Methodology for Real-Time Generation and Visualization of Three-Dimensional Models Using *Mesh Shaders*

Gael Rial Costas, Susana Mata, and Óscar David Robles Sánchez

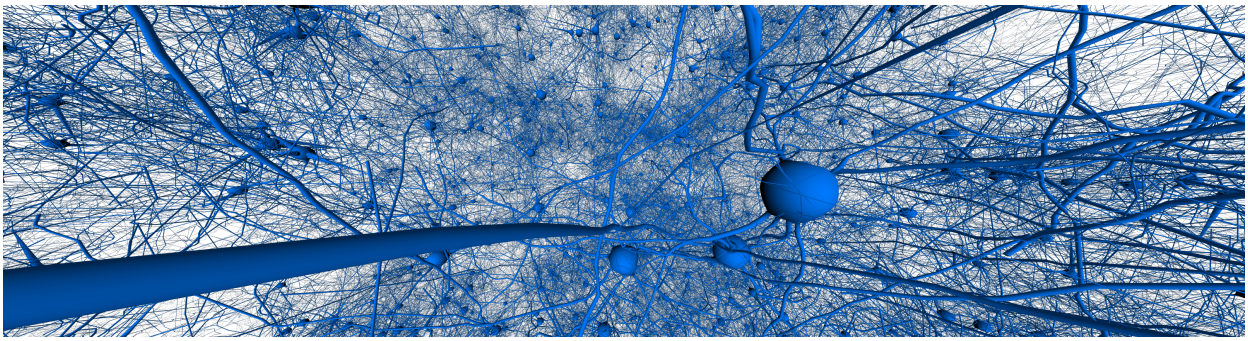

Fig. 1: A scene full of neurons

Abstract—Efficient generation and visualization of three-dimensional models based on compact representations requires the generation of high-fidelity meshes while minimizing graphics resource consumption. Current solutions often rely on precomputed meshes, which limits flexibility and leads to high memory usage, especially for dynamic or high-resolution models. This paper introduces a novel methodology for the procedural generation of complex three-dimensional geometry, taking advantage of the *Mesh Shader* pipeline to define models entirely on the *GPU* from compact representations. This approach dynamically constructs geometry each frame, enabling real-time morphological changes, aggressive culling, and adaptive levels of detail according to visual relevance. This design dramatically reduces memory consumption, scales efficiently to complex scenes, and supports instant adaptation to interactive or analytic needs. The methodology is validated through a demanding neuroscience use case involving the real-time generation of detailed neuronal morphologies, demonstrating substantial improvements in memory usage and rendering performance over existing solutions.

Index Terms—mesh shaders, visualization tools, procedural mesh generation, rendering, GPU-driven rendering, geometry compression

1 Introduction

Efficient generation and visualization of three-dimensional models based on compact representations is a key area within computer graphics, especially in scientific and medical applications where geometric accuracy and resource management are critical. This challenge is particularly acute when dealing with extremely complex, detailed, and often dynamic structures, which require advanced techniques to handle them without overwhelming the available graphics resources.

One of the main limitations faced by current techniques is the considerable consumption of graphics memory (*VRAM*). Conventional tools for mesh generation and visualization typically follow an approach that first generates a base mesh (using either *CPU* processes or hybrid methods that include *Compute Shaders* or *multi-threading*) and then transfers it to the *GPU*, where further refinement processes such as tessellation or culling are applied. This procedure causes high-definition meshes to consume an excessive amount of graphics memory, significantly limiting their applicability in scenarios involving many elements. Moreover, the reliance on a fixed base mesh severely restricts flexibility, as any substantial modification to the representation requires a complete regeneration of the mesh.

In response to these limitations, this work proposes an innovative methodology for the dynamic generation of three-dimensional models directly on the *GPU* from compact representations. This approach avoids storing pre-generated meshes, providing an effective solution for managing graphics memory consumption and enabling instant, adaptive modifications at runtime.

The main objectives of the proposed methodology are:

- **GPU-driven procedural generation**: Develop a *GPU-driven* process that enables the dynamic generation of three-dimensional meshes in each frame without the need to store explicit pre-generated geometries.
- **Real-time geometric flexibility**: Enable instant and dynamic modifications to the generated geometry, allowing rapid and effective adaptations to changes in visual conditions or specific user requirements.
- **Significant reduction of graphics memory consumption**: Utilize compact representations instead of predefined geometries, thereby minimizing graphics memory usage and efficiently managing high-density scenes.
- **Performance optimization**: Integrate advanced techniques such as *culling* (discarding non-visible geometry) and dynamic levels of detail (*LOD*) within the new methodology, enabling the adjustment of geometric complexity according to criteria such as the visual and functional relevance of scene elements.

As a result, this approach provides high performance in the visualization of complex scenes, scalability that depends primarily on the capabilities of the *GPU*, and flexibility that facilitates adaptation to different configurations and application environments.

This methodology will be validated through a specific use case in the field of neuroscience, demonstrating its ability to efficiently and accurately visualize complex neuronal morphologies. This validation

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

will allow for the assessment of both the visual quality and computational performance of the proposed system in a realistic and demanding context.

Finally, future directions are suggested to further expand and enhance this methodology. Among these, the development of hybrid representations that combine realistic and symbolic elements, the incorporation of temporal components to represent dynamic changes, support for multiple concurrent windows for complex visualizations, and the effective exploitation of configurations with multiple *GPUs* to further optimize the performance and scalability of the proposed system stand out.

2 Background

2.1 Modern Techniques for three-dimensional Mesh Generation

In recent decades, there has been a **paradigm shift** in 3D geometry generation, moving from explicit models stored vertex by vertex to symbolic and compact representations that enable geometry to be generated *on the fly*. Usually, 3D models are defined by static polygonal meshes, that is, sets of precomputed vertices, edges, and faces. This explicit approach entailed **high storage costs and difficulties** in modifying and refining geometry. In response to these limitations, the graphics community explored more abstract representations (such as continuous parametric surfaces or procedural descriptions) that describe shape in an implicit or compact way instead of enumerating every triangle.

Current strategies for generating triangular meshes are based on various representations. We can distinguish three main categories: **parametric representations** (continuous surfaces defined by mathematical functions, such as NURBS or subdivision surfaces), **implicit representations** (distance fields or occupancy functions that describe volumes), and **compact/procedural representations** (generative models, instances, geometry compression).

2.1.1 Parametric Representations: NURBS, Bézier, and Subdivision

Parametric surfaces, such as *NURBS*, **Bézier**, or subdivision surface representations, describe geometry using continuous functions. Generating a triangular mesh from them involves *sampling* these surfaces adaptively to an appropriate level of detail. In early implementations, this sampling was performed on the *CPU* or using offline algorithms; however, since the advent of programmable tessellation hardware, it is now possible to delegate this task to the *GPU*. For example, ***OpenSubdiv***, released by *Pixar* [1], established the foundations for efficient parallel evaluation of subdivision surfaces. *OpenSubdiv* provided a set of libraries capable of evaluating **Catmull-Clark** and **Bézier** patches on massively parallel *GPUs*, enabling real-time generation of smoothed meshes from mesh templates.

Various authors have explored adaptive tessellation variants for *NURBS* and trimmed surfaces. *Li et al. (2015)* [2] introduced a pioneering *GPU*-based tessellation technique for parametric surfaces that achieved **crack-free real-time rendering**, dynamically adjusting triangle density according to parameters such as curvature. For their part, *Aubry et al. (2015)* [3] proposed a mesh refinement method based on the viewpoint, converting procedural geometry into triangular meshes with fine control over the approximation error. Recently, subdivision surfaces have experienced a renaissance thanks to neural approaches. *Liu et al. (2020)* [4] presented a **neural subdivision** framework in which a neural network learns to displace the vertices generated at each subdivision step to approximate complex shapes, thus combining the compactness of a parametric rule with the expressive power of a trained model. This *Neural Subdivision* technique enables the representation of detailed geometries by training the network to add geometric details at each refinement level.

In parallel, the graphics industry has adopted parametric surfaces in practical contexts: engines such as *Unreal Engine* incorporate tessellation into their materials to generate geometry from height maps, and APIs like *DirectX* and *Vulkan* offer dedicated *Hull* and *Domain Shaders* stages to evaluate parametric surfaces. In summary, parametric representations provide a continuous description of shape and, supported by tessellation hardware and adaptive algorithms, allow for the generation of detailed triangular meshes while maintaining geometric continuity (avoiding cracks between patches) and optimizing the number of triangles according to visual criteria.

2.1.2 Implicit Representations: Distance Fields and Neural Networks

Another powerful approach are **implicit representations**, where geometry is defined not by its parametric surfaces but by a **scalar volume** or function that indicates the presence of matter. Common examples include signed distance functions (*SDF*) and occupancy functions. These representations have gained particular prominence with the advent of *Deep Learning* in graphics: instead of building meshes directly, neural networks are trained to represent shape as a continuous function that can be evaluated at any point in space. To visualize such shapes in real time, it is necessary to extract an isosurface mesh from the implicit function, a task that has traditionally been solved with algorithms such as **Marching Cubes**. However, obtaining real-time meshes from implicit models posed performance challenges that recent research has successfully addressed.

An important milestone was ***DeepSDF*** [5], which demonstrated the viability of **learning a high-quality continuous *SDF*** for a family of objects. *DeepSDF* encodes geometries into latent vectors, and an *MLP* network produces signed distances. Although mesh extraction was not its primary focus, it highlighted the need for fast methods to triangulate *SDFs*. Shortly thereafter, **Occupancy Networks** [6] introduced similar representations using occupancy functions, emphasizing geometry generation through direct sampling of the function on an adaptive 3D grid. To accelerate mesh conversion, differentiable variants and *feed-forward* strategies on the *GPU* were proposed.

The most recent advances aim to improve the **efficiency and quality of surface detection from implicit representations**, especially in the case of **unsigned distance fields (*UDFs*)**, where the lack of sign complicates surface extraction. *Stella et al. (2024)* [7] present a **Neural Surface Detection** method specifically designed for *UDFs*: a deep network locally converts a generic *UDF* into a well-behaved *SDF*, thus facilitating standard triangulation. By performing learned surface detection, the method achieves higher precision in locating the interface and can be combined with dual meshing algorithms for *UDFs*, improving performance and eliminating the need for manual parameter tuning. On the other hand, hybrid solutions have emerged that combine implicit representations with explicit hierarchical structures. An example is ***HybridSDF*** [8], which integrates basic geometric primitives (such as spheres or cubes) into the implicit representation to improve sampling efficiency. Ren et al. developed a **sparse hierarchical voxelization** scheme called ***XCube*** [9], which leverages voxel hierarchies to quickly generate large-scale meshes from dense implicit fields, demonstrating its effectiveness in synthesizing complete scenes (for example, environments with multiple objects). In general, implicit representations supported by neural networks offer enormous flexibility (enabling combinations of shape and learning) but require specialized techniques to extract meshes in real time. Recent works show that this is feasible through the combination of *deep learning* (to infer the surface) and classic computer graphics algorithms optimized on the *GPU* (for final sampling and triangulation).

2.1.3 Compact and Procedural Representations: Generative Geometry and Compression

A third approach for obtaining real-time meshes is based on **compact or procedural representations**, whether through algorithmic rules, massive instancing, or data compression. Instead of describing a continuous surface (parametric or implicit), here the geometry is defined by *generative programs, growth rules, or compressed datasets*. The goal is to exploit regularities or repetitions in the geometry to drastically reduce its effective size, and then expand it when needed for rendering.

In the field of **procedural modeling**, methods have been explored that generate cities, vegetation, or terrain using formal grammars and

stochastic algorithms. *Schinko et al. (2015)* [10] presented a comprehensive state of the art on *algorithmic modeling*, including techniques for generating urban and organic geometry through formal rules, demonstrating that it is possible to create very complex scenes from very small symbolic descriptions. These methods can produce triangular meshes in real time if the rules are evaluated on the fly. For example, modern graphics engines integrate **particle systems and procedural geometry** that generate meshes (such as foliage or volumetric particles) from random seeds and predefined patterns.

Another important field is **3D mesh compression**. Here, the idea is to store geometry in a compressed form and decompress it at visualization time. A notable advance was the ***Google Draco*** [11] library, which introduced specific algorithms for compressing meshes and point clouds, achieving very high compression ratios without significant quality loss. Draco made it possible to efficiently transmit 3D models over the web, and its decoder is fast enough to run during scene loading. *Cherchi et al. (2022)* [12] provide a thorough analysis of the current state of geometric compression techniques for 3D meshes and their integration into real-time rendering pipelines. They argue that controlled lossy compression can be effectively employed in combination with adaptive *Level of Detail* strategies, enabling the *GPU* to dynamically decode lower-resolution representations as the visual importance of objects decreases. This synergy between compression and *LOD* not only reduces memory and bandwidth usage but also improves performance without significantly compromising visual quality.

In the video game industry, the need to manage massive geometries led to hybrid solutions that combine procedural generation, *mesh streaming*, and aggressive culling. A paradigmatic example is ***Nanite***, the geometry virtualization technology introduced by Epic Games (2021) as part of the Unreal Engine 5. *Nanite* enables the loading of scenes with billions of triangles by converting static meshes into a hierarchical structure of small triangle clusters (similar to *meshlets*), which are then instanced and refined according to the camera. Although the internal details of *Nanite* are proprietary, Brian Karis [13] revealed that it uses vertex compression, *cluster culling*, and selective triangle generation within shaders. Essentially, *Nanite* can be considered a compact representation approach: instead of storing all triangles explicitly, it stores a hierarchy of patches that the *GPU* adaptively expands in real time, ensuring a fixed triangle budget per pixel. This idea of **geometry virtualization** connects with concepts such as *mega-meshes* or *out-of-core rendering* previously explored in academia, but *Nanite* demonstrated its viability in a commercial environment, managing levels of detail continuously and automatically.

In essence, compact and procedural representations aim to minimize the amount of geometric information handled directly, delegating the task of generating or refining the mesh to the moment of visualization. Whether through generative algorithms, instancing, compression, or virtualization, these techniques enable the **scaling to complex scenes** without sacrificing interactivity.

2.2 Applications with Mesh Shaders: Generation and Culling on the *GPU*

The introduction of ***Mesh Shaders*** [14] in modern APIs has revolutionized the way geometry is generated and processed on the *GPU*. *Mesh Shaders* combine the flexibility of *Compute Shaders* with the traditional functionality of primitive assembly, eliminating rigid stages of the pipeline (such as *Vertex Shaders* and *Geometry Shaders*) and allowing developers to directly define the emission of triangles via algorithms. This has led to several applications and improvements in the real-time rendering of complex geometries.

One of the first domains to benefit was **large-scale terrain rendering**. *Santerre et al. (2020)* [15] demonstrated how the mesh shading pipeline could significantly improve the tessellation of large terrains on the *GPU*. In their work, using *Mesh Shaders*, they implemented an adaptive tessellation algorithm for terrains that overcame the limitations of the fixed-function tessellation stage, achieving *continuous LOD* without cracks and with less *CPU* overhead. Independently, *Englert (2020)* [16] presented a research project that employed *Mesh Shaders* to achieve **continuous level of detail in elevation terrain visualization**. In his *SIGGRAPH 2020 talk*, Englert showed that a terrain composed of millions of triangles could be entirely managed on the *GPU*, where each terrain *meshlet* decided whether to subdivide based on the screen, resulting in smooth transitions between levels of detail and drastically reducing memory consumption compared to previous schemes.

Mesh Shaders have also been applied to **planetary rendering and massive scenes**. *Rumpelnik (2020)* [17], in his bachelor's thesis *Planetary Rendering with Mesh Shaders*, explored how to handle the visualization of an entire planet with increasing detail around the observer. By sending rectangular regions of terrain to the new pipeline (that is, using *Task Shaders* to distribute work and *Mesh Shaders* to generate the mesh), he achieved real-time planetary rendering with extremely high levels of detail around the camera, avoiding *popping* or *swimming* artifacts when updating the mesh. This work demonstrated that *Mesh Shaders* could fully replace classic *LOD* management techniques such as *quadtrees* or *clipmaps*, while providing better performance and visual quality.

In the context of stylized particle rendering, *Eriksson (2022)* [18] investigates different approaches for generating smooth *ribbons* from particle flows, combining tessellation hardware and B-splines. The work compares three implementations (based on the traditional pipeline with *Tessellation Shaders*, *Mesh Shaders*, and *Mesh and Task Shaders*), evaluating their performance and visual quality. To improve efficiency without sacrificing detail, optimizations such as *adaptive LOD*, *culling*, and *Gouraud shading* are introduced. The system enables the generation of smooth and detailed surfaces through *GPU* evaluations of tensor splines, capable of representing even cylindrical geometry with complex displacements. The results show that, although the traditional pipeline is faster at low tessellation levels, the combined use of *Mesh* and *Task Shaders* achieves higher geometric quality at a comparable cost when the proposed optimizations are applied.

Independently, ***FieldView* (2022)** [19] extended its use to mesh generation, producing isosurface meshes from a volume on the *GPU* in real time for scientific visualization, where each *meshlet* corresponded to a portion of the volume, locally evaluating a specific isovalue.

Within the graphics industry, *NVIDIA* and *AMD* have published guides and examples for optimizing the use of *Mesh Shaders*. *Kubisch (2020)* [20] from NVIDIA introduced best practices for leveraging them in professional scenarios, such as CAD and scientific visualization. Kubisch showed that dividing the scene into small *meshlets* and using *Task Shaders* for group culling could double the frame rate in CAD scenes with millions of micro-triangles. For its part, Apple incorporated support for *Mesh Shaders* in *Metal*, enabling their use on iOS/macOS platforms and demonstrating how mesh shading can be used to animate and transform large numbers of geometric instances with less *CPU* load. AMD has followed suit by integrating *Mesh Shaders* into its low-level API and presenting, on *GPUOpen* [21], use cases aimed at games: for example, fully procedural generation of camera-oriented particles (billboards) within the mesh shader, reducing *CPU-GPU* latency. Ultimately, the advent of *Mesh Shaders* has been rapidly embraced, as it offers **extreme flexibility for geometry generation**. Whether for continuous *LOD*, massive culling, procedural primitive generation, or combining compute and rasterization, this technology is redefining how we design render pipelines and promises to unify many previously disparate tricks under a single programming paradigm.

2.3 Specialized Tools for Neuronal Mesh Generation

An interesting case study that combines many of the previously discussed ideas is the **reconstruction of neuronal geometries** from morphological data. In computational neuroscience, it is common to represent neurons using skeletal descriptions (trees of segments with radii) obtained from microscopy. Efficiently generating a 3D mesh that accurately represents the entire cell membrane of a neuron is a challenge that has motivated the development of several specialized tools over the years.

One of the first approaches was ***Neuronize*** [22], which introduced a physics-based method for constructing the neuronal surface from the skeleton. In *Neuronize*, the central idea is to start from an initial shape (for example, a sphere for the soma) and then *deform* it by applying a

mass-spring model attracted by the neurites, producing a smooth mesh that realistically approximates both the soma and dendrites. Brito et al. thus succeeded not only in reproducing the visible geometry of the neuron, but also in adjusting important morphological parameters (such as the somatic volume) to match experimental data. The tool generates meshes with different fixed levels of detail per segment, allowing export of models ready for simulation or visualization.

Subsequently, ***NeuroTessMesh*** [23] brought these concepts into the field of interactive visualization. This tool combines an initial coarse mesh generation with adaptive on-the-fly refinement via *Tessellation Shaders*. First, *NeuroTessMesh* constructs a base mesh of the neuronal tree, including an approximation of the soma using a novel method based on the **finite element method (*FEM*)** to deform an initial sphere. Then, during visualization, it uses *GPU* tessellation to dynamically subdivide parts of the neuron as needed (for example, to inspect fine branches up close), generating additional geometry only when necessary. This made it possible to visualize **complete neuronal scenes** at good detail and in real time, something difficult to achieve with static meshes due to the immense number of triangles required to discretize all neurons at maximum resolution. Garcia-Cantero et al. highlighted how adaptive refinement on the *GPU* kept memory and rendering time costs reasonable, integrating their tool into scientific workflows such as the visual exploration of neuronal circuits.

Simultaneously, tools emerged focused on integration with simulation and editing platforms. ***NeuroMorphoVis*** [24] was introduced as a collaborative and extensible framework for the **editing and visualization** of neuronal morphologies. Although *NeuroMorphoVis* focuses on skeleton visualization (the central lines of neurons), it also incorporated functions to generate surfaces, acting as a bridge between raw neuronal tracing data and the meshes required for biophysical simulations or advanced rendering. Abdellah et al. implemented their tool as a Blender add-on, leveraging its rendering capabilities, and emphasized its interactive and cross-platform nature, making it easier for neuroanatomists to inspect reconstructed neuronal geometry.

Recently, the efforts of the *Blue Brain Project* in collaboration with other institutes have led to ***Ultraliser*** [25], a comprehensive *framework* for generating high-fidelity 3D neuronal models at multiple scales. *Ultraliser* addresses the challenge of creating bio-realistic models by integrating neurons, glial cells, and cerebral vasculature, all from sparse and incomplete data. For neurons, *Ultraliser* combines implicit and explicit methods: it uses intermediate volumetric representations and robust **mesh repair algorithms** to ensure that the resulting meshes are **watertight surfaces without self-intersections**, suitable for finite element simulations and other demanding applications. Abdellah et al. describe *Ultraliser* as a "multiscale, realistic 3D modeling framework for in silico neuroscience," emphasizing its ability to handle large neuron datasets while maintaining subcellular detail (e.g., dendritic spines). The result is optimized, watertight meshes that preserve the original morphological measurements. *Ultraliser* has been released as open-source software and is complemented by a dataset on *Zenodo* to validate the quality of the generated meshes.

Pioneering tools from the Blue Brain Project such as ***RTNeuron*** [26] should also be mentioned. *RTNeuron* was initially conceived as an application for visualizing the results of detailed neuronal simulations, but evolved into an interactive framework for rendering complete circuits. Although *RTNeuron* did not generate meshes from scratch (rather, it used preconstructed geometries) it did implement advanced instancing and level-of-detail techniques to render hundreds of thousands of wired neurons in real time. Its open-source release in 2018 enabled the community to analyze parallel visualization strategies for neurons, such as using instances of cylinders for neural segments and visual substitution via *shaders* to achieve an organic appearance without multiplying the geometry. This philosophy of "domain-specific visualization" was a precursor to ideas now incorporated into more recent frameworks.

Finally, it is relevant to mention ***LiveMesh*** [27], a project developed at École Polytechnique Fédérale de Lausanne focused on *GPU*-based generation of neuronal membranes from morphological descriptions. *LiveMesh* explored the use of the then-new tessellation unit in *GPUs* to directly convert each neuronal segment into a smooth triangular tube, connecting these segments with the somatic portion (even if very simple per segment), which the *GPU* dynamically subdivides.

The use of *tessellation shaders* made it possible to delegate to the *GPU* the creation of new, detailed geometry from a base shape. In the neuronal context, works such as *NeuroTessMesh* and *LiveMesh* demonstrated that adaptive tessellation could produce smooth surfaces for each neurite depending on the observer's proximity, all at render time. This approach reduced *CPU* load (which only needed to provide the base geometry, typically generated as a *Bézier* patch). The prototype demonstrated the **feasibility of *GPU* tessellation for neurons**, achieving continuous and visually high-fidelity representations of complete cells, with the advantage of dynamically adjusting the level of detail. This idea from *LiveMesh* (generating neurons "on the fly" on the graphics card) establishes the conceptual foundations that tools like *NeuroTessMesh* and *Ultraliser* later expanded upon with more sophisticated approaches.

Taken together, these specialized tools demonstrate how the combination of mesh generation techniques with accelerated computing platforms enables tackling a complex problem: reconstructing and visualizing **complex biological structures in real time**. Each contributed particular innovations (mass-spring, adaptive tessellation, implicit methods, etc.), but all share the goal of achieving precise and optimized triangular meshes from compact data (traces or volumetric representations), relying on the general advances previously described in this state of the art. The evolution from *Neuronize* to *Ultraliser* over the span of a decade mirrors, in parallel, the evolution in computer graphics: from generating geometry with specific algorithms on the *CPU*, to delegating much of the work to programmable *GPUs* with mesh shading and massive compute capabilities, even integrating artificial intelligence into the process. This multidisciplinary progress has brought neuronal visualization (and many other generative geometry applications) to a new level, making it possible to interact in real time with models that were previously prohibitively complex.

3 METHODS

3.1 Algorithmic Design Principles

To fulfill the high-level objectives outlined in the introduction, the core generation algorithm is built upon several fundamental design principles. These principles ensure that the resulting process is not only fast and efficient but also flexible and scalable.

- **Principle 1: High-Degree of Parallelism.** The geometry generation process must be "embarrassingly parallel". This dictates that the computation for each output element (vertex or primitive) should be as independent as possible, allowing the workload to be distributed efficiently across the thousands of threads on a modern *GPU*. This principle is the primary enabler of the **GPU-driven procedural generation** objective, allowing the system to scale with the *GPU*'s computational power.
- **Principle 2: Efficient Use of the Memory Hierarchy.** The algorithm must prioritize the use of the fastest tiers of *GPU* memory. Calculations should leverage the low-latency **shared memory** available within a workgroup to exchange data between threads, minimizing costly reads from and writes to global *VRAM*. By minimizing traffic to global *VRAM*, this principle contributes to both the **GPU-driven procedural generation** and **reduced memory consumption** objectives.
- **Principle 3: Decoupled Procedural Logic.** The algorithm must be built around a procedural abstraction that separates the generation *logic* from the input *data*. Instead of processing raw geometric data, the algorithm should operate on high-level "blueprints" that define *how* to generate a shape. This abstraction is the cornerstone for achieving **GPU-driven generation**, **morphological flexibility** (by parameterizing the blueprints), and **reduced memory consumption** (by ensuring the blueprints are compact).
- **Principle 4: Native Integration of Optimizations.** Performance-enhancing techniques such as frustum culling and Level of Detail (*LOD*) must not be separate, post-generation steps. Instead, they

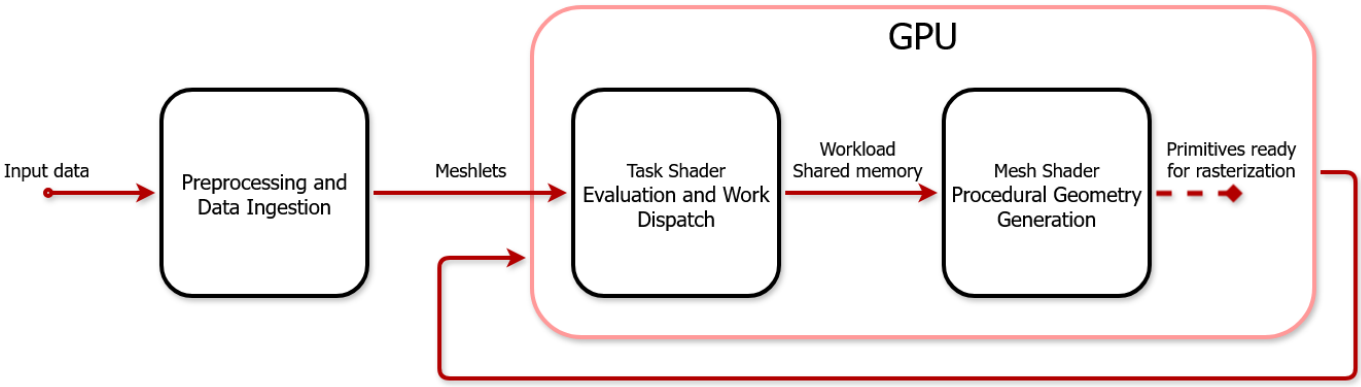

Fig. 2: Diagram of the generation pipeline architecture, showing the flow from data ingestion to the evaluation and geometry generation.

should be natively integrated into the generation algorithm itself. This proactive approach to optimization is a direct implementation of the **performance optimization** objective, ensuring that computational resources are never wasted on irrelevant geometry.

These four principles define a clear architectural path: a system that transforms a lightweight, symbolic description of a scene into a detailed, optimized 3D mesh entirely within the *GPU* pipeline. The following sections describe the concrete architecture designed to fulfill these principles.

3.2 Hardware and API Constraints

The design of any algorithm for the modern graphics pipeline is fundamentally shaped by the opportunities and constraints of the underlying hardware and API. For this methodology, the *Mesh Shader* pipeline provides a powerful but structured environment for procedural generation.

The pipeline offers great flexibility, allowing for the dispatch of a virtually unlimited number of workgroups. However, each individual workgroup operates within a strict set of resource limits. On a typical NVIDIA architecture, for example, a workgroup is composed of a fixed number of 32 threads (aligning with the idea of a *warp*), has access to a limited pool of high-speed shared memory (usually 32 KB), and can generate a maximum number of vertices and primitives (256 of each).

These hardware constraints are the primary motivation for the core architectural decisions in this methodology. It is not feasible to generate a whole, complex object within a single workgroup. Therefore, any solution must be based on a **divide and conquer** strategy, breaking down large, complex objects into smaller, independent pieces that a single workgroup can process within its budget. This directly leads to the procedural abstraction detailed in the following section.

Furthermore, the API provides the *Task Shader*, a preceding compute stage, which allows for dynamic, data-driven dispatch of *Mesh Shader* workgroups. This feature is the key to implementing the principle of native optimization, as it provides the mechanism for culling and refining work before any geometry generation is attempted.

3.3 The Schemlet: A Blueprint for Procedural Geometry

To implement the principle of Decoupled Procedural Logic, this work introduces a novel abstraction called the ***schemlet***. The *schemlet* is the concrete data structure that acts as the "blueprint" for geometry generation. It is the fundamental unit of work that is processed by the proposed pipeline.

Unlike a traditional *meshlet* (a small, data-centric and self-contained patch of explicit vertices and indices) a *schemlet* is logic-centric. It contains no geometric data itself. Instead, it encapsulates the symbolic parameters required to generate a piece of geometry on the fly. For instance, a *schemlet* might store high-level parameters such as the start and end points of a cylinder, its radius, and its radial resolution. Another *schemlet* could define a branching point with pointers to child structures, while another could describe a parametric surface via its control points. This distinction is critical:

- A **meshlet** is a unit of *geometry*. It is optimized for rendering existing, static meshes more efficiently.
- A **schemlet** is a unit of *work*. It is a command packet for a *Mesh Shader* workgroup, optimized for generating new, dynamic geometry from a compact description.

By representing geometry symbolically, *schemlets* are the key to achieving the objectives of reduced memory consumption and real-time morphological flexibility.

3.3.1 Data Structure vs. Generation Logic

It is critical to clarify that a *schemlet* itself contains no executable code. It is a plain, passive data structure. The "rules" and "logic" for generation reside on the *GPU* as a set of polymorphic *Mesh Shader* programs. Each shader is specialized in interpreting a specific type of *schemlet*.

3.3.2 Lifecycle of a Schemlet

In this methodology, a *schemlet* has a clear lifecycle. It is (1) **created** during an offline CPU preprocessing step, where source data is converted into this compact format. It then (2) **resides** in a large buffer in *GPU* memory for the duration of the session. In each frame, it is (3) **evaluated** by a *Task Shader*. If the *schemlet* wasn't discarded, it is (4) **consumed** by the corresponding *Mesh Shader* to generate geometry. Finally, at the end of the frame, the generated geometry is discarded, and the *schemlet* in the buffer awaits evaluation in the next frame. A simplified representation of this lifecycle can be seen in Figure 2.

3.4 Architecture of the Generation Pipeline

The lifecycle of a schemlet, described above, is implemented through a multi-stage pipeline that transforms a high-level scene description into a renderable mesh entirely on the *GPU*. The process begins with an offline preparation step and is followed by a series of programmable shader stages that execute for every frame.

3.4.1 Stage 1: Preprocessing and Data Ingestion (*CPU*)

The pipeline begins with a one-time, offline preprocessing step performed on the *CPU*. This stage is responsible for "ingesting" the source data (for example, a file describing a neuronal morphology) and converting it into a *GPU-friendly* format. Its primary task is to decompose the source object into a list of the aforementioned *schemlets*. During this decomposition, it can also pre-calculate any static data that would be expensive to compute in real-time on the *GPU*. Examples include optimizing path trajectories, calculating initial importance values for *LOD*, or solving for static intersection angles in branching structures. Once this process is complete, the resulting collection of *schemlets* and any other required data are packed and transferred to *VRAM*.

3.4.2 Stage 2: Evaluation and Work Dispatch (*Task Shader*)

The real-time portion of the pipeline begins each frame with the *Task Shader*. This stage acts as a dynamic filter and work dispatcher. It is launched with a workload corresponding to the full set of *schemlets* for a scene. Each workgroup in the *Task Shader* evaluates a set of *schemlets* against a set of criteria. This stage provides the ideal place to implement optional, on-the-fly optimizations, directly realizing the "Native Integration of Optimizations" principle. Common implementations include:

- **Culling:** *Schemlets* are tested for visibility against the camera frustum. Occlusion culling could also be performed at this stage. If a *schemlet* is determined to be not visible, it is discarded, and no further work is done on it for this frame.
- **Level of Detail (*LOD*):** For visible *schemlets*, a level of detail is calculated, typically based on distance to the camera or screen-space size. This *LOD* value is then passed along to the next stage.

Based on this evaluation, the *Task Shader* dynamically determines how many *Mesh Shader* workgroups are needed and dispatches them, effectively telling the *GPU* to only generate geometry for the objects that are actually visible and relevant.

3.4.3 Stage 3: Procedural Geometry Generation (*Mesh Shader*)

The final stage is the *Mesh Shader*, where the actual geometry is created. Each dispatched *Mesh Shader* workgroup receives a single *schemlet* as its input. Using the procedural rules and parameters contained within the *schemlet*, combined with the information received from the *Task Shader* and other global parameters, the threads within the workgroup collaborate to generate the vertices and primitives that form the corresponding mesh patch.

Following the "High-Degree of Parallelism" principle, the work is distributed among the threads of the workgroup. While all threads execute the same shader code to ensure lock-step execution and prevent divergence, they can collaborate on complex calculations by using the workgroup's shared memory as a high-speed scratchpad memory. A common pattern is for each thread to compute a piece of data, write it to a known location in shared memory, and then synchronize with the other threads. After synchronization, any thread can read the data computed by its peers. For example, to generate a smooth, continuous surface, the position of a vertex might depend on the positions of its neighbors. Instead of re-computing these neighboring positions, a thread can simply read them from shared memory after they have been calculated and stored there by other threads in the same workgroup. This collaborative, parallel process allows the workgroup to generate the final geometry without ever accessing an explicit, pre-stored mesh, completing the on-the-fly generation process.

4 Use Case: Real-Time Neuronal Visualization

To validate the proposed methodology, a use case has been developed focusing on the **real-time generation and visualization of detailed neuronal morphologies**. This application, named *Neoneuron*, serves as an ideal testbed. Neuronal structures, with their complex, branching morphologies and high geometric detail, present a significant challenge for traditional rendering techniques and thus provide a demanding environment to assess the performance and flexibility of the proposed approach.

This section details how the abstract principles and pipeline architecture described in Section 3 are concretely applied to transform a compact, skeletal description of a neuron into a detailed, continuous three-dimensional mesh entirely on the fly. The core of the implementation lies in mapping the biological components of a neuron to the procedural abstractions of the pipeline. This process is not merely a data conversion, but a direct application of the design principles established in Section 3.1. The complex, hierarchical structure of a neuron is systematically deconstructed into a flat collection of *schemlets*, each serving as a self-contained blueprint for a piece of geometry. This ensures the entire process is parallelizable, memory-efficient, and procedurally driven.

4.1 Stage 1: Ingestion and Decomposition (CPU)

Following the architecture laid out in Section 3.4.1, the process begins with a one-time, CPU-based preprocessing step. A standard neuronal morphology file (e.g., in SWC format), which describes the neuron as a tree of points, is ingested. This skeletal data is then converted into a flat list of specialized *schemlets*.

This act of decomposition is a direct embodiment of the **Decoupled Procedural Logic** principle. Instead of a monolithic mesh, a list of symbolic command packets is created. This approach also fulfills the **Efficient Use of the Memory Hierarchy** principle, as this compact list of *schemlets* is the sole representation of the neuron's geometry stored in VRAM, drastically reducing the memory footprint.

Three distinct types of *schemlets* are defined to represent the entire neuronal structure:

- **Segment Schemlet**: The fundamental unit, representing a section of a dendrite or axon. It contains the start and end points, their respective radii, and hierarchical information. This provides the *Mesh Shader* with all the necessary parameters to procedurally generate a cylindrical tube. Furthermore, a point importance value is pre-calculated using a modified *Ramer-Douglas-Peucker* algorithm and stored within the schemlet. This allows the Task Shader to later perform efficient, data-driven LOD culling.
- **Bifurcation Schemlet**: Represents a point where a neurite branches. It stores references to the parent and two child segments. To ensure a seamless geometric join, an optimal division angle is pre-calculated on the CPU and stored within the schemlet, offloading complex calculations from the real-time pipeline.
- **Soma Schemlet**: Represents the cell body. It stores the soma's central position, its radius, and pointers to the initial neurites that connect to it. This schemlet acts as the root of the structure and contains the parameters for generating a deformable sphere.

4.2 Stage 2: Evaluation and Work Dispatch (Task Shader)

Following the architecture from Section 3.4.2, the real-time pipeline **begins** with the Task Shader. This stage **functions** as the primary dynamic filter and work dispatcher for *Neoneuron*, **embodying Principle 4: Native Integration of Optimizations** by applying visibility and LOD criteria *before* any geometry is generated.

The stage is launched with a workload covering the entire *schemlet* list residing in VRAM. Each *schemlet* then undergoes a two-phase evaluation process within the Task Shader workgroups.

First, a set of **common evaluations** is applied **universally** to every *schemlet*, regardless of its type:

- **Visibility Culling**: The parameters within each *schemlet* (e.g., control point positions and radii) are used to compute a bounding box, **which is then** tested against the camera frustum. If a *schemlet* is not visible, it is discarded, and no further work is done on it for this frame.
- **LOD Value Calculation**: For all visible *schemlets*, a detail level is calculated, typically based on the distance to the camera. This LOD value is a normalized parameter that will be interpreted by the specialized generation algorithms in the next stage.

Second, **specialized evaluations** are performed **based on the type** of each surviving *schemlet*, **determining** the precise workload to dispatch to the Mesh Shader stage.

- For a **Segment Schemlet**, the Task Shader applies a **two-fold LOD process**: It uses the calculated LOD value to determine the segment's radial vertex density (e.g., from 4 to 16 vertices per circle). Concurrently, it evaluates the control point's importance using the pre-calculated *Ramer-Douglas-Peucker* value from Stage 1. If a control point is deemed omissible at the current LOD, **no workgroups are dispatched**. Otherwise, **parental linkages are recomputed** to connect the segment to its nearest

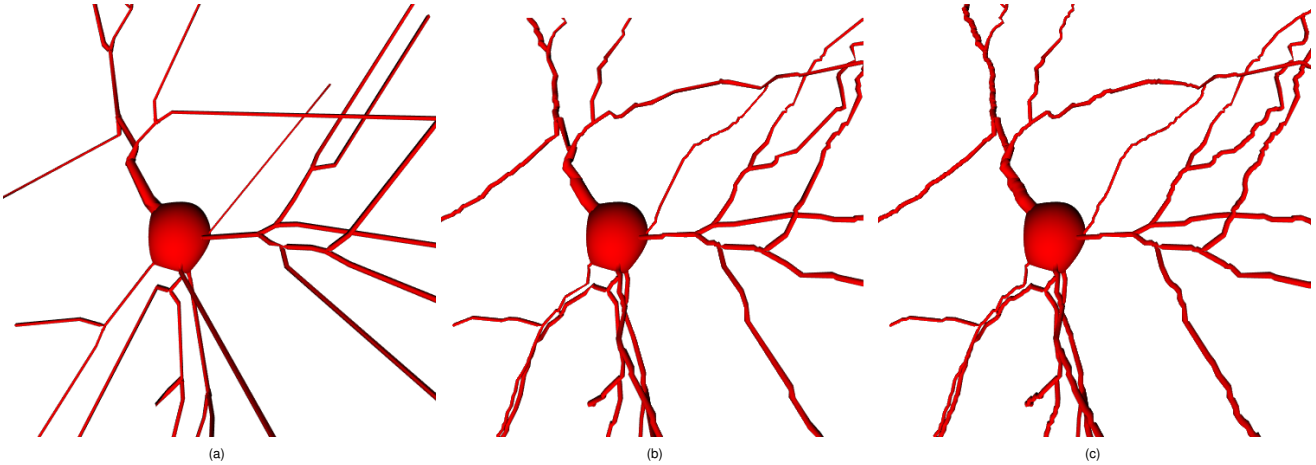

Fig. 3: Neuron generation using several *LOD* levels. 3a Minimum level. 3b Medium level. 3c Maximum level.

visible ancestor, and **one** Mesh Shader workgroup is dispatched, passing it the *schemlet* data and the computed vertex density. The result of this approach can be seen in Figure 3.

- For a **Bifurcation Schemlet**, the evaluation **focuses on geometric** continuity. The LOD value is used to determine the vertex density for all connecting circles to ensure a seamless, crack-free fit. It then dispatches a single Mesh Shader workgroup, passing along the *schemlet*'s pre-computed optimal division angle.
- For a **Soma Schemlet**, the **evaluation is the most sophisticated**. The Task Shader first uses the LOD value to determine the base tessellation of the soma's sphere (e.g., 16x16 or 64x64, as shown in Figure 4). It **then** performs the necessary intersection tests, iterating over the soma's base primitives to **locate** the connection points for its child segments. Based on these results, it dispatches a **variable number** of Mesh Shader workgroups: one (or more) to generate the main deformed sphere, plus additional, specialized workgroups to procedurally generate each connection mesh.

4.3 Stage 3: Procedural Geometry Generation (Mesh Shader)

The final stage of the pipeline executes the Mesh Shader workgroups dispatched by the Task Shader. This stage is the concrete implementation of **Principle 1 (High-Degree of Parallelism)** and **Principle 2 (Efficient Use of the Memory Hierarchy)**.

Each workgroup is responsible for generating the full geometry for a single *schemlet*. Threads within the workgroup collaborate extensively, using workgroup **shared memory** to broadcast *schemlet* parameters and share computed vertex positions. This prevents redundant calculations and minimizes costly access to global VRAM. For example, when generating a segment, the start and end circle vertices are computed once, stored in shared memory, and then accessed by all threads to construct the triangles of the tube.

The generation logic is polymorphic, executing a specific procedural algorithm based on the *schemlet* type:

- **Segment Schemlet**: The workgroup reads the start/end points, radii, and the LOD-defined vertex density (from Stage 2). It procedurally generates two circles of vertices, oriented correctly in 3D space. It then weaves a triangle strip between them to form the cylindrical segment.
- **Bifurcation Schemlet**: This workgroup receives the bifurcation *schemlet* and its corresponding LOD density. It calculates the vertex positions for the three connecting circles (parent, two children) using the same density to ensure no gaps. As shown in Figure 5, the pre-calculated optimal angle from Stage 1 is used to seamlessly suture the three segments together, generating the complex mesh that forms the join.
- **Soma Schemlet**: This is the most complex generation task. The primary workgroup generates a tessellated sphere based on the LOD value (e.g., 16x16 or 64x64). A deformation algorithm, **conceptually similar to skeletal animation skinning**, is then applied. Each vertex of the base sphere calculates its final position based on **weighted influences** from the set of neurite connection points, which effectively act as a "skeleton". This allows the connection points to slightly pull and deform the sphere's geometry, creating a more organic shape. Concurrently, the specialized "connection mesh" workgroups (dispatched in Stage 2) **focus exclusively on generating the "stitching" mesh**, connecting the base of the child segment to the edge of the hole left by the

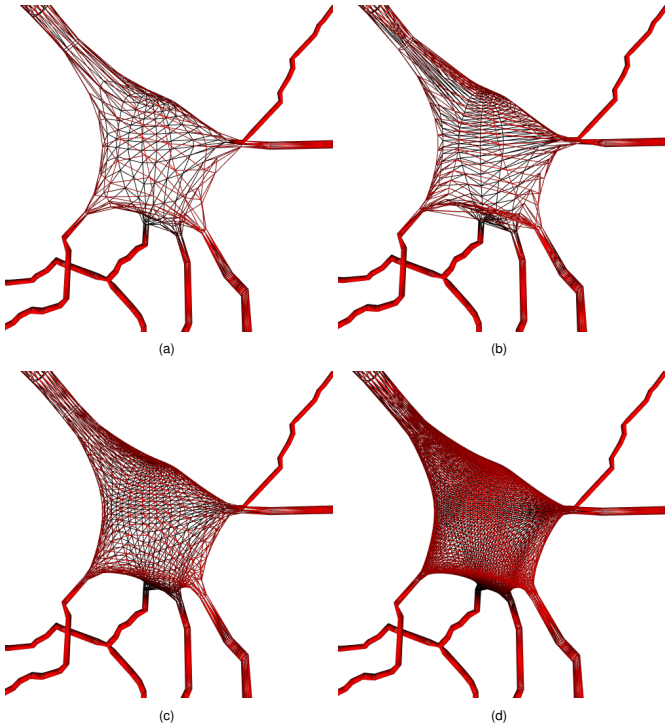

Fig. 4: Soma morphologies using different vertex density configuration. 4a 16x16 vertices. 4b 10x32 vertices. 4c 32x32 vertices. 4d 64x64 vertices.

primary workgroup, ensuring a continuous surface.

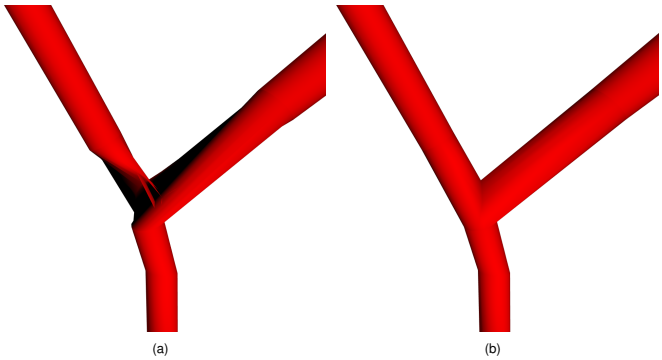

Fig. 5: Result of the proposed bifurcation generation algorithm. 5a without angle optimization. 5b with angle optimization.

This parallel, collaborative, and polymorphic generation process completes the transformation from a compact symbolic representation to a detailed, render-ready mesh in a single pass, entirely on the *GPU*.

5 Results

This article presents a methodology for the procedural generation and adaptive visualization of three-dimensional models using *Mesh Shaders*. To validate the fulfillment of the four stated objectives, a series of specific tests and *benchmarks* have been carried out.

5.1 Morphological Flexibility

The first two objectives of this work are to develop a fully *GPU*-driven generation that eliminates the need to store explicit geometries and to enable rapid and effective adaptations to changes in visual conditions or specific user requirements. These objectives are closely related: by generating the required geometry in each frame, it is possible to adjust the generator's parameters and have those changes applied instantly.

In the proposed use case, the generator is sensitive to various parameters that allow altering the resulting geometry. Some of these parameters include: camera position, the relative radius of segments and soma, the space occupied by bifurcations, and the influence strength of the weights deforming the soma.

During execution of the use case, this adaptive capability is clearly evident. When parameters are changed, modifications are reflected instantaneously, without delays or degradation in frame rate. This immediate response contrasts with techniques based on pre-stored geometries, where significant morphological changes require regeneration times or recompilation of graphics data.

5.2 Memory Usage Reduction

One of the main objectives of the proposed methodology is to minimize graphics memory consumption (*VRAM*). In previous approaches, especially those requiring explicit storage of complete geometries, graphics memory quickly becomes a limiting factor when dealing with highly complex scenes or multiple detailed objects. This issue is further exacerbated in scientific, medical, or large-scale visualizations, where the need to simultaneously represent hundreds or thousands of detailed three-dimensional models places a considerable burden on the available graphics resources.

To validate the reduction in graphics memory consumption (*VRAM*), multiple comparative tests have been conducted against existing tools that pre-generate meshes. Specifically, the memory usage of individual neuronal morphologies was measured using *NeuroTessMesh* and *Neuronize*, and these results were compared with the consumption of *Neoneuron*. Additionally, the morphologies generated by *Neoneuron* were exported, allowing for an evaluation of *VRAM* consumption in the use case versus a hypothetical scenario in which morphologies are managed as standard three-dimensional models.

The quantitative comparison of graphics memory consumption (*VRAM*) shows notably favorable results for the proposed methodology. As shown in Figure 8, the average *VRAM* consumption compared to *Neoneuron* is **6.3 times** higher in *NeuroTessMesh*, **38.5 times** higher in *Neuronize*, and **19 times** higher when using the models exported directly from *Neoneuron*, managing them as traditional meshes (*Neoneuron OBJ*. These results confirm that the approach based on *schemlets* enables substantial reductions in memory consumption, greatly facilitating the efficient management of complex and highly detailed scenes. This efficiency in the use of graphics resources represents an advantage over the other evaluated techniques, enabling the creation of interactive visualizations where the number and complexity of visualized elements would be prohibitive with other approaches.

These average values are further contextualized in Figure 6, which shows the distribution of *VRAM* consumption for each evaluated tool. The plot demonstrates that *Neoneuron* not only achieves the lowest mean value but also exhibits less dispersion, indicating much more predictable and stable memory usage. In contrast, tools like *Neuronize* display greater variability and significantly higher maximum values, reaching up to 15 MiB per instance in some cases. Similarly, models exported from *Neoneuron* when managed as traditional meshes, show a notable increase in memory usage compared to the procedural version, further reinforcing the advantage of avoiding explicit geometry storage.

5.3 Performance

Closely related to *GPU-driven procedural generation*, another objective of this work is performance optimization through techniques such as culling and dynamic levels of detail. These two objectives are closely linked, since eliminating *CPU-GPU* transfers while dynamically managing geometric complexity should lead to improved efficiency and smoother graphics applications.

To evaluate this performance improvement, multiple comparative tests have been conducted between the proposed use case (*Neoneuron*) and the tool *NeuroTessMesh*. Specifically, graphics performance was measured in terms of frames per second (*FPS*) when rendering different scenes with numerous detailed neuronal morphologies. These tests were designed to ensure that both applications visualize exactly the same scenes in each iteration, thus guaranteeing a fair and objective comparison.

As shown in Figure 9, the obtained results demonstrate a substantial improvement in performance when using the proposed methodology. Specifically, *Neoneuron* achieves an average FPS more than twice as high as that recorded by *NeuroTessMesh* in the same scenes. This significant improvement is mainly attributed to the fully *GPU-driven* generation combined with the advanced optimization techniques implemented in the methodology, which leverage the new graphics pipeline to more aggressively adapt the graphics workload according to the conditions of each frame.

The tests also showed that *Neoneuron* makes intensive use of the *GPU*'s capabilities, to the point of saturating its computational resources. This intensive utilization is even reflected in a perceptible deterioration of the general performance of the operating system's graphics environment. However, despite this high *GPU* load, *Neoneuron* maintains a frame rate that doubles that obtained with *NeuroTessMesh*, which clearly demonstrates superior efficiency in the use of available graphics resources.

6 Conclusions and Future Work

This work presents a methodology for the real-time generation and visualization of three-dimensional geometry using a pipeline based on *Mesh Shaders*. Unlike traditional approaches that require storing explicit meshes in memory, the proposed technique generates geometry dynamically in each frame from compact descriptions, allowing for minimized graphics memory usage and efficient adaptation to the dynamic conditions of the scene.

The modular architecture of the system, structured around the concept of *schemlets*, enables flexible, scalable, and parallelizable geometric construction, fully leveraging the computational capabilities of

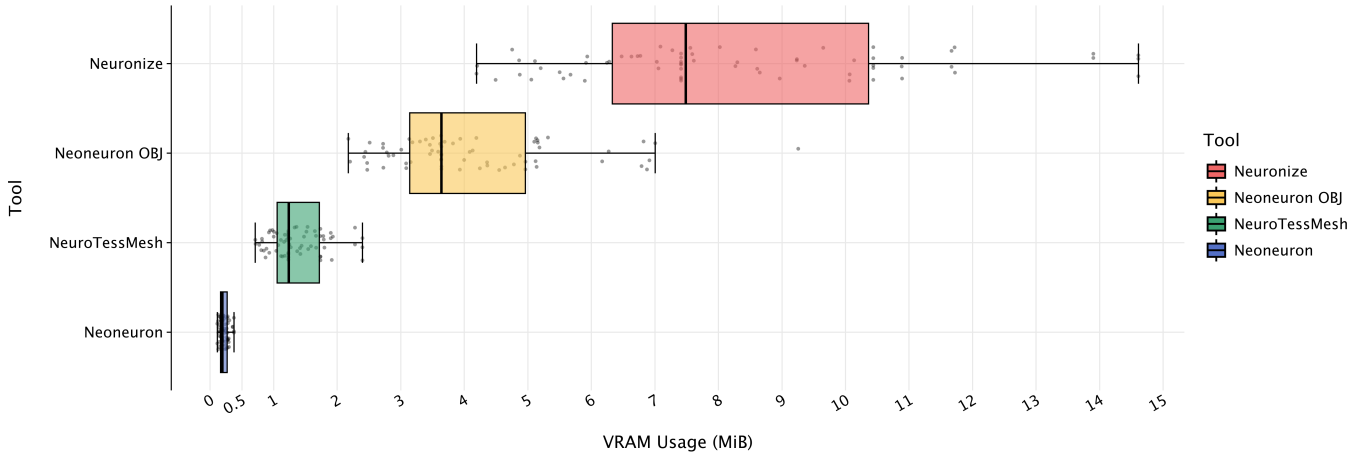

Fig. 6: Distribution of *VRAM* memory used per neuron by application

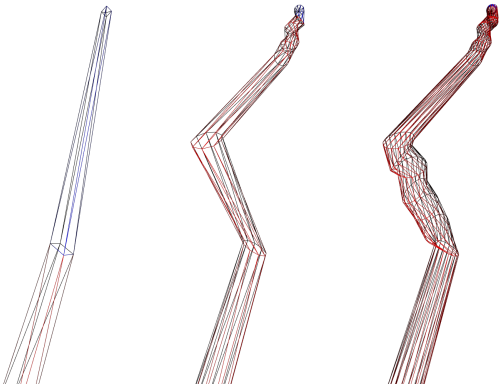

Fig. 7: Segment generation using different *LOD* levels.

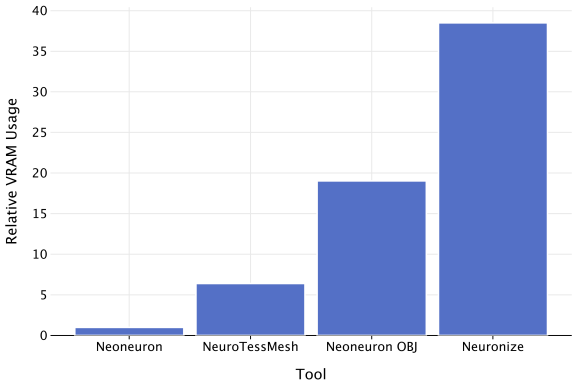

Fig. 8: Relative *VRAM* memory usage compared to other solutions

modern graphics hardware. Thanks to its staged design and the strategic use of *Task Shaders* and *Mesh Shaders*, the methodology enables fine-grained control over visible geometric complexity, facilitating the integration of techniques such as *frustum culling* or *Level of Detail* without compromising performance.

The experimental results obtained through the use case show a significant reduction in graphics memory consumption compared to existing tools, as well as a notable improvement in performance. Moreover, the use of *schemlets* allows for instantaneous morphological flexibility without relying on auxiliary meshes, enabling new scenarios for interaction and visual exploration that would be impractical with other approaches.

Regarding future work, the modular system architecture allows to improve the solution in several directions. First, the methodology could be extended to support **multiple simultaneous visualizations**, enabling the analysis and comparison of different scenes or perspectives in real time. Additionally, effective **parallelization in *multi-GPU* configurations** is envisaged, which would open the door to interactive visualization of even more complex scenes and much greater scalability in high-performance environments. Both lines of work would contribute to consolidating the proposed methodology as a versatile and robust solution for the visual exploration of three-dimensional models in demanding scenarios.

Acknowledgments

Funded by the European Union under Grant Agreements 101137289 (*Virtual Brain Twin*) and 101147319 (*EBRAINS 2.0*). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Pixar Animation Studios, "OpenSubdiv: High performance subdivision libraries," Accessed on Pixar Graphics Website, 2014, http://graphics.pixar.com/opensubdiv. 2

[2] Y. Li, X. Lu, W. Zhang, and G. Wang, *Adaptive NURBS Tessellation on GPU*. Singapore: Springer Singapore, 2015, pp. 69–84. [Online]. Available: https://doi.org/10.1007/978-981-287-134-3_5 2

[3] R. Aubry, S. Dey, E. Mestreau, B. Karamete, and D. Gayman, "A robust conforming nurbs tessellation for industrial applications based on a mesh generation approach," *Computer-Aided Design*, vol. 63, pp. 26–38, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010448515000032 2

[4] H.-T. D. Liu, V. G. Kim, S. Chaudhuri, N. Aigerman, and A. Jacobson, "Neural subdivision," *ACM Trans. Graph.*, vol. 39, no. 4, 2020. 2

[5] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, "Deepsdf: Learning continuous signed distance functions for shape representation," 2019. [Online]. Available: https://arxiv.org/abs/1901.05103 2

[6] L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger, "Occupancy networks: Learning 3d reconstruction in function space," 2019. [Online]. Available: https://arxiv.org/abs/1812.03828 2

[7] F. Stella, N. Talabot, H. Le, and P. Fua, "Neural surface detection for unsigned distance fields," 2024. [Online]. Available: https://arxiv.org/abs/2407.18381 2

[8] S. Vasu, N. Talabot, A. Lukoianov, P. Baqué, J. Donier, and P. Fua, "Hybridsdf: Combining deep implicit shapes and geometric primitives for 3d shape representation and manipulation," 2022. [Online]. Available: https://arxiv.org/abs/2109.10767 2

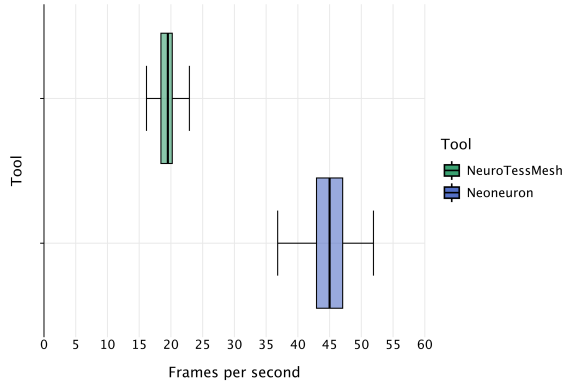

Fig. 9: Distribution of average frame rates for scene visualization by application

[9] X. Ren, J. Huang, X. Zeng, K. Museth, S. Fidler, and F. Williams, "Xcube: Large-scale 3d generative modeling using sparse voxel hierarchies," 2024. [Online]. Available: https://arxiv.org/abs/2312.03806 2

[10] C. Schinko, U. Krispel, and T. Ullrich, "Know the rules – tutorial on procedural modeling," 03 2015. 3

[11] J. de Hoog, A. N. Ahmed, A. Anwar, S. Latré, and P. Hellinckx, "Quality-aware compression of point clouds with google draco," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, Ed. Cham: Springer International Publishing, 2022, pp. 227–236. 3

[12] G. Cherchi, F. Pellacini, M. Attene, and M. Livesu, "Interactive and robust mesh booleans," 2022. [Online]. Available: https://arxiv.org/abs/2205.14151 3

[13] B. Karis, "Nanite: A deep dive," in *SIGGRAPH 2021: Advances in Real-Time Rendering in Games*, 2021. [Online]. Available: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf 3

[14] NVIDIA Corporation, "Nvidia turing gpu architecture: Graphics reinvented," https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018, whitepaper. 3

[15] B. Santerre, M. Abe, and T. Watanabe, "Improving gpu real-time wide terrain tessellation using the new mesh shader pipeline," in *2020 Nicograph International (NicoInt)*, 2020, pp. 86–89. 3

[16] M. Englert, "Using mesh shaders for continuous level-of-detail terrain rendering," in *ACM SIGGRAPH 2020 Talks*, ser. SIGGRAPH '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388767.3407391 3

[17] M. Rumpelnik, "Planetary rendering with mesh shaders," Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Feb. 2020. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2020/rumpelnik_martin_2020_PRM/ 3

[18] O. Eriksson, "Smooth particle ribbons through hardware accelerated tessellation," 2022. 3

[19] P. Nowakowski and P. Rokita, "Fieldview: An interactive software tool for exploration of three-dimensional vector fields," *SoftwareX*, vol. 22, p. 101406, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711023001024 3

[20] C. Kubisch, "Introduction to turing mesh shaders," NVIDIA Technical Blog, Mar 2020. [Online]. Available: https://developer.nvidia.com/blog 3

[21] AMD, "Mesh shaders + work graphs: A perfect pair," https://gpuopen.com/presentations/2024/Mesh_Shaders_Work_Graphs-Perfect_Pair.pdf, 2024, presentation slides. 3

[22] J. P. Brito, S. Mata, S. Bayona, L. Pastor, J. DeFelipe, and R. Benavides-Piccione, "Neuronize: a tool for building realistic neuronal cell morphologies," *Frontiers in neuroanatomy*, vol. 7, p. 15, 2013. 3

[23] J. J. Garcia-Cantero, J. P. Brito, S. Mata, S. Bayona, and L. Pastor, "Neurotessmesh: a tool for the generation and visualization of neuron meshes and adaptive on-the-fly refinement," *Frontiers in neuroinformatics*, vol. 11, p. 38, 2017. 4

[24] M. Abdellah, J. Hernando, S. Eilemann, S. Lapere, N. Antille, H. Markram, and F. Schürmann, "Neuromorphovis: a collaborative framework for analysis and visualization of neuronal morphology skeletons reconstructed from microscopy stacks," *Bioinformatics*, vol. 34, no. 13, pp. i574–i582, 06 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty231 4

[25] M. Abdellah, J. J. G. Cantero, N. R. Guerrero, A. Foni, J. S. Coggan, C. Calì, M. Agus, E. Zisis, D. Keller, M. Hadwiger *et al.*, "Ultraliser: a framework for creating multiscale, high-fidelity and geometrically realistic 3d models for in silico neuroscience," *Briefings in bioinformatics*, vol. 24, no. 1, p. bbac491, 2023. 4

[26] Blue Brain Project, "RTNeuron: real-time visualization of detailed neural simulations," Open source release announcement, EPFL Blue Brain Project News, Aug 2018. 4

[27] M. Defferrard, "Livemesh, a tool for real-time rendering of neuronal cells from morphologies," 2014. 4