specific graphics adaptations [GBC*12; MGR*18; DMP*19] but cannot cope with topological and visual changes that a dynamic cut would introduce for instance.

Massive 3D data rendering methods such as *QSplat* [RL00] or *PCC* [SKW21] display large-scale 3D scans by transforming meshes into point primitives organized in a hierarchy. In our context, however, the fiber connectivity is critical and point-sampling is not an option. Similarly, methods such as *DUODECIM* [KSW05] or *Sequential Point Trees* [DVS03] would not preserve the individual fibers. *Adaptive TetraPuzzles* [CGG*04] and *Far Voxels* [GM05] run out-of-core and can display large meshes but they consume respectively 32 and 70 MB per million vertices. Raw polylines are encoded with 12 MB per million vertices, and compression techniques reduce this size even further, under 1.3 MB per million vertices, which led us to adopt a more flexible in-core approach. Modern billboard techniques [MB20] are used to render massive numbers of small objects sharing a similar shape. In contrast, each fiber of a brain tractogram follows a unique data-driven pattern.

Occlusion culling has been used in the case of massive data rendering, for instance in the case of particles [IRR*22] using a depth confidence map. However, in our context, each fiber is a set of ordered points following a unique path, and it can be hard to keep track of the connectivity if we were trying to represent each segment as a particle. In terms of occlusion, a lot of techniques use hierarchies [BWPP04; LJSL21], combined with temporal coherence [MBJ*15], or a mix of temporal and spatial coherence [MBW08]. However, computing a hierarchy on our data is costly in memory. We build upon some elements of these concepts and use a form of occlusion technique in our rendering pipeline with a reprojection of the coverage buffer, using camera warping techniques [LKE18] without the need for a hierarchy over the scene. More recently, real-time visualization of massive geometries was introduced through *Nanite* [KSW21]. However, it requires triangles and models that can be simplified using edge decimation and is not suitable for datasets composed of lines such as brain tractograms.

Culling of primitives is also a critical task for massive data rendering, Pantazopoulos et al. [PT02] wrote a comprehensive survey of these techniques. In our context, fibers cross the entire brain and every single one has a unique shape, complicating their culling. However, we can rely on our fiblet-based layout to cull geometry. We use the temporal coherence of the visibility by applying morphological operators on the depth buffer to cull fiblets, taking inspiration from existing techniques in 2.5D [DDS03].

Voxel-based methods that exploit Sparse Voxel Octrees such as *GigaVoxels* [Cra11], *SVDAGs* [KSA13] and *SSVDAGs* [VMG17] are designed mostly to exploit the sparsity of surfaces in 3D space, whereas our tractogram data is densely packed. Moreover, a voxelized solution would make per-fiber interaction nearly impossible. State-of-the-art methods also exist for 3D lines/curves rendering. Kanzler et al. [KRW19] proposed a voxel-based rendering method that uses ray-casting and can render global illumination on large datasets. The curves are discretized using their intersection points with the surface of the voxels that are then quantized. This discretization process can introduce errors beyond the dMRI acquisition error and the LoD that is computed afterward would again lead to the loss of per-fiber control. This method could however be a good perspective to build a LoD over a brain tractogram. A recent survey [KNM*20] explains the different methods to render transparent 3D lines.

We can also relate our work to flow visualization that typically needs to display a large number of curves. Such methods exploit transparency, or the selection of relevant lines [Gün20]. While some flow visualization approaches [PWK20] are adapted to visualize the dMRI data, they do not fit the brain tractograms constraints. Streamline methods [LS07] generate representative streamlines, but any approximation or representation that is not medically justified should be avoided in our context.

2.3. Tractogram compression

The main issue when designing any algorithm working on brain tractograms is their size. They require high-precision encoding and are represented using millions of fibers encoded as rather long 3D polylines [TML11], typically representing dozens of gigabytes. This makes them hard to process, store, share, or visualize, especially in the context of clinical use [RHD17; MRG*20]. In practice, they are stored as a set of unorganized 3D polylines for which computer graphics can provide efficient compression techniques.

In our application scenario – visualization – we take interest in GPU-friendly compression methods that are fast enough to decompress the tractogram on the fly on the GPU. General compression algorithms sush as ZFP for floating-point data [Lin14] or *Draco* [Goo17] for general 3D meshes and point clouds can be used. However, as shown in *TRAKO* [HFZ*20], the versatility of such algorithms makes them less efficient than data-specific techniques, both in terms of speed and compression factor. These are critical to achieve fast GPU decompression and render massive datasets. *ZFib* [PJHD15] is a compression algorithm that makes use of linear approximation, quantization, and dictionary-based compression. This method can efficiently reduce the size of the data but happens to be too slow for our application scenario, while not scaling to large datasets.

QFib [MRG*20] achieves a high compression ratio with a fast on the fly compression and decompression technique thanks to a simple representation of the fiber and a quantization process based on the construction of the fibers from the dMRI data. This algorithm appears to fit our application scenario, as each fiber can be individually compressed and decompressed, making the algorithm embarrassingly parallel and enabling efficient GPU decompression. It scales to large datasets as an out-of-core version is available, has an error at least an order of magnitude under the precision of the dMRI, especially when the distance between the points of the fibers is small, which is the case in large datasets. We base our primitive generation process on this method.

3. Compressed Brain tractogram visualization

3.1. Overview

The input of our method is a brain tractogram, that typically contains dozen millions of 3D polylines. This tractogram is compressed and stored in VRAM using the method described in Sec.3.2

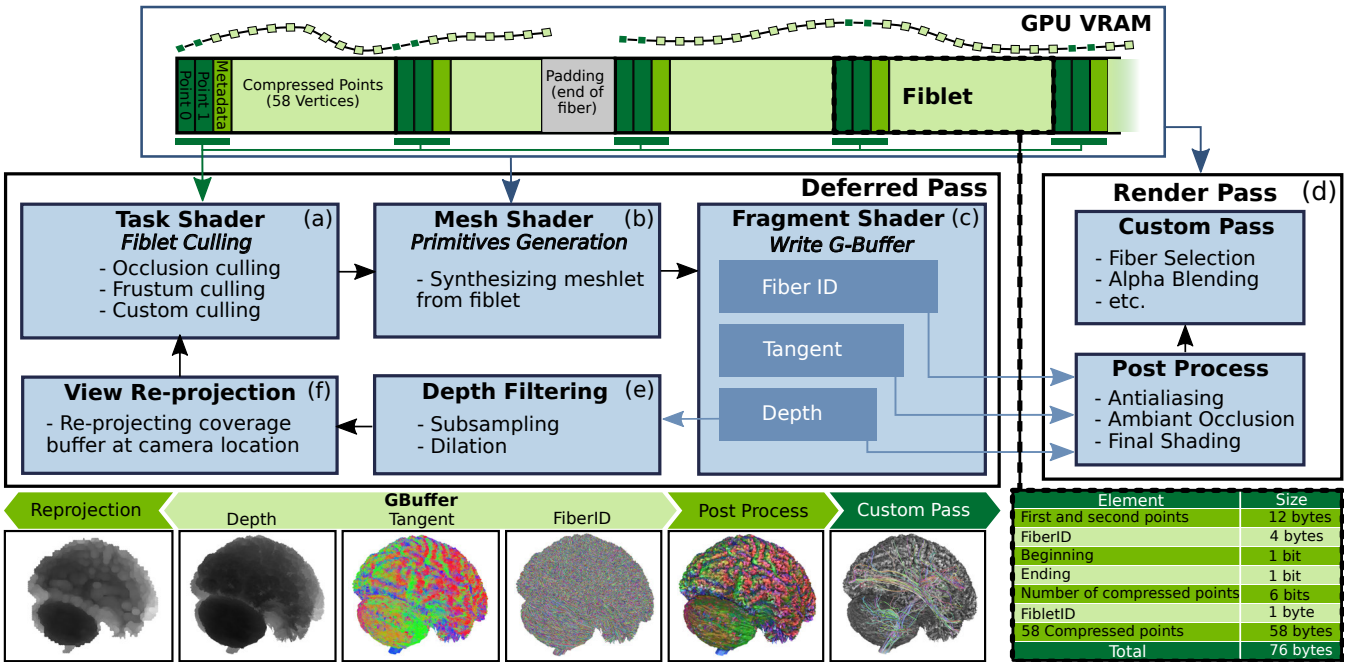

Figure 3: *In our pipeline, the compressed representation (fiblets) is stored directly in VRAM. (a) First, the Task Shader reads the metadata and the first two points of every fiblet to perform the culling tests. The occlusion test relies on temporal coherence with a texture lookup to a coverage buffer. (b) The surviving fiblets pass through the mesh shader that retrieves from it a polyline meshlet using a warp-scale parallel algorithm. (c) The fragment shader writes the z-buffer and the G-buffer. (d) The rendering pass is shaded from the G-buffer. A custom pass can be added to include other effects such as user-selected fibers highlighting. (e) The z-buffer is subsampled and undergoes a morphological dilation with a spherical kernel to create a coverage buffer. (f) That coverage buffer is re-projected to the camera of the next frame.*

with an adaptation of *QFib* [MRG*20]. This adaptation consists of dividing fibers into fixed-size fiblets, for better parallelism and spatial coherence. We then propose the pipeline described in Fig. 3. The key idea is to discard occluded fiblets while decompressing and rendering the relevant ones. Therefore, we fully take advantage of the mesh GPU pipeline, using the task shader to discard fiblets failing the conditions described in Sec.3.4. The remaining fiblets are then decompressed to 3D polylines within the mesh shader stage as described in Sec.3.3. The shaded fragments supply the G-buffer used for the rendering pass while the z-buffer serves as an input to define the fiblet rejection criteria for the next frame. We also show in Sec.5 that our deferred pipeline supports visualization tools as well as various shading post processes.

3.2. Fiblet Compression

3.2.1. *QFib* algorithm

Our compressed fiblet scheme is an evolution of *QFib* [MRG*20] improving its parallel efficiency for GPU. *QFib* compresses each fiber of a tractogram individually, taking into account both the anatomical properties and the constraints known from the FACT algorithms used to build the fibers. The first property is that each segment of a given fiber measures the same length δ, making possible to express any point using:

$$\mathbf{P_i} = \mathbf{P_{i-1}} + \delta \cdot \frac{\mathbf{P_i} - \mathbf{P_{i-1}}}{||\mathbf{P_i} - \mathbf{P_{i-1}}||} \quad (1)$$

With this representation, a fiber can be stored using the first two points, δ, and a set of unit vectors. The second property – the maximal angle α between consecutive segments – ensures that the representation space of a given unit vector according to the previous one is limited to a small spherical cap. A uniform mapping [RB20] is used to transform the unit vectors from those spherical caps to the surface of the whole unit sphere to improve the precision of the unit vector quantization method.

3.2.2. Adaptation of *Qfib* to the GPU scenario

We modify the compression scheme by splitting the fibers into fiblets encoding a fixed number of points. Each fiblet implicitly stores the compressed coordinates of the points using *Qfib* uniform mapping and octahedral quantization [MSS*10]. The memory structure of a fiblet is described in Fig.3. Prior to any compression, all vertices from the raw tractogram are translated and scaled so that every point lies in $[0;1]^3$. The first two points of each fiblet have their spatial coordinates quantized with 2 bytes on $[0;1]$, hence counting 12 bytes in total. The subsequent quantization is not harmful since for a $15cm$ brain, its typical length is $2\mu m$, which is three orders of magnitude below the typical dMRI resolution of $1.25mm^3$. Each fiblet is distinctively referenced using a unique *FiberID* and a local *FibletID* within that fiber. As every fiblet contains a fixed number of points – 60 in our implementation – it is possible to compute a unique identifier for any point of any fiber in the tractogram, keeping the possibility of visual effects

at the granularity of the fiber, the fiblet, or even the vertex despite dealing with massive data. Each fiblet contains a beginning, an ending flag, and the 6 bits remaining store the *number of compressed points*. This parameter is essential for fiblets that are ending a fiber as they encode less than 60 points.

We use the uniform mapping to map the spherical cap of angle α to a hemisphere while *Qfib* maps to a sphere. This choice offers a slightly better precision but is mainly motivated by performance. Since the UV unfold of a half-octahedron is a square, a direction is quantized using the UV coordinates of its projection on the half-octahedron. An 8 bits half-octahedral quantization (4 bits per UV coordinate) of the mapped directions is precise enough compared to the dMRI precision (see Sec.4) to encode a direction.

There is a trade-off in the choice of the number of encoded points per fiblet. GPU hardware specification stipulates that structures must be 4 bytes aligned, constraining the number of points in a fiblet to be a multiple of 4 (1 point = 1 byte with our representation) to avoid unnecessary padding. Using 64 points per fiblet requires a 65^{th} point to ensure continuity between fiblets. From a hardware perspective, exceeding the limit of 64 vertices severely slows down the mesh shader, making 60 points per fiblet the maximum value possible. Lowering the number of points per fiblet (by multiples of 4) results in a worse compression ratio, less warp occupancy, more fiblets to test within the task shader, but higher culling rejection. All things considered, the trade-off in memory consumption makes 60 points our optimal choice.

Qfib encodes its quantized values using an octahedron whose orientation is implicitly defined at each point by the previous decompressed vector, thus forcing a sequential decompression scheme. To make it possible to retrieve the fiblet vertices using concurrently the full 32 threads of a warp, (see Sec.3.3), we propagate a local frame from point to point, storing the quantized coefficients of the directions expressed in that local frame. This change of frame is a critical difference with *Qfib* and part of our contribution. Detailed mapping and quantization formulas can be found in Appendix A.

3.3. Meshlet synthesis from fiblet

At run time, the GPU is responsible for retrieving fiblets into a polyline meshlet (later drawn as GL_LINE_STRIPS) representing the part of the fiber encoded by the processed fiblet. Continuity between meshlets is made possible since the first point of the next fiblet is accessed by simply unpacking the 6 first bytes, which is a coherent memory access relatively to the processed fiblet.

3.3.1. Sequential fiblet synthesis

After unpacking the first two points of the fiblet, decompressing a fiblet consists in retrieving the unmapped direction and updating the current point with a translation of the constant length δ in that direction. Then, the shader is responsible for generating the 60 points of the fiblet in addition to the first of the next fiblet for continuity, as well as the 60 corresponding primitives. If the fiblet is ending a fiber, only the required number of vertices and primitives is generated. The decompression formula can be found in Appendix B of the additional materials.

3.3.2. Parallel fiblet synthesis

Parallel prefix sums [HSO07; MG16; MYB16] are commonly used in decompression schemes. The key idea for the parallel fiblets synthesis is to design a parallel prefix sum approach specifically for spatial transforms, which turns to be a parallel prefix transform matrix multiplication. This section describes this spatial adaptation for fiblets. The notations defined below are illustrated in Fig. 4.

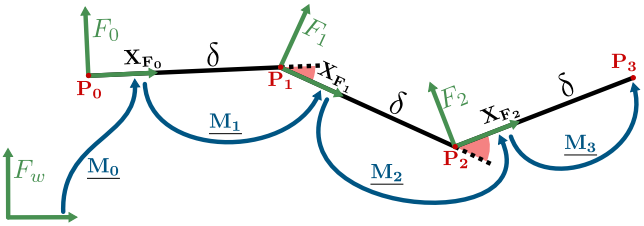

Figure 4: *Notations for a fiblet encoding 4 points. The red angles illustrate the variation of direction that is directly mapped and quantized on 8 bits each. By construction, α is the maximal admissible value for these angles.*

In the world frame F_w, let $\mathbf{P_i}$ be the coordinates of the point i, let F_i be the local frame of the point i. Let $\underline{\mathbf{M_i}}$ be the 4×4 transformation matrix morphing F_{i-1} to F_i, and $\underline{\mathbf{M_0}}$ the transformation matrix from the world frame F_w to F_0.

The rotational submatrix of M_0 transforms the unit vector $\mathbf{X_{F_w}}$ to the unit vector $\mathbf{X_{F_0}} = \frac{\mathbf{P_1}-\mathbf{P_0}}{||\mathbf{P_1}-\mathbf{P_0}||}$. However the rotational part of $\underline{\mathbf{M_i}}$ describes the local rotation from the vector $\mathbf{X} = (1,0,0)$ to the vector $\mathbf{XLocal_{F_i}} = \frac{\mathbf{P_i}-\mathbf{P_{i-1}}}{||\mathbf{P_i}-\mathbf{P_{i-1}}||}$, expressed in the F_{i-1} coordinate system. Our compression scheme encodes the mapped and quantized vectors $\mathbf{XLocal_{F_i}}$, knowing that the angle between $\mathbf{XLocal_{F_i}}$ and $\mathbf{X}$ is smaller than α. For $\underline{\mathbf{M_0}}$, the rotation submatrix is obtained thanks to the first two points of the fiblet while the rotation submatrices of $\underline{\mathbf{M_i}}$ are directly derived by decompressing the directions $\mathbf{XLocal_{F_i}}$. Note that the rotation submatrix relative to the last point of a fiblet is discarded since the frames are not propagated beyond.

The remaining constraint is that the UV coordinates of the half-octahedron are stored for a given rotation of that half-octahedron around $\mathbf{X}$. To keep track of that rotation, a unique 3D frame F is built in a simple and deterministic way knowing only $\mathbf{F_{orward}}$, by introducing an additional constant vector $\mathbf{V}$:

$$\mathbf{U_p} = \frac{\mathbf{F_{orward}} \times \mathbf{V}}{||\mathbf{F_{orward}} \times \mathbf{V}||} \qquad \mathbf{L_{eft}} = \mathbf{U_p} \times \mathbf{F_{orward}} \tag{2}$$

In our case the degenerate situation where $\mathbf{F_{orward}}$ and $\mathbf{V}$ are colinear is avoided since $\mathbf{F_{orward}}$ is quantized on only 256 values so $\mathbf{V}$ can easily be chosen out of that set. Note that the algorithm must use the same vector $\mathbf{V}$ to propagate the local frames at compression time and decompression time.

The last column of $\underline{\mathbf{M_0}}$ defines the position of the first point in F_w coordinate system, which is simply the translation needed to reach F_0 from F_w. The last column of $\underline{\mathbf{M_i}}$ describes the translation from F_{i-1} to F_i in the F_{i-1} coordinate system which is simply $\delta \cdot \mathbf{X}$ by

construction. We finally define the 4×4 matrix $\underline{\mathbf{W_i}}$ as following:

$$\underline{\mathbf{W_i}} = \prod_{k=0}^{i} \underline{\mathbf{M_k}} \tag{3}$$

The coordinates of $\mathbf{P_i}$ in F_w boils simply down to the translation submatrix of $\underline{\mathbf{W_i}}$.

From equation (3) it appears that a parallel prefix algorithm is suitable to retrieve all $\underline{\mathbf{W_i}}$. For a fiblet storing N points, an N elements array of 4×4 matrices is stored within the shared memory of the invoked warp. The matrix list is initialized with the $\underline{\mathbf{M_i}}$ which are retrieved from the decompressed fiblet data. The parallel prefix algorithm ensures that the list of the $\underline{\mathbf{W_i}}$ can be concurrently computed in $\lceil log_2(N) \rceil$ passes.

A practical illustration of that parallel prefix spatial transform can be found in additional material. Each pass requires $\lceil N/2 \rceil$ matrices computation therefore at least the same number of available threads. Since each fiblet encodes 60 points, the 32 threads are enough to compute the parallel prefix multiplications. The SIMD architecture forces some thread idling because the instructions to compute M_0 (and M_0 of the next fiblet, required to link the mehslets) differs from the rest of the M_i. Once all the $\underline{\mathbf{W_i}}$ are calculated, each thread writes to the mesh shader buffers two vertices and four primitives indices that build the meshlet. The first thread also writes the number of primitives to be consumed by the rasterizer according to the fiblet metadata.

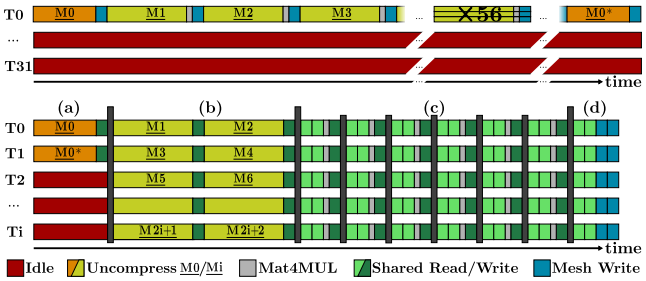

Figure 5: *Simplified chronograms of a mesh shader invocation on a warp. On top, only one thread is synthesizing the meshlet while the remaining 31 are idling. At the bottom, the parallel scheme takes advantage of shared memory (barriers in grey) to cooperatively process the fiblet.* $\underline{\mathbf{M_0}}*$ *represents the first matrix of the following fiblet, required to keep continuity along the fiber. (a) The first matrix is decompressed (b) then each thread is responsible for calculating two matrices so that (c) the 6 passes matrix array reduction can be computed in order to (d) write the meshlet vertices.*

Figure 5 compares the chronograms of the sequential and parallel prefix algorithms. For performance comparison see Sec.4. In our implementation, discarding constant coefficients of the transformation matrices to minimize the shared memory bandwidth and also storing all coefficients in the same memory bank per matrix resulted in great performance improvement compared to a naive implementation of the parallel prefix multiplication.

3.4. Fiblet pipeline

So far, we greatly reduced the GPU bandwidth by expressing vertices in a compressed fiblet structure and ensured an efficient way to concurrently build the meshlet from the fiblet within the invoked warp. This section describes our fiblet culling technique, inspired from existing hierarchical z-buffer and temporal coherence techniques, as mentioned in Sec.2.2. Our contribution stems from a coverage buffer built from morphological dilation on the z-buffer. For discarding fiblets, it results in finer culling borders compared to a direct lookup in a hierarchical z-buffer.

3.4.1. Fiblet culling

By construction, a simple bounding proxy for a fiblet can be expressed as a sphere S_{fiblet} centered on the first point of that fiblet with a radius $R_S = N \times \delta$. By nature, testing if a sphere intersects a volume corresponds to testing if the center of that sphere is inside the volume dilated with that spherical kernel. We exploit this property to reduce the fiblet bounding test to a much faster point test against a coverage buffer built by dilating the z-buffer. Figure 6 gives an insight of our culling method that, once the coverage buffer is generated, simplifies the task shader fiblet culling test to a single depth test requiring one texture fetch. In practice, since all meshlets are connected to each other unless the fiber reaches an end, it is profitable to also test the first point from the next fiblet against the coverage buffer. Fiblets and texture coherence reduce the cost of that test. If the depth of at least one point is beyond the coverage buffer, the fiblet is then discarded.

That fiblet culling test is performed within the task shader (Fig. 2). The performance gain is significant since each saved mesh shader invocation decreases the memory bandwidth, the number of shaded fragments, and saves that warp for another invocation.

3.4.2. Building the coverage buffer

In our implementation, the coverage buffer is derived from a subsampled z-buffer. Subsampling serves two purposes: reducing the number of sprites and texels to process while increasing the texture cache coherence when the task shader massively fetches from it during the next frame. Four levels of mipmaps are computed using a *max* filter to compute the depth of a texel from the 2×2 cell of the previous level. As a consequence, the subsampled z-buffer is greater or equal to its full-scale value in each direction, preventing faulty fiblet culling artifacts.

Dilation is computed by drawing a hemisphere centered in the 3D coordinate of each texel with the z-test set to *greater* function. In practice, the visibility of those spheres is equal to the visibility of a disc sprite of the same radius. Depths of the ovoid set of fragments shaded by a disc are carved to the depth of the hemisphere (formula can be found in Appendix C) while discs sprites are generated with a geometry shader stage. Note that subsampling increases γ, the angular opening of a texel. Therefore, to remain conservative, the radius of the hemispheres generating the coverage buffer should depend on z as follow:

$$R_S(z) = N \times \delta + z \times \tan(\gamma/2) \tag{4}$$

Figure 7 displays the coverage buffer and the resulting culling.

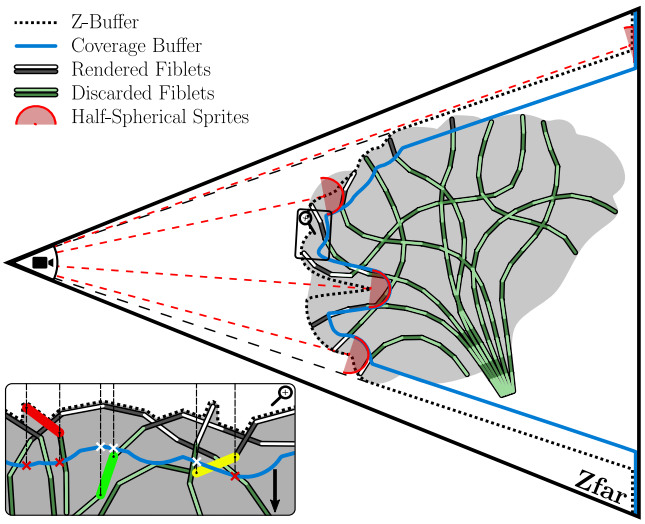

Figure 6: *2D analogy of the fiblet culling test based on the coverage buffer. To avoid cluttering the figure with a re-projection, the camera position is considered static. The red hemispheres illustrate how the coverage buffer is computed from the z-buffer at the previous frame. The zoomed inset illustrates the two visibility tests performed by the task shader. In that case, only the red fiblet needs to be synthesized and rendered since both points fail the depth test against the coverage buffer.*

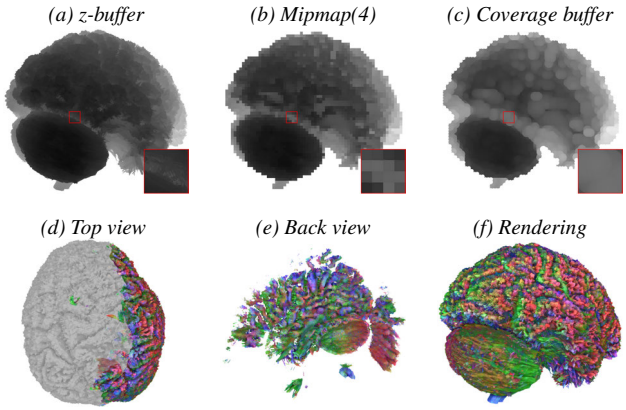

Figure 7: *Illustration of the z-buffer (a), the max mipmap subsampling (level 4) (b), its dilation with carved sphere sprites results in the coverage buffer (c). (d) and (e) shows from a different location the subset of fiblets synthesized and rendered (f).*

Temporal coherence techniques require a mechanism to update the relevant information from the previous frame. Here, the coverage buffer is re-projected and rendered in the novel view using a mesh-grid of that coverage buffer. Section 4.4 discusses effects of re-projection on culling conservativeness.

3.4.3. Frustum Culling

Frustum culling enhances performances when the camera is inside the tractogram. Fast frustum fiblet culling leans on the same morphological considerations described in Sec.3.4.1. Every first point of a fiblet outside a trapezoidal volume dilated from the camera frustum leads to culling the entire fiblet. Homogeneous coordinates come in handy to test that volume. Let x_h, y_h, z_h, and w_h be the homogeneous coordinates of a projected point p. Given the (half) field of view $Hfov_x$ and $Hfov_y$, the fiblet frustum belonging shortens to

$$\frac{|x_h|}{w_h + \frac{R_s}{\sin(HFov_x)}} < 1 \;\; and \;\; \frac{|y_h|}{w_h + \frac{R_s}{\sin(HFov_y)}} < 1 \qquad (5)$$

$$Z_{near} - R_s < w_h < Z_{far} + R_s$$

3.4.4. Far view rendering

When seen from very far, many redundant fiblets are not discarded while many segments are rasterized in the same pixel, harming performances. To counter that effect, a LoD of the fiblet is considered. The screen size upper bound of a fiblet rendering can be estimated thanks to S_{fiblet}. When that size is smaller than a few pixels (4 in our implementation), the mesh shader does not synthesize the fiblet. Instead, it is restricted to write a line meshlet linking the first point of that fiblet to the first point of the next fiblet, avoiding sub-pixelic lines rasterization and unnecessary decompression.

4. Results and analysis

4.1. Data and compression

To evaluate our method, we use dMRI data from the Human Connectome Project (*HCP*) [VUA*12], which shares anonymized data of hundreds of people at a 1.25 mm^3 voxel size. It is recommended to use a tenth of this size as the stepsize for the tracking (seed propagation) algorithm – $0.1mm$ here [TCC12]. Tractograms are computed using *Mrtrix3* [TCC12] with the *iFOD1* and the *SD_STREAM* algorithms. We randomly selected three different patients from the dataset and generated 6 different tractograms per algorithm and patient, resulting in 36 different tractograms. Table 1 presents the different files we obtain for one patient (the other two are in the additional materials). These tractograms are recorded in the (raw) tck format of *Mrtrix3* [TCC12] and range from $5GB$ to almost $100GB$. They contain at least hundreds of millions of segments, with 8 billion lines for the most massive one. Table 1 also contains the compressed size of the same files, both with *Qfib* [MRG*20] (using a 8 bits octahedral quantization) and our approach. We notice that *Qfib* achieves a higher compression ratio, which is expected due to our fiber subdivision into fiblets, required to fulfill the GPU needs.

4.2. Compression precision and comparison with *Qfib*

Our compression – as *Qfib* – is lossy. Indeed, we quantize the successive directions, resulting in small displacements compared to the original raw data. The maximum and average error of both techniques (measured at the point level) are exposed in Tab. 1. Our error is always significantly lower than that of *Qfib*. This was expected as our clustered approach allows for readjustments at each fiblet where *Qfib* relies only on the two first points of each fiber. This comes at the expense of our compression ratio. Our error remains low for both tractogram algorithms, way under the dMRI

Table 1: *Size (GB), maximum and average error (µm) of the different tractograms generated for one patient for the raw data (tck),* Qfib *[MRG*20], and our approach. The data for other patients can be found in additional materials.*

Algorithm	iFOD1						SD_STREAM					
# Fibers	500k		3M		7M	10M	500k		3M		7M	10M
Stepsize (mm)	0.1	0.05	0.1	0.05	0.1	0.1	0.1	0.05	0.1	0.05	0.1	0.1
# Segments (billion)	0.43	0.89	2.60	5.37	6.07	8.10	0.38	0.75	2.29	4.50	5.35	7.65
	Size (GB)											
Tck	5.21	10.7	31.3	64.4	72.9	97.3	4.59	8.99	27.6	54.0	64.3	91.9
Qfib	**0.446**	**0.906**	**2.67**	**5.44**	**6.24**	**8.34**	**0.394**	**0.761**	**2.37**	**4.57**	**5.52**	**7.89**
Ours	0.572	1.16	3.43	6.93	8.00	10.7	0.507	0.971	3.04	5.83	7.09	10.1
	Maximum error (μm)											
Qfib	47.3	25.4	49.3	27.1	49.8	48.9	907	895	944	928	941	934
Ours	**17.1**	**8.20**	**17.8**	**8.64**	**19.7**	**12.0**	**22.5**	**10.9**	**22.8**	**11.2**	**24.0**	**23.8**
	Average error (μm)											
Qfib	18.5	9.76	18.6	9.79	18.7	21.6	192	188	192	189	192	192
Ours	**5.03**	**2.23**	**5.04**	**2.23**	**5.05**	**4.17**	**7.34**	**2.99**	**7.34**	**2.99**	**7.34**	**7.34**

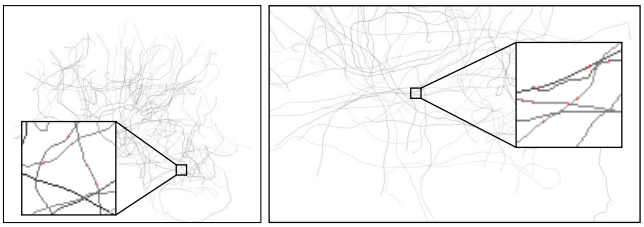

Figure 8: *Pixel difference between our fiblet rendering of a few fibers and a rendering without compression. Pixel differences are shown in **red**. The left image gives a view from far while the right image is close.*

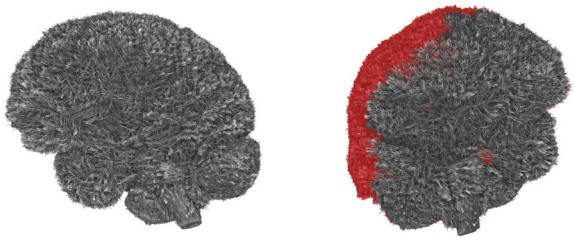

Figure 9: *Screenshot of a failure case of the coverage buffer re-projection under big transforms. Red zones highlights faulty culled fiblets when applying a* 60° *rotation between two frame.*

precision, and an order of magnitude under the stepsize. A major benefit of our fiblet approach is that our error only depends on the stepsize (which infers the maximum angle by construction). The maximum error is reached after 58 compressed points (even using delta-coding, see [MRG*20]), instead of being dependent on the longest fiber with *Qfib*.

4.3. Effects of compression on rendering

To ensure that the loss is not visually significant, we highlighted in Fig.8 the differences between an OpenGL rendering of the raw fibers and our fiblet rendering. It is noticeable that these differences are sparse and of no more than a single pixel. Indeed, since compression error is small (see Tab. 1), at any relevant zoom level the visual error never exceeds the rasterizer quantization step: a pixel.

4.4. Effect of coverage re-projection on rendering

The use of mipmapping to reduce the cost of the z-buffer dilation also results in a dilation with a radius larger than R_S, typically around $1.2 \times R_S$. This extra radius almost guarantees conservativeness of the coverage buffer re-projection under small camera transforms. Since our pipeline is designed for high framerate, we were only able to experience failures of the re-projection on the biggest tractogram (meant to test the GPU limits), when moving the camera extremely fast. Figure 9 displays this artifact. It is worthy to highlight that such artifacts would only last for one frame.

4.5. Performance analysis and comparison between graphics pipelines

Table 2 presents the average time in milliseconds to draw a frame using our method on different graphics pipelines, on a machine equipped with an Intel Xeon e5-1650 v4 (6 cores - 12 threads), 64 GB of RAM, and an Nvidia RTX 2080Ti with 11GB of VRAM. The viewport is set to 1920×1080, and we compute the exact same frames around the different tractograms. Averaging on a rotation is necessary to provide an accurate measurement of a typical use case: firstly, the number of culled fibers and even fragments shaded depends on the brain shape in the camera frame, and secondly, since our culling technique makes use of temporal coherence, the efficiency of the coverage buffer is directly impacted by the amplitude of the transformation from one view to another. Our step choice of (1.14°) per frame encompasses the behavior of a normal interaction case, reaching a $90^\circ/s$ speed at 80 fps (and $68^\circ/s$ at 60 fps).

4.5.1. Geometry shader

This graphics pipeline is the most straightforward in terms of complexity. Indeed, its role is to amplify the geometry, which is done here by synthesizing each fiblet within the shader, emitting a maximum of 60+1 vertices per polyline. The implementation of the iterative algorithm is fairly simple as the vertices are emitted while building the chain matrix product. The culling approach is achieved on this pipeline by testing for the fiblet visibility and terminating the invocation without emitting any vertex if the test fails. As expected the performances are not great since geometry shaders struggle when generating more than four vertices. The improvement brought by the culling mechanism is in proportion the worst of the four pipelines. The reason is that geometry shaders stall under an unbalanced load. As a consequence of culling, some invocations are required to generate 61 vertices while others generate no vertex.

4.5.2. Tessellation shader

Unlike the geometry shader, the tessellation shader stage aims at generating on-chip a massive number of vertices. Tessellation natively supports generation of 3D polylines, up to 65 vertices (MAX_TESSELLATION = 64) on most hardware. Culling is fairly well supported as it is specified that a tessellation level set to 0 leads to end the processing of the patch. The Tessellation Control Shader (TCS) is responsible for testing the fiblet visibility and setting the tessellation level accordingly, 0 if the fiblet should be culled or up to 60, thus calling 61 Tessellation Evaluation Shader (TES) invocations. It is usually advised to have the TES compute the vertices location in parallel. However, the sequential characteristics of the chain product that needs to be computed prevent synthesizing each vertex in its own TES invocation, since those invocations cannot share dynamic read/write memory between each other. Consequently, we iteratively synthesize the fiblet inside the TCS and store them in the specific patch memory that can be read by any TES invocation called by that TCS. Then, the TES simply projects the vertex to the camera frame. Performances of this pipeline under culling outperform geometry pipeline almost by a factor of two. For older GPUs not supporting mesh shaders, our tessellation pipeline still offers way better performances compared to existing tools (See Tab.3).

4.5.3. Single thread mesh shader

We first implemented the sequential – *Qfib* like – decompression of each fiblet inside the mesh shader. The result of such an implementation is presented on the third line of Tab.2. Because of threads idling, the performances without fiblet culling are very poor. However, timings, when the task shader culling is activated, improve significantly, proving the efficiency of our implementation to massively cull fiblets.

4.5.4. Parallel mesh shader

In this implementation, we exploit the two stages of the mesh pipeline as described in Sec.3.3.2 and Sec.3.4. With culling disabled, our parallel scheme with 32 threads warps is almost 5 times faster than the single-threaded decompression. Most of the time is spent computing the parallel prefix matrix multiplication, where shared memory accesses introduce some stalling. When combined with the task shader culling, we reach significant and scalable performances.

4.6. Comparison to other tractograms rendering tools

We compare our performances and capabilities with existing tractogram visualization tools (Tab. 3). Tractograms whose raw size is bigger than the VRAM cannot be visualized by other methods. Consequently, we limit our comparison to sizes supported by other methods. The 500k fibers tractogram can only be visualized using *TrackVis* [WBSW07], our basic OpenGL implementation, the Point Cloud method (*PCC*) [SKW21], and our method. The Point Cloud method is included in the comparison as a high performances computer graphics timing reference. However, it only displays points, where tractograms approaches are designed for lines, meaning that zooming on the data makes the visualization unreadable. We also experimented with *Nanite* [KSW21], however, our data is not adapted to the triangle-based approach of the technique, requiring triangle strips to represent the fibers, thus increasing data size and complexity, making unfair the comparison. Useful frame profiling information under different conditions are presented in the additional materials.

5. Extended visualization tools

We aim to describe some implemented tools compatible and accelerated with our pipeline. Indeed, our fiblet culling pipeline makes possible fast additional custom culling while the deferred nature of the pipeline allows many per-pixel shading effects.

5.1. Custom culling: mesh culling

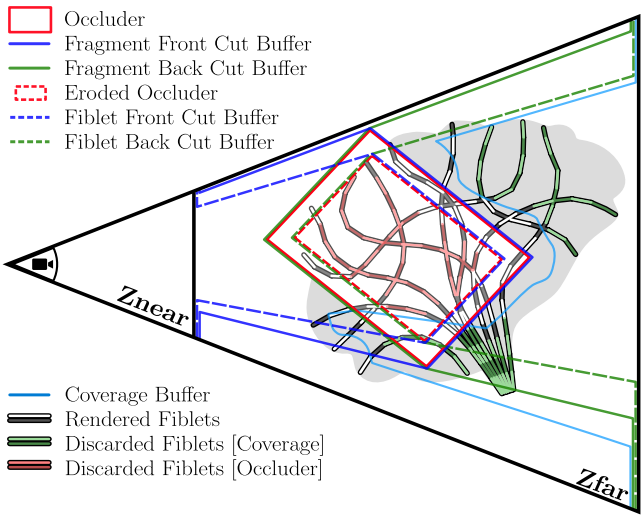

Figure 10: *2D analogy of the fiblet culling method when a mesh cut is added to the coverage buffer. The test to cull a fiblet or discard a fragment is the same, but applied with back/front buffers from the dilated mesh in the task shader and applied with back/front buffers from the mesh in the fragment shader.*

A classic medical data feature is to allow axis-aligned plan cuts.

Table 2: *Rendering times in milliseconds (ms) of our approach compared to alternative pipelines, with and without culling. Results are presented for a single patient, other patients are presented in additional materials.*

Algorithm	iFOD1						SD_STREAM					
# Fibers	500k		3M		7M	10M	500k		3M		7*M*	10M
Stepsize (mm)	0.1	0.05	0.1	0.05	0.1	0.1	0.1	0.05	0.1	0.05	0.1	0.1
	Without culling											
Geometry	104	203	621	1223	1446	1934	92.1	171	550	1029	1281	1816
Tessellation	76.5	138	456	835	1062	1420	68.1	116	403	701	938	1339
Mesh (1 thread)	211	434	1278	2600	2975	3969	190	367	1132	2183	2632	3748
Mesh (parallel)	**44.6**	**86.9**	**267**	**559**	**626**	**837**	**41.1**	**74.6**	**238**	**441**	**555**	**791**
	With culling											
Geometry	84.4	140	448	723	1005	1209	80.2	128	427	668	960	1351
Tessellation	44.0	85.9	251	485	579	762	40.0	73.0	225	418	520	741
Mesh (1 thread)	25.6	33.2	94.3	132	196	182	30.0	37.5	111	137	226	307
Mesh (parallel)	**7.84**	**8.92**	**28.0**	**33.9**	**55.9**	**55.9**	**8.45**	**10.1**	**29.8**	**36.2**	**59.9**	**81.8**

Table 3: *Rendering times in milliseconds (ms) and file sizes of our fiblet culling and rendering method compared to existing solutions on various datasets. N/A represents the datasets that the corresponding method was not able to load and render.*

Dataset		3D Slicer	Paraview	TrackVis	Open Walnut	Fiber Navigator	Basic OpenGL	PCC	Ours
# Fibs	# Segs								
10k	8M	≈ 6.90	<16.7	<8.00	<6.94	<6.06	≈ 2.70	≈ **1.50**	≈ 2.08
Encoding size (GB)		0.270		0.963	0.128	0.964		0.289	**0.011**
100k	81M	N/A	≈ 25.0	≈ 20.0	≈ 21.7	<6.06	≈ 17.2	≈ 8.70	≈ **3.62**
Encoding size (GB)		2.81		0.972	1.30	0.973		2.92	**0.107**
250k	202M	N/A	≈ 50	≈ 50	≈ 55.6	N/A	≈ 40	≈ 22.2	≈ **4.88**
Encoding size (GB)		7.14		2.43	3.24	2.43		7.28	**0.268**
500k	434M	N/A	N/A	≈ 500	N/A	N/A	≈ 90.9	≈ 33.3	≈ **7.87**
Encoding size (GB)		15.4		5.20	6.94	5.21		15.6	**0.572**
Data format		vtk		trk	fib	tck		las	fft

We propose the option to combine to our pipeline a more complete cutting scheme using a custom closed orientable surface mesh. The morphological culling strategy described in Sec.3.4 applies in a very similar manner by computing a back and front coverage buffer for the occluding mesh. Those two buffers are processed in two passes, each one rendering the dilated surface mesh. Backface culling with the *less* depth test function is used for the front buffer and front-face culling with the *greater* depth test function is used for the back buffer. Those buffers are combined within the task shader to cull fiblets inside or outside of the mesh, depending on user preference. The fragment shader is also adapted to include a discarding condition based on a front and back buffer of the rendered surface mesh. Figure 10 summarizes the strategy, while Figure 11 displays cuts result. Screenshots of back and front depth buffers are available in the additional materials. Concave surface meshes might infer some approximation from certain views since the back/front buffers allow only one range of depth culling. Complex meshes cut might result in a framerate drop as the number of rendered fiblets is roughly proportional to the surface of the interface viewed from the camera. On top of that, discarding at the scale of fragment forces to disable GPU early z-test.

5.2. Custom culling: criterion culling

In addition to mesh culling, our pipeline makes it possible to cull fiblets and fibers according to a custom criterion computed for each fiber and stored in a VRAM buffer. Figure 11 displays renderings using a fiber-proximity criterion and a fiber-length criterion. More is presented in the additional materials.

5.3. Shading effects

We display renders with SSAO since the medical data visualization community mostly opts for informative shading to emphasize the perception of mesostructures. For better light dependent shading, we propose a shading of *Illuminated Line* inspired from [MPSS05]. This is possible since per-fragment normal can be retrieved thanks to tangents stored in the G-buffer. Fiber albedo can be defined using orientation or even with a custom criterion (see Sec 5.2) combined with a colormap. Figure 11 illustrates different shading effects. In this paper, we rely on native FXAA for antialiasing.

5.4. Selection rendering

Since our G-buffer keeps track of a 32bits individual fiber ID, any rendered fragment written during the deferred phase can be picked with the mouse, returning the selected IDs to the CPU. This allows

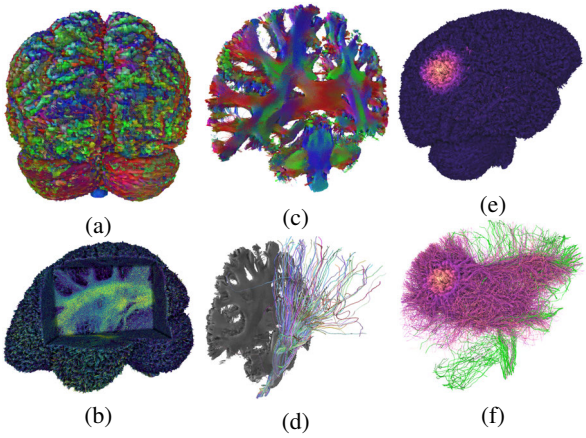

Figure 11: *Screen captures of our fiblet renderings from a 7M fibers tractogram. (a) illuminated lines with SSAO, (b) colormap shading depending on fiber length with a box mesh cut (cull inside), (c) orientation shading with mesh cut (cull outside) using a long flat box, (d) selection rendering from clicked fibers on brain slice, (e) colormap shading depending on the distance of the fibers to a zone, (f) custom culling displaying only closest-to-zone fibers, with additional selected fibers in green.*

the generation of an index buffer flagging the fiblets composing the selected fibers. Then, a draw call triggers a tessellation shader used to decompress and generate cylindrical volumetric fibers. The rendering is finally blended into the whole tractogram, as displayed in Fig. 11. It is similarly possible to draw clusters of individual fibers generated from the result of a prior tractogram analysis.

6. Conclusion

With *fiblets*, we introduced a new brain tractogram rendering approach that takes full advantage of the GPU mesh pipeline. The task shader running our morphological adaptation of temporal coherence depth culling while the threads running mesh shader invocations concurrently cooperate to swiftly synthesize the fibers from the fiblets within a warp, following a parallel prefix scheme. We exposed remarkable framerates and are able to accurately visualize massive brain tractograms for the first time while allowing for the use of interaction tools, down to the fiber-level granularity.

Our compression scheme improves upon *QFib* and shares its limitations. The main one is the need for a constant stepsize at tractogram scale, a fundamental property for our culling strategy. Note however that most tractogram generation algorithms do use such a constraint. We also expect the compressed dataset to fit entirely in VRAM. To render an even larger tractogram, a streaming strategy – retrieving only relevant sets of fiblets from out-of-core storage – could be deployed at the cost of reduced performances. When the screen-space fiber density becomes too high, some z-fight aliasing might appear.

Our fiblet pipeline could be adapted to other applications containing lines with similar properties, for example, medical data such as muscles or capillaries. The memory efficiency of our method opens a way to superpose multiple large tractograms together to perform all sorts of comparisons – something only performed in a very indirect manner today. In that perspective, dynamic transparency would be interesting to study, potentially using multi-frame depth peeling. A custom anti-aliasing strategy could be deployed through an MSAA G-buffer resolving the most represented fibers within a pixel. In the context of virtual reality – more and more studied for medical training – the coverage buffer computation, as well as the synthesis of the fiblets, could be factorized for the two eyes cameras. Our fiblet pipeline speeds up cases where a fast visualization of massive raw line sets is required. For instance, streamlines, as well as networks representation, may benefit from fiblets. In the medical context, massive visualization of blood capillary networks and muscle fibers can also be addressed with our method.

References

[AGL05] AHRENS, JAMES, GEVECI, BERK, and LAW, CHARLES. "Paraview: An end-user tool for large data visualization". *The visualization handbook* 717.8 (2005), 717–731. DOI: 10.1016/B978-012387582-2/50038-1 2.

[Aya15] AYACHIT, UTKARSH. *The paraview guide: a parallel visualization application*. Kitware, Inc., 2015 2.

[BWPP04] BITTNER, JIRI, WIMMER, MICHAEL, PIRINGER, HARALD, and PURGATHOFER, WERNER. "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful". *CGF* 23.3 (2004), 615–624. DOI: 10.1111/j.1467-8659.2004.00793.x 3.

[CBF*15] CHAMBERLAND, MAXIME, BERNIER, MICHAEL, FORTIN, DAVID, et al. "3D interactive tractography-informed resting-state fMRI connectivity". *Frontiers in Neuroscience* 9 (2015), 275. DOI: 10.3389/fnins.2015.00275 2.

[CGG*04] CIGNONI, PAOLO, GANOVELLI, FABIO, GOBBETTI, ENRICO, et al. "Adaptive Tetrapuzzles: Efficient out-of-Core Construction and Visualization of Gigantic Multiresolution Polygonal Models". *ACM ToG*. Vol. 23. 3. 2004, 796–803. DOI: 10.1145/1015706.1015802 3.

[Cra11] CRASSIN, CYRIL. "GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes". PhD thesis. UNIVERSITE DE GRENOBLE, July 2011 3.

[CWF*14] CHAMBERLAND, MAXIME, WHITTINGSTALL, KEVIN, FORTIN, DAVID, et al. "Real-time multi-peak tractography for instantaneous connectivity display". *Frontiers in Neuroinformatics* 8 (2014), 59. DOI: 10.3389/fninf.2014.00059 2.

[DÇ15] DEMIR, A. and ÇETINGÜL, H. E. "Sequential Hierarchical Agglomerative Clustering of White Matter Fiber Pathways". *IEEE TBME* 62.6 (2015), 1478–1489. DOI: 10.1109/TBME.2015.2391913 2.

[DDS03] DÉCORET, XAVIER, DEBUNNE, GILLES, and SILLION, FRANÇOIS. "Erosion Based Visibility Preprocessing". *Proc. EGSR*. 2003, 281–288 3.

[DMP*19] DELMONTE, ALESSANDRO, MERCIER, CORENTIN, PALLUD, JOHAN, et al. "White matter multi-resolution segmentation using fuzzy set theory". *ISBI*. 2019, 459–462. DOI: 10.1109/ISBI.2019.8759506 3.

[DVS03] DACHSBACHER, CARSTEN, VOGELGSANG, CHRISTIAN, and STAMMINGER, MARC. "Sequential Point Trees". *ACM ToG* 22.3 (2003), 657–662. DOI: 10.1145/882262.882321 3.

[EBRI09] EVERTS, MAARTEN H, BEKKER, HENK, ROERDINK, JOS BTM, and ISENBERG, TOBIAS. "Depth-dependent halos: Illustrative rendering of dense line data". *IEEE TVCG* 15.6 (2009), 1299–1306. DOI: 10.1109/TVCG.2009.138 2.

[EHS12] EICHELBAUM, SEBASTIAN, HLAWITSCHKA, MARIO, and SCHEUERMANN, GERIK. "LineAO—Improved three-dimensional line rendering". *IEEE TVCG* 19.3 (2012), 433–445. DOI: 10.1109/TVCG.2012.142 2.

[EHS13] EICHELBAUM, SEBASTIAN, HLAWITSCHKA, MARIO, and SCHEUERMANN, GERIK. "Openwalnut: An open-source tool for visualization of medical and bio-signal data". *Biomedical Eng./Biomedizinische Tech.* 58.SI-1-Track-G (2013), 000010151520134183. DOI: 10.1515/bmt-2013-4183 2.

[GBA*14] GARYFALLIDIS, ELEFTHERIOS, BRETT, MATTHEW, AMIRBEKIAN, BAGRAT, et al. "Dipy, a library for the analysis of diffusion MRI data". *Frontiers in Neuroinformatics* 8 (2014), 8. DOI: 10.3389/fninf.2014.00008 2.

[GBC*12] GARYFALLIDIS, ELEFTHERIOS, BRETT, MATTHEW, CORREIA, MARTA MORGADO, et al. "QuickBundles, a Method for Tractography Simplification". *Frontiers in Neuroscience* 6.175 (2012). DOI: 10.3389/fnins.2012.00175 2, 3.

[GM05] GOBBETTI, ENRICO and MARTON, FABIO. "Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms". *ACM ToG* 24 (2005), 878–885. DOI: 10.1145/1186822.1073277 3.

[Goo17] GOOGLE. *Draco: 3D graphics compression*. 2017. URL: https://google.github.io/draco/ 3.

[Gün20] GÜNTHER, T. "Visibility, Topology, and Inertia: New Methods in Flow Visualization". *IEEE CG&A* 40.2 (2020), 103–111. DOI: 10.1109/MCG.2019.2959568 3.

[HFZ*20] HAEHN, DANIEL, FRANKE, LORAINE, ZHANG, FAN, et al. "TRAKO: Efficient Transmission of Tractography Data for Visualization". *MICCAI*. 2020, 322–332. DOI: 10.1007/978-3-030-59728-3_32 3.

[HSO07] HARRIS, MARK, SENGUPTA, SHUBHABRATA, and OWENS, JOHN D. "Parallel Prefix Sum (Scan) with CUDA". *GPU Gems 3*. Aug. 2007. Chap. 39, 851–876 5.

[IRR*22] IBRAHIM, MOHAMED, RAUTEK, PETER, REINA, GUIDO, et al. "Probabilistic Occlusion Culling using Confidence Maps for High-Quality Rendering of Large Particle Data". *IEEE TVCG* 28.1 (2022), 573–582. DOI: 10.1109/TVCG.2021.3114788 3.

[Ise15] ISENBERG, TOBIAS. "A Survey of Illustrative Visualization Techniques for Diffusion-Weighted MRI Tractography". *Visualization and Processing of Higher Order Descriptors for Multi-Valued Data*. wiley, 2015, 235–256. DOI: 10.1007/978-3-319-15090-1_12 2.

[JCLR19] JANSSON, ERIK SVEN VASCONCELOS, CHAJDAS, MATTHÄUS G., LACROIX, JASON, and RAGNEMALM, INGEMAR. "Real-Time Hybrid Hair Rendering". *EGSR*. 2019, 1–8. DOI: 10.2312/sr.20191215 2.

[KAC15] KRESS, J., ANDERSON, E., and CHILDS, H. "A visualization pipeline for large-scale tractography data". *LDAV*. 2015, 115–123. DOI: 10.1109/LDAV.2015.7348079 2.

[KNM*20] KERN, MICHAEL, NEUHAUSER, CHRISTOPH, MAACK, TORBEN, et al. "A Comparison of Rendering Techniques for 3D Line Sets with Transparency". *IEEE TVCG* 27.8 (2020). DOI: 10.1109/TVCG.2020.2975795 3.

[Kra18] KRAEMER, MANUEL. *Using Turing Mesh Shaders: NVIDIA Asteroids Demo*. 2018. URL: https://developer.nvidia.com/blog/using-turing-mesh-shaders-nvidia-asteroids-demo/ 2.

[KRW19] KANZLER, MATHIAS, RAUTENHAUS, MARC, and WESTERMANN, RÜDIGER. "A Voxel-Based Rendering Pipeline for Large 3D Line Sets". *IEEE TVCG* 25.7 (2019), 2378–2391. DOI: 10.1109/TVCG.2018.2834372 3.

[KSA13] KÄMPE, VIKTOR, SINTORN, ERIK, and ASSARSSON, ULF. "High resolution sparse voxel dags". *ACM ToG* 32.4 (2013), 1–13. DOI: 10.1145/2461912.2462024 3.

[KSW05] KRUGER, J., SCHNEIDER, J., and WESTERMANN, R. "DUODECIM - a structure for point scan compression and rendering". *PBG 2005*. 2005, 99, -–146. DOI: 10.1109/PBG.2005.194070 3.

[KSW21] KARIS, BRIAN, STUBBE, RUNE, and WIHLIDAL, GRAHAM. *Nanite, A Deep Dive*. Siggraph course. 2021 3, 9.

[Kub18] KUBISCH, CHRISTOPH. *New Rendering Techniques for Real-Time Graphics: Turing - Mesh Shaders*. Talk at SIGGRAPH 2018. 2018 2.

[Kub20] KUBISCH, CHRISTOPH. *Using Mesh Shaders for Professional Graphics*. 2020. URL: https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/ 2.

[Lin14] LINDSTROM, PETER. "Fixed-Rate Compressed Floating-Point Arrays". *IEEE TVCG* 20.12 (2014), 2674–2683. DOI: 10.1109/TVCG.2014.2346458 3.

[LJSL21] LEE, GI BEOM, JEONG, MOONSOO, SEOK, YECHAN, and LEE, SUNGKIL. "Hierarchical Raster Occlusion Culling". *CGF* 40.2 (2021), 489–495. DOI: https://doi.org/10.1111/cgf.142649 3.

[LKE18] LEE, SUNGKIL, KIM, YOUNGUK, and EISEMANN, ELMAR. "Iterative Depth Warping". *ACM ToG* 37.5 (2018). DOI: 10.1145/3190859 3.

[LS07] LI, L. and SHEN, H. "Image-based streamline generation and rendering". *IEEE TVCG* 13.3 (2007), 630–640. DOI: 10.1109/TVCG.2007.8093671 3.

[MB20] MICHEL, ÉLIE and BOUBEKEUR, TAMY. "Real Time Multiscale Rendering of Dense Dynamic Stackings". *CGF* 39.7 (2020), 169–179. DOI: https://doi.org/10.1111/cgf.14135 3.

[MBJ*15] MATTAUSCH, OLIVER, BITTNER, JIRI, JASPE, ALBERTO, et al. "CHC+RT: Coherent Hierarchical Culling for Ray Tracing". *CGF* 34.2 (2015), 537–548. DOI: 10.1111/cgf.12582 3.

[MBW08] MATTAUSCH, OLIVER, BITTNER, JIRI, and WIMMER, MICHAEL. "CHC++: Coherent Hierarchical Culling Revisited". *CGF* 27.2 (2008), 221–230. DOI: 10.1111/j.1467-8659.2008.01119.x 3.

[MCCV99] MORI, SUSUMU, CRAIN, BARBARA J, CHACKO, VADAPPURAM P, and VAN ZIJL, PETER CM. "Three-dimensional tracking of axonal projections in the brain by magnetic resonance imaging". *Annals of Neurology* 45.2 (1999), 265–269. DOI: 10.1002/1531-8249(199902)45:2<265::AID-ANA21>3.0.CO;2-3 1.

[MG16] MERRILL, DUANE and GARLAND, MICHAEL. "Single-pass Parallel Prefix Scan with Decoupled Lookback". 2016 5.

[MGR*18] MERCIER, CORENTIN, GORI, PIETRO, ROHMER, D., et al. "Progressive and Efficient Multi-Resolution Representations for Brain Tractograms". *EG VCBM*. 2018, 89–93. DOI: 10.2312/vcbm.20181232 3.

[Mou18] MOURS, PATRICK. *Mesh Shaders in Turing*. Talk at GTC Europe. 2018 2.

[MPSS05] MALLO, OVIDIO, PEIKERT, RONALD, SIGG, CHRISTIAN, and SADLO, FILIP. "Illuminated lines revisited". *VIS 05. IEEE Visualization, 2005*. 2005, 19–26. DOI: 10.1109/VISUAL.2005.1532772 2, 10.

[MRG*20] MERCIER, CORENTIN, ROUSSEAU, SYLVAIN, GORI, PIETRO, et al. "QFib: Fast and Efficient Brain Tractogram Compression". *Neuroinformatics* 18.4 (2020), 627–640. DOI: 10.1007/s12021-020-09452-0 3, 4, 7, 8.

[MSS*10] MEYER, QUIRIN, SÜSSMUTH, JOCHEN, SUSSNER, GERD, et al. "On floating-point normal vectors". *CGF* 29.4 (2010), 1405–1409. DOI: 10.1111/j.1467-8659.2010.01737.x 4.

[MV02] MORI, SUSUMU and VAN ZIJL, PETER CM. "Fiber tracking: principles and strategies–a technical review". *NMR in Biomedicine* 15.7-8 (2002), 468–480. DOI: 10.1002/nbm.781 1.

[MYB16] MALEKI, SEPIDEH, YANG, ANNIE, and BURTSCHER, MARTIN. "Higher-Order and Tuple-Based Massively-Parallel Prefix Sums". *Proc. SIGPLAN* 51.6 (2016), 539–552. DOI: 10.1145/2908080.2908089 5.

[PFK07] PETROVIC, V., FALLON, J., and KUESTER, F. "Visualizing Whole-Brain DTI Tractography with GPU-based Tuboids and LoD Management". *IEEE TVCG* 13.6 (2007), 1488–1495. DOI: 10.1109/TVCG.2007.70532 2.

[PJHD15] PRESSEAU, CAROLINE, JODOIN, PIERRE-MARC, HOUDE, JEAN-CHRISTOPHE, and DESCOTEAUX, MAXIME. "A new compression format for fiber tracking datasets". *NeuroImage* 109 (2015), 73–83. DOI: 10.1016/j.neuroimage.2014.12.058 3.

[PLSK06] PIEPER, STEVE, LORENSEN, BILL, SCHROEDER, WILL, and KIKINIS, RON. "The NA-MIC Kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the medical image computing community". *IEEE ISBI*. 2006, 698–701. DOI: 10.1109/ISBI.2006.1625012 2.

[PT02] PANTAZOPOULOS, IOANNIS and TZAFESTAS, SPYROS. "Occlusion Culling Algorithms: A Comprehensive Survey". *JIRS* 35 (2002), 123–156. DOI: 10.1023/A:1021175220384 3.

[PWK20] PREUSS, DANIEL, WEINKAUF, TINO, and KRÜGER, JENS HARALD. "A Discrete Probabilistic Approach to Dense Flow Visualization". *IEEE TVCG* 27.12 (2020), 4347–4358. DOI: 10.1109/TVCG.2020.3006995 3.

[Rah19] RAHUL SATHE, MANUEL KRAEMER. *Applications of Mesh Shading with Dx12*. Talk at SIGGRAPH 2019. 2019 2.

[RB20] ROUSSEAU, SYLVAIN and BOUBEKEUR, TAMY. "Unorganized Unit Vectors Sets Quantization". *JCGT* 9.3 (2020), 92–107 4, 13.

[RHD17] RHEAULT, FRANCOIS, HOUDE, JEAN-CHRISTOPHE, and DESCOTEAUX, MAXIME. "Visualization, Interaction and Tractometry: Dealing with Millions of Streamlines from Diffusion MRI Tractography". *Frontiers in Neuroinformatics* 11 (2017), 42. DOI: 10.3389/fninf.2017.00042 2, 3.

[RL00] RUSINKIEWICZ, SZYMON and LEVOY, MARC. "QSplat: A Multiresolution Point Rendering System for Large Meshes". *ACM ToG*. 2000, 343–352. DOI: 10.1145/344779.344940 3.

[SKW21] SCHUTZ, MARKUS, KERBL, BERNHARD, and WIMMER, MICHAEL. "Rendering Point Clouds with Compute Shaders and Vertex Order Optimization". *CGF* 40.4 (2021), 115–126. DOI: 10.1111/cgf.14345 3, 9.

[SMAS13] SOARES, JOSE, MARQUES, PAULO, ALVES, VICTOR, and SOUSA, NUNO. "A hitchhiker's guide to diffusion tensor imaging". *Frontiers in Neuroscience* 7 (2013), 31. DOI: 10.3389/fnins.2013.00031 2.

[TCC12] TOURNIER, J.-DONALD, CALAMANTE, FERNANDO, and CONNELLY, ALAN. "MRtrix: Diffusion tractography in crossing fiber regions". *IJIST* 22.1 (2012), 53–66. DOI: 10.1002/ima.22005 7.

[TML11] TOURNIER, JACQUES-DONALD, MORI, SUSUMU, and LEEMANS, ALEXANDER. "Diffusion tensor imaging and beyond". *Magnetic Resonance in Medicine* 65.6 (2011), 1532–1556. DOI: 10.1002/mrm.22924 1, 3.

[Tor19] TORABI, PEYMAN. "Skeletal Animation Optimization Using Mesh Shaders". PhD thesis. Blekinge Institute of Technology, 2019 2.

[VMG17] VILLANUEVA, ALBERTO JASPE, MARTON, FABIO, and GOBETTI, ENRICO. "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". *JCGT* 6.2 (2017), 1–30 3.

[VUA*12] VAN ESSEN, D.C., UGURBIL, K., AUERBACH, E., et al. "The Human Connectome Project: A data acquisition perspective". *NeuroImage* 62.4 (2012), 2222–2231. DOI: 10.1016/j.neuroimage.2012.02.018 7.

[WBK*07] WARD, KELLY, BERTAILS, FLORENCE, KIM, TAE-YONG, et al. "A Survey on Hair Modeling: Styling, Simulation, and Rendering". *IEEE TVCG* 13.2 (2007), 213–234. DOI: 10.1109/TVCG.2007.30 2.

[WBSW07] WANG, RUOPENG, BENNER, THOMAS, SORENSEN, ALMA GREGORY, and WEDEEN, VAN JAY. "Diffusion toolkit: a software package for diffusion imaging data processing and tractography". *Magnetic Resonance in Medicine* 15.3720 (2007) 2, 9.

[WY17] WU, KUI and YUKSEL, CEM. "Real-time Fiber-level Cloth Rendering". *I3D*. 2017. DOI: 10.1145/3023368.3023372 2.

[ZLB16] ZHAO, SHUANG, LUAN, FUJUN, and BALA, KAVITA. "Fitting Procedural Yarn Models for Realistic Cloth Rendering". *ACM ToG* 35.4 (2016). DOI: 10.1145/2897824.2925932 2.

Appendix A: compression (on CPU)

We establish the unit vector mapping and unmapping according to [RB20] with the two differences that our chained frames only calls mapping on the average vector $\mathbf{X} = (1,0,0)$ and that, since quantization is performed on an half-octahedron, the mapping factor somewhat differs. We recall α the maximal admissible angle to define $Ratio = 1 - \cos\alpha$ and $\mathbf{V}$ the vector to be mapped:

$$let\ \mathbf{X}^* = \frac{\mathbf{V} - (\mathbf{V}\cdot\mathbf{X})\mathbf{X}}{||\mathbf{V} - (\mathbf{V}\cdot\mathbf{X})\mathbf{X}||}\ \ and\ \ c = 1 - \frac{1 - \mathbf{V}\cdot\mathbf{X}}{Ratio}$$
$$V_{map} = c\mathbf{X} + \sqrt{1-c^2}\mathbf{X}^*$$

The unit vector quantization on an half-octahedron of $\mathbf{V_{map}}$ with 4bits per coordinates results in the following integer(4bit) coefficients u, v that are packed in a single byte B :

$$q_y = \frac{\mathbf{V_{map}}y}{||\mathbf{V_{map}}||_1}\ \ and\ \ q_z = \frac{\mathbf{V_{map}}z}{||\mathbf{V_{map}}||_1}$$
$$u = \lfloor 0.5 + 7.5 * (1 + q_y + q_z) \rfloor$$
$$v = \lfloor 0.5 + 7.5 * (1 + q_y - q_z) \rfloor$$
$$B = u + (v << 4)$$

Appendix B: Uncompression (on GPU)

Operations are duals with the compression/quantization, retrieving $\mathbf{V_{map}}$ from B:

$$q_1 = \frac{B\%16}{7.5} - 1\ \ and\ \ q_2 = \frac{B >> 4}{7.5} - 1$$
$$\mathbf{W}y = \frac{q1 + q2}{2}\ \ and\ \ \mathbf{W}z = \frac{q1 - q2}{2}$$
$$\mathbf{W}x = 1 - |\mathbf{W}y| - |\mathbf{W}z|$$
$$\mathbf{V_{map}} = \frac{\mathbf{W}}{||\mathbf{W}||}$$

The next formula (already simplified) unmaps $\mathbf{V_{map}}$ to retrieve $\mathbf{V}$:

$$\mathbf{V}x = 1 - Ratio + Ratio\mathbf{V_{map}}x$$
$$\mathbf{V}y = \mathbf{V_{map}}y\sqrt{\frac{1-\mathbf{V}x^2}{1-\mathbf{V_{map}}x^2}}\ \ and\ \ \mathbf{V}z = \mathbf{V_{map}}z\sqrt{\frac{1-\mathbf{V}x^2}{1-\mathbf{V_{map}}x^2}}$$

Appendix C: Fragment shader sphere carving

To retrieve the depth of an half-sphere (of radius R) aligned with the viewing direction from a disc proxy of the same radius by manually writing the depth in each fragment, all coordinates are expressed in the camera frame (where $\mathbf{z}$ is the direction of the camera). Let

C be the center of the disc, **S** the location of the fragment on the disc, **d** the direction from the camera to the fragment and **P** the unknown location of the intersection between **d** and the half-sphere. We define $\mathbf{T} = \mathbf{S} - \mathbf{C}$, $X = ||\mathbf{T}||$, $L = ||\mathbf{P} - \mathbf{S}||$ and $D = ||\mathbf{C}||$, with $\beta = (\mathbf{P} - \mathbf{C}, \mathbf{T})$ and $\theta = (\mathbf{P} - \mathbf{S}, \mathbf{T})$. The projection of the lengths along **T** and $\mathbf{T}^{\perp}$ results in:

$$R\cos\beta = T + L\cos\theta$$
$$R\sin\beta = L\sin\theta$$

Summing the squares of those equalities results in removing β:

$$R^2 = (X + L\cos\theta)^2 + L^2\sin^2\theta$$
$$i.e.\ \ 0 = L^2 + 2L\cos\theta + X^2 - R^2$$

By construction we have also $\theta = \arctan\frac{D}{X}$, therefore when solving the equation, L is written as :

$$L = \frac{\sqrt{R^2(X^2+D^2) - D^2X^2} - X^2}{\sqrt{X^2+D^2}}$$

Since R, X, and D are accessible by the fragment, the depth of **P** can finally be simply calculated:

$$\mathbf{P}z = \mathbf{S}z + L\mathbf{d}\cdot\mathbf{z}$$

The bottom-right projection submatrix is finally used to transform $\mathbf{P}z$ in the NDC depth range.