Towards Real-Time Visualization of Detailed Neural Tissue Models: View Frustum Culling for Parallel Rendering

Juan B. Hernando∗
CeSViMa
Universidad Politécnica de Madrid

Felix Schürmann
Blue Brain Project
École Polytechnique Fédérale de Lausanne

Luis Pastor
GMRV
Universidad Rey Juan Carlos

ABSTRACT

Recent efforts to model neural tissue in great anatomical detail play an important role in the shift towards simulation-based brain research. Visualization of those detailed models is an important part of the scientific workflow, allowing domain scientists to inspect and analyze the models as a virtual tissue. However, these visualizations pose several computational and interface challenges, from issues such as data size and geometrical and visual complexity. Manual or query driven selection of visual attributes such as transparency, transfer functions, visibility are important components for the usability of those visualizations, yet this requires the rendering to be as responsive as possible. Given the data complexity (several million triangles) - despite the power of current GPUs - parallel rendering is the only choice to visualize the full circuits at interactive levels.

In particular, sort-first is an appropriate choice to deal with transparency, however it requires efficient view frustum culling. This paper presents two novel techniques for view frustum culling based on a bounding volume skeleton which is created from the geometrical skeleton of the neurons. These skeletons are combined with spatial partitions and some GPU processing to extract the portions of the neurons that need to be rendered while still facilitating interactive changes in the attributes. Experimental results show that the proposed method yields better results than classical approaches such as OBB-trees and looks promising for sort-first environments.

Keywords: Cortical neuron simulation visualization, view frustum culling

Index Terms: I.3.3 [Computer Graphics]: Picture/Image generation—Display algorithms; I.3.6 [Computer Graphics]: Computational Geometry and Object Modeling—Graphics data structures and data types; J.3 [Computer Applications]: Life and Medical Sciences—Biology and genetics;

1 INTRODUCTION

Recent improvements in supercomputing power, scientific workflows and quantitative experimental data brought modeling portions of neuronal tissue into reach ([31], [26]). These neuronal tissue models go beyond detailed models of individual neurons or large-scale network models of simplified neurons in that they model tens of thousands of cells in great anatomical detail. Using a discretized cable equation for describing the current and voltage dependencies along the branched structure of each neuron within the tissue, as well as Hodgkin-Huxley equations [20] to describe the transmembrane currents, it is possible to accurately describe the biophysical behavior of the cells.

Publicly available simulation packages have been scaling to massively parallel execution of the numerical solvers for those differential equations (NEURON[7], Moose[4], [25]). Analysis and visualization on the other hand - previously also provided by those simulation packages or auxiliary tools (e.g. [15]) - has not scaled the same way, which severely limits the accessibility to those simulations for domain scientists.

For simulation-based research it is crucial that the domain scientist can inspect the simulation model as if it was a virtual specimen. Interactive customization of the visual representation parameters (e.g. color maps) makes it easier find and highlight specific features. Getting as close as possible to real-time is also necessary because the spatial complexity of a circuit makes it difficult to understand the spatial relationships in a still view, calling for motion parallax as an additional depth cue. Additionally, simulation playback becomes difficult to follow whenever frame rates drop below 25 fps.

Even though the rendering hardware and software have experienced impressive advances during the last years, a single GPU is not enough for dealing with the geometrical complexity of a whole circuit of detailed neuronal morphologies, making parallel rendering techniques necessary for reaching the interactivity requirements (even more whenever advanced rendering techniques such as transparency are used). Practical parallel rendering algorithms on graphics hardware typically fall into two categories: sort-first (assign screen tiles to processors) and sort-last (assign data subsets to processors and compose the resulting images)[32]. Despite sort-last seems a more appropriate parallelization for this problem, in its basic conception it precludes correct transparency, leaving sort-first as a better starting point if transparency is an important option, as it is the case.

The established tools available to neuroscientists for visualizing simulations are the same as those used for modeling and simulation. These tools are not tuned for the specific problems found during the visualization of large and densely packed neuronal networks (for neuron counts above 1000 neurons), parallel rendering being completely out of their scope. In the context of the Blue Brain [31], where a unifying tissue model of a piece of cortex of a young rat is being created, we therefore have been developing dedicated high performance visualization technology [18], [12] to allow scientists to visualize those detailed tissue models interactively.

This paper presents part of our work towards the real-time, fully user customizable visualization of the simulation results from large cortical circuits; in particular, two algorithms for efficient view frustum culling are presented. Both algorithms are characterized by: a) the efficient dispatching of the draw commands for each object, which is accomplished by minimizing the primitive fragmentation caused by frustum culling and improving cache efficiency; and b) the preservation of the objects’ independence, which eases the application of different shaders, materials and textures to each object while minimizing state changes. The basis of both algorithms is a bounding volume skeleton comprised of capsules which is computed for each neuron. The first algorithm uses a spatial partition to cull these skeletons while the second one takes advantage of the compute power of the GPU for that purpose.

The rest of the paper is organized as follows: The following section presents some actual facts about the visualization problem. Next, a brief overview of related work follows. Subsequent sections present the description of both algorithms together with some
implementation details, the exploitation of levels of detail and the results achieved in our cortical circuit models. The last section summarizes the conclusions and future work.

2 Visualizing cortical column simulations

From a visualization perspective, neuron ensembles are particularly hard to render and visualize. Neurons are geometrically complex objects with very large convex bounding volumes compared to the volume fraction they actually fill. For instance, the dendritic arborizations of cortical neurons can span as much as 100 µm in diameter in their zenith view and around 1 mm in vertical extent; yet, axons can be as thin as 0.1 µm in width. Additionally, neurons are densely packed and entwined within cortical tissue. Figure 1 shows an example of a minicolumn (100 neurons) and the same minicolumn embedded into another 10.

The reconstruction of a cortical neuron morphology is composed of 4,700 segments on average. This reconstruction process is a manual digitization task performed by trained operators on tissue samples and its output is a collection of lists of points and radii and the hierarchical organization of these lists. These reconstructions, together with recordings from electrophysiological experiments are the basis to create the mathematical models that are simulated in a parallel version of NEURON [19].

Polygonal meshes that model the cell membrane can be obtained from the skeletal morphologies [28], having 150,000 triangles and 75,000 vertices per neuron on average, so in a 10,000-neuron circuit (this is the estimated neuron count within the structural and functional unit of the rat neocortex) there are more than $10^9$ triangles. Levels of detail can be used to reduce the model complexity; however, the artifact free rendering of closeup views requires the fine detail that only meshes can provide. The exploration of these data sets requires thus the use of techniques that reduce the amount of geometry sent to the graphics pipeline.

The basic problem can be formulated as the rendering of these geometrical models with simulation results mapped to them and based on user selected parameters that determine the visual appearance.

Given the rendering power of current GPU accelerators hardware accelerated rasterization has been chosen as the basis for this work. Effective view frustum culling has a remarkable importance to achieve scalable sort-first parallelization of rasterization and selecting an appropriate spatial data partition is key to speed up this process. The atypical problem that needs to be addressed in order to cull the branchy and entangled geometry of cortical neurons is that the bounding volumes of nearby neurons overlap extensively and are too large compared to the actual geometry contained inside.

Thus, effective view-frustum culling requires neurons to be split in pieces in order to create a spatial data structure; yet, we want to minimize both the overhead of traversing the data structures themselves as well as the primitive fragmentation, because at the same time geometry is reduced there is an associated overhead that may reduce scalability in practice. The reason is that with current graphics hardware the best performance is obtained when large chunks of geometry are dispatched with as few API calls as possible. Thus, there is a balance between how fined grained our data partition and how much overhead it is introducing due to primitive fragmentation.

Resorting and merging of primitives could be considered in order to alleviate the fragmentation problem and minimize draw calls, but then, each object cannot be treated independently and applying different rendering attributes to each object becomes much more difficult. At this point, sort-last seems to be a better alternative for scalability since view frustum culling needs not to be so effective and the load balancing is much easier, however when transparency is considered we still need to process many different neurons in each rendering node due to the extensive overlap of neurons.

3 Related work

As previously stated, the tools used for visualizing simulation results from cable model simulations are components of the tools used for simulating them. NEURON [7] has limited rendering capabilities, presenting neurons as varying thickness lines with no shading. It also lacks features to display simulation data mapped onto the neurons. Neuroconstruct [15] is another tool which can be used to visualize neuronal circuits. It uses faceted pipes to represent the cell processes, but as a circuit modeling tool, it cannot display simulation. GENESIS [6] is a general purpose simulation framework that can also be used to show simulation results of networks created by Neruconstruct (and any other tool using NeuroML). GENESIS uses also simple representations and basic algorithms for its 3D visualizations. Whole Brain Catalog [27] is another application that can be used for exploring not only simulations but also experimental data from a wide variety of sources. It uses shaded prisms to render the neuronal skeletons as well as simpler representations. As a common fact, to the best of our knowledge none of the previous tools considers parallel rendering in order to speed up the rendering of networks comprising thousands of cells.

View frustum culling (VFC) of hierarchical geometrical data structures [10] is commonly used to speed up the interactive rendering of scenes where only a fragment of the (complex) scene geometry falls within the view frustum. There are two main issues of VFC methods that characterize the different approaches: the hierarchical data structure and the visibility algorithm used (including the intersection tests). Bounding volume hierarchies (BVHs) are data structures commonly used for VFC. The nodes of a BVH can be axis-aligned and oriented bounding boxes (AABB and OBB), spheres and k-DOPs to name a few. Other alternatives are binary space partitions (BSPs)[14], octrees [33] and k-d trees, though the latter are more suitable for other purposes. BVHs are used extensively for collision detection as well as for ray-tracing. In the case of collision detection, the requirements met by the construction algorithms make the output BVHs also suitable for VFC; notable examples are the algorithms to create sphere-trees [22, 21], OBB-trees [16] and k-DOP-trees [24]. These algorithms typically work by minimizing a cost function, a process which is computationally expensive. Combining different volume shapes can improve the results, as shown in [9]. Data structures and algorithms for dynamic scenes have also been proposed, an example are loose-octrees, in which a certain overlap between the nodes is allowed. BVHs can also be applied to dynamic scenes better than other classic structures and AABBs have also been considered to be better than OBBs for this purpose [35]. Some recent works in interactive ray-tracing present GPU-acceleration of BVH construction for deformable objects [29]. Different authors present a series of optimizations for faster view frustum culling for AABBs, OBBs [1, 2] and spheres [5] (octant tests, plane caching and masking, etc.), most of which are
typically implemented in scene graph toolkits. These optimizations are independent of the algorithm that creates the spatial partition; for this reason they can only provide moderate accelerations when the spatial partition is more relevant. BVHs for massive models can be very memory consuming, this problem can be addressed by compression and more compact representations as in HCCMeshes [23].

4 VIEW FRUSTUM CULLING FOR CORTICAL CIRCUITS

The following sections describe the view frustum culling algorithms that have been designed and implemented for this visualization problem. The first section describes how the skeleton of bounding volumes is extracted from the morphological structure of the neurons. The following sections describe a two-level algorithm based on an octree partition and the bounding volume skeletons and then a stream computing approach to solve all the bounding volume-frustum intersection tests on the GPU.

4.1 Creating bounding skeletons of neuronal models

The bounding volume skeleton used for each neuron is a set of capsules (semi-spherically capped cylinders) with no hierarchical information. This skeleton is obtained from the geometry to render and the skeletal description of the neurons from which that geometry has been derived (refer to [28] for details about the mesh models used). The input geometry is assumed to meet certain requisites:

- It has a tree structure consisting of tubular-shaped pieces that branch with no particular arrangement. The skeletal structure is a hierarchy of sections, which are poly-lines that connect one to each other at branching points and are composed by chains of segments (see figure 2 for details).

- For each vertex of the input geometry, there is a mapping that relates that vertex to a skeletal section and to a relative position along that section. The relative position is the normalized distance from the section start to the Euclidean projection of the vertex onto the section poly-line (an arbitrary function then determines the section and relative position of a triangle based on its three vertices).

- Without loss of generality, the input geometrical primitives to consider will be indexed triangle lists. Each list is sorted using the section identifiers of the triangles. Within each section, triangles are sorted by their relative positions along the section.

Skeleton generation and primitive preparation The bounding volume skeleton consists of a set of bounding volumes that cover the geometrical model. There are several types of bounding volumes that can be fitted tightly to chunks of the input geometry: among all of them, capsules were preferred because of the properties of their intersection tests against AABBs and pyramidal frustums (both tests can be solved approximately and efficiently if a small number of false positive is allowed).

Each capsule will bound a portion, that is, the geometry along an interval of a single section. We will refer to the set of bounding capsules that encloses a neuron as the capsule skeleton, which is stored in a very simple data structure. For each capsule we store the start and end points ($\vec{C}_s$ and $\vec{C}_e$), its radius ($r$) and section and portion identifiers. Portion identifiers are assigned to capsules in an orderly fashion starting at 0 from the beginning of the section. The capsule data are stored in a flat array sorted first by section identifier and then by portion identifier.

In order to create the capsule skeleton, every section of the neuron skeleton (which is used to generate the mesh) has to be divided into portions, each one to be enclosed by a single capsule. The maximum number of portions per section is limited to 32, so that a 32 bit integer can tell which portions are visible within each section. The portion identifiers are used to operate with bit-masks later on.

For each section, the portions are constructed by iterating over its segments. During the iteration, segments that are almost collinear are grouped as a single portion as long as the total length of the portion is below a specific threshold and no segment deviates from the portion axis (which is defined by the first segment) beyond another threshold. If the number of portions exceeds 32, the thresholds are increased and the section is re-split. The procedure is repeated until only 32 or fewer portions remain within that section. Eventually, the beginning of the first segment and the end of the last segment of a portion define the capsule’s end points ($\vec{C}_s$ and $\vec{C}_e$).

Reasonable values for the maximum length and distance to the axis depend on the model dimensions; nevertheless, those parameters were not found to be performance critical for our meshes having fixed them to 8 $\mu$m for the maximum capsule length and 2.5 $\mu$m for the maximum off-axis distance.

Some extra data to be used in the next step are temporarily stored while this computation is carried out. These are the normalized position of $\vec{C}_s$ and $\vec{C}_e$ within the section, which will be referred to as $\vec{C}_t$ and $\vec{C}_r$. These relative positions are analogous to the vertex mapping (see figure 2) and can be easily obtained from the skeletal points.

The worst case run time of the previous process is not strictly $O(n)$ due to repetition of the section split stage, but for the kind of models considered here, the expected running time is in practice proportional to the total number of segments.

The previous iteration returns the initial points ($\vec{C}_r$ and $\vec{C}_s$) of each capsule together with their section and portion identifiers. The capsules need to be post-processed further so that $r$, the final $\vec{C}_r$ and $\vec{C}_s$ and some extra information required for geometry dispatching are computed. The extra information to be extracted is an interval of the triangle list of the mesh that contains all the triangles within the capsule’s portion. Let $M$ be the set of triangles of the triangle mesh, and $p(t)$ the relative position of a triangle along its section; then the interval $[i, j]$ defined as $\min\{i | t_i \in M, C_s \leq p(t_i)\}$ and $\max\{j | t_j \in M, p(t_j) < C_e\}$ will be the triangle list interval that contains all the triangles that belong to that capsule/portion (the pre-conditions from Section 4.1 guarantee that these triangles belong to a continuous piece of section). The final capsule end points and radius are computed so that the capsule fully encloses this triangle set.

Since capsules are convex objects, ensuring that a capsule contains the vertices of a triangle is a sufficient condition to guarantee that the capsule contains the triangle. The capsule radius may be increased and/or its end points shifted along the capsule’s axis in

Figure 2: Left, a simple geometrical skeleton with 3 sections, each of them composed by a certain number of segments and portions. Right, the triangle mesh for that skeleton. Some vertices from the green section are labelled with their skeleton mapping tuple (section identifier and relative position).
C  
skull skeletons. At the right, the tree structures with the addi-
tional data structures, and an octree of capsules (known as
capsule octrees). The root node contains the list of tuples
(skeleton, position, orientation, leaves) for each inserted skeleton
instance. The leaves contain lists of pairs (section, mask) for each skeleton
they intersect (only leaves a, b and h are shown in the figure) and
a pointer to its additional data tuple.

Figure 3: The two main steps of the capsule extraction process. Step 1 shows the portion extraction and the initial calculation of the end points for capsules. Step 2 depicts the capsules with their final radii and end points adjusted to encompass the triangles that belong to their sections and ranges. Each section is colored with a similar hue.

Figure 4: 2D representation of a simple capsule quadtree with 2 skeletons inside (A and B). At the left, the nodes and the capsule skeletons. At the right, the tree structures with the additional data structures. The root node contains the list of tuples (skeleton, position, orientation, leaves) for each inserted skeleton instance. The leaves contain lists of pairs (section, mask) for each skeleton they intersect (only leaves a, b and h are shown in the figure) and a pointer to its additional data tuple.

order to include the three vertices of every triangle. Shifting $C_i$ and $C_j$ is allowed because tighter capsules can be obtained this way.

Taking advantage of the fact that both the capsule list and the triangle list are sorted by section and relative position along the section, the final size of all the capsules can be computed in $O(n)$ time, being $n$ the number of points of the neuronal skeleton (the number of triangles in the mesh appears to be also proportional to $n$ [28]).

Figure 3 shows the whole process schematically.

4.2 Two level view frustum culling

The first of the culling algorithms proposed here is based on an octree-based spatial partition which stores the capsules of the skeletons. The capsule skeletons are thus the bottom level of the culling data structures, and an octree of capsules (known as skeleton octree) is built on top of them. In a frame basis, the skeleton octree is traversed to annotate potentially visible capsules and then determine the pieces of the geometrical primitives that need to be rendered.

4.2.1 Skeleton octree construction

A scene consists of a set of neuron models (and their skeletons) which are instantiated once or more and placed in space by some transformation. The skeleton octree construction process starts by computing the bounding box of the whole scene and creating an octree root with that box. Then, the skeleton instances are inserted one by one in the octree. Each skeleton is tested for intersection against the octree leaves starting from the root and descending recursively. Each leaf stores a list of pointers to some additional information of every intersected skeleton $S$, a list of lists of pairs $(\text{section}, \text{mask})$ that compacts the intersected capsules of each skeleton $M$, and the total number of capsules intersected by that leaf (see figure 4). Since each section has at most 32 portions, the output of the intersection test of a section’s capsules and the AABB of a leaf can be coded in a 32-bit integer, the visibility mask.

When a leaf is reached during the insertion of a skeleton all the capsules from the skeleton must be tested for intersection. If the number of intersected capsules plus the current capsule count exceeds the maximum allowed and the maximum depth has not been reached, then the leaf has to be split (the maximum capsules per leaf and maximum depth are user defined parameters).

To split a leaf, we need to know the spatial transformation of the skeletons it stores in order to perform the intersection tests against the new children. Another useful information to store is the number of leaves intersecting each skeleton (it helps detect fully visible objects later on). These data, together with a pointer to each skeleton, are stored at the root node as a list of tuples. These pointers are passed during the construction process to store them in the $S$ lists of the leaves (see figure 4 for details).

Whenever a leaf is to be split, it is first cleared and then the leaf count of all the skeletons inside the leaf is decreased by one. The skeletons are then added to the 8 new children as needed. When a skeleton is inserted inside a leaf, the leaf count is increased by one, the list of visibility masks $M$ is computed and pushed into $M$ and the extra information pointer $S$ is added into $S$.

Once the octree has been constructed, unless the scene is going to be modified, most of the skeleton data can be discarded, only the arrays of triangle list ranges for each portion are needed.

4.2.2 Octree cull traversal

This is the first of the steps executed every frame by the algorithm. During the cull traversal the visibility information of each skeleton is annotated. Every skeleton instance contains a visibility type attribute and a list of section visibility masks. Before processing the octree, all skeletons stored in the octree are initially tagged as not visible and their visibility masks are cleared to 0.

The traversal proceeds as a regular octree culling traversal. During this process, a reference to a table $T$ is passed from parent to children nodes. This table stores the leaf counts of each skeleton instance using the instance pointer as the key. Before the traversal, these counts are initialized to the number of leaves that intersect each skeleton (this information is stored in the root node). For each leaf that is reached, the section visibility masks that it contains $(M)$ are merged with the section visibility masks of the skeletons it points to $(S)$, tagging the skeletons as partially visible. When the leaf is fully visible, the node counts in $T$ of the skeletons in $S$ are also decreased, and every skeleton whose count reaches 0 is tagged as fully visible.

At the end of the traversal each skeleton has been tagged as visible, partially visible or not visible. For each partially visible skeleton it is also known which capsules are potentially visible and the visibility masks of the skeletons are fully updated.

4.2.3 Final primitive elaboration and rendering

The final step before rendering is the computation of the visible ranges of the indexed triangle lists based on the visibility bits of each section. This step only needs to be performed for those objects whose skeleton has been tagged as partially visible.

First, two empty vectors are created to store the start indices and element counts that define the triangle list ranges to draw. These ranges are constructed in a single iteration over the visibility bits of the capsules. When a capsule’s bit is set, the portion range array is accessed to retrieve the start and end indices of the capsule over the triangle list. Ranges are pushed into the vectors and collapsed with previous ranges when possible. Only the last range in the vectors is checked for overlapping which makes the process strictly $O(n)$, with $n$ the number of capsules.
The range list is expected to be compact based on two observations: a) there is a full order between ranges from different sections; and b) in practice and within the same section, a later capsule portion (as they appear in the arrays) does not start before an earlier portion. If the second observation happens to be false, the algorithm will still work but some triangles will be overwritten.

Once the ranges have been computed (or the object is fully visible), the actual draw calls can be issued. Objects that share the same mesh model can be rendered in a row, which minimizes the number of geometry buffer bindings and reduces the working set size, improving the overall cache hit ratios.

4.3 Stream computing approach

Classic view frustum algorithms assume that intersection tests are expensive operations and are less concerned with primitive fragmentation. On the other hand, current GPUs need to be fed with models as large as possible, hitting a CPU or bus bottleneck and stalling the GPU if the triangle count per batch is too low.

The previous algorithm uses an octree to speed up the culling of the capsule skeleton of the model. Instead of using an octree it is possible to use a BVH of capsules or even the capsules directly. In principle, testing capsules one by one is much more expensive than using a hierarchical spatial partition for that purpose; this is true if all the calculations must be carried out on a single core, but checking the visibility of a capsule array is an embarrassingly parallel task. In consequence, the performance benefits of hierarchical partitions are disputable if hundreds of capsules can be tested in parallel in the same time required by a single check. Additionally, the pre-processing steps to generate the capsule skeletons are needed once per model and not per scene and instance as with hierarchical spatial partitions. This is a clear advantage for dynamic scenes and for reducing the pre-processing time of static scenes.

This section presents a culling algorithm that uses the GPU to compute the visibilities of an array of capsules. The core of the algorithm is the compute kernel that carries out the capsule-view frustum intersection test. The other main issues to solve are how to dispatch the kernels and read the visibility results back to the CPU for computing the final triangle lists ranges to render. The implementation has been carried out using CUDA [11] for convenience, but the algorithm could be adapted to OpenCL as well.

4.3.1 Skeleton uploading

This algorithm requires the same data than the previous one for each skeleton, but most of it needs to be uploaded to the GPU memory. For each capsule the intersection test needs: the capsule end points and radius, the section identifier, the portion identifier and the capsule index (actually provided by the runtime as the thread index). Uploading the skeletons in local coordinates is useful to share the same data between neurons with identical morphologies.

A single flat array per skeleton is used to store the capsule skeleton with one sub-array per data type. Sub-arrays with the largest numerical data types are placed first to avoid memory alignment issues and to allow coalesced reads. At the beginning of the array a small structure is added with the information required to find the starting address of each sub-array; this header is padded to make sure that the first sub-array is aligned.

4.3.2 Kernel execution

Before solving any test in the GPU, a simple optimization is to test the bounding box of each skeleton against the view frustum. The capsule-frustum intersection test kernel is launched only for those skeletons instances whose bounding box is partially visible. In view frustum culling, false positives may be allowed because they do not compromise the correctness of the rendering results. Taking advantage of this fact, the number of operations of the capsule-frustum intersection test can be reduced. The first approximation is to extrude all the frustum planes outwards by the capsule radius; hence, the intersection test is reduced to a segment-frustum test.

The second approximation is more aggressive: a capsule is tagged as visible if each of the pairs of opposite planes that define the extruded view frustum (top & bottom, left & right, near & far) contain at least one of the capsule’s end points (a 2D version of this test is shown in Figure 5). This simplification is unsuitable for other applications (e.g. collision detection), however, given the dimensions of the view frustum as compared to the lengths and radii of the capsules generated for the models used here, the impact of false positives in the rendering performance is expected to be negligible.

Each thread carries out the test of a single capsule. The input data to the kernel are the pointer to the flat array of skeleton data, the model-view matrix of the model instance, the frustum specification, a pointer to the global visibility array and the capsule index. The output array contains a mask per section and needs to be cleared to 0 before the kernel is executed. The kernel parameters are stored in shared on-chip device memory, so the reason to pass the skeleton data as a single pointer is to reduce shared memory usage. The extra operations to find the sub-array addresses are negligible.

Once the addresses have been computed, the data for the calculation are fetched. Data arrays have not been interleaved, which means that for each read, all threads access a contiguous range of memory addresses. The sub-arrays have been properly aligned to allow coalesced read operations and maximize bandwidth usage. After the data is fetched, the capsule end points are transformed by the model-view matrix and the intersection test is evaluated. All the threads that detect a visible capsule compute the bit mask of the capsule using its portion identifier and merge the mask with its accumulated section mask, which is stored in the global memory output array. For that, an atomic global or operation is used.

We tried to optimize the merge operation by doing part of it in on-chip shared memory instead of global memory. The idea was to use one thread per section and thread block to collapse the masks of the visible capsules (previously written to a shared memory array by the evaluator thread) and do a single global atomic memory operation per thread block and section. This solution improved considerably the performance in older hardware but in Fermi and newer NVidia graphics cards this is not the case, being even slower than the simpler solution.

4.3.3 Gathering kernel output

To get results back from the GPU into CPU memory, the best choice seems to be to allocate an output array per skeleton and then issue
a memory set, kernel launch and asynchronous copy back for each skeleton. Another possibility is to launch all kernels and then issue all copy operations. In theory this scheme allows the overlap of the copy backs with the final primitive construction of the draw step, however, the copy back operations are small and the overhead reduces the effective global bandwidth obtained. Also computing and rendering tasks are going to be issued concurrently and compete for the GPU, which is a known cause of performance problems.

An alternative is to use a single output array for all the skeletons in the scene and read it back in a single operation. Although apparently under-performing, this approach yields better throughput in practice.

### 4.4 Final primitive construction and rendering

The last step is the same as for the skeleton octree algorithm. If for a given system the multi-stream approach is efficient, instead of a global unique synchronization point for all kernels, this step would include a synchronization barrier for each primitive/skeleton.

### 5 Integration with levels of detail

To a certain extent, it is possible to combine levels of detail (LOD) with the two culling techniques presented here; in particular, discrete LODs can be easily integrated. Each object representation is assumed to consist of primitives for which the primitive range contained within each capsule can be computed. By storing all the intervals for each capsule and resizing the capsules appropriately it is possible to keep a single skeleton for all LODs. Every frame, the skeleton capsules are cut and a particular LOD is selected for rendering. For that LOD, the primitive ranges to render are computed based on the visible capsules and the primitive ranges for that LOD stored within them. As a downside to this approach, the parameters chosen to generate the capsule skeleton may not provide good performance for all the LODs (capsules too big or too small).

A simplification of the neuronal models based on a rendering technique for pseudo-cylinders [34] has been used to evaluate this approach. The morphological skeletons are processed to create line primitives that are transformed in the GPU into Phong shaded screen-aligned quads, producing a convincing result. For this lower level of detail, the core of the cells, known as soma, is cropped from the detailed mesh and combined with the pseudo-cylinders.

### 6 Experiments

A number of experiments has been performed to assess the results of the culling algorithms. Essentially, they are based on the rendering of a camera rotation around 3 model circuits composed of 1,000, 2,000 and 5,000 neurons respectively. 1.35, 2.86 and 4.67 average neurons per model respectively. The skeleton construction uses 8 μm and 2.5 μm for the initial maximum capsule length and the maximum distance to the first segment’s axis. Skeleton octrees have at most 200 capsules per leaf. The test system was an Intel Xeon X5690 6@3.47 Gzh with 24GB of RAM and an NVidia GTX 580 graphics card with 3GB of DRAM (driver version 270.41.19) running RHEL 6.1 64bit, kernel 2.6.32. OpenSceneGraph 3.0.1 has been used in the implementation.

For comparison purposes another culling technique based on OBB-trees ([16]) has been implemented. OBB-trees were chosen because of their simplicity and high citation count. Sphere trees were also evaluated but preliminary tests indicated a much worse performance. Our OBB-tree implementation creates a tree per neuron morphology with one modification to the original algorithm: to split an OBB in 2, we compute the 2-means clustering of the triangles instead of halving the OBB to divide the triangles in two groups. This change provides a tighter hierarchy of OBBs (a balanced partition using the median along the split axis was also tested but it performed worse). For primitive dispatch we store the relevant primitive ranges in each tree node and to speed up the final dispatch we sort and collapse the ranges that the cull traversal determines as visible (this optimization notably outperforms straight-forward dispatching). In the experiments, subdivision levels from 3 to 6 were tested; results are only presented for 5 and 6 because for 3 and 4 the results are notably worse. The resulting OBB-tree could be improved by using a better approximation of the optimal OBB as presented in [8], but the original approximation has been kept for simplicity. The hierarchical partition can also be improved, but this optimization if far from trivial.

The translucent renderings have been done assigning to each vertex an alpha value that depends on the cross-section of the neuron at that point and rendered with dual depth-peeling [3]. Better algorithms exist for more common geometry but they are either approximate [13][30] or are limited by hardware resources in the maximum depth-complexity they can cope with [17] (many pixels have more than 60 layers until saturated in the 1,000-neuron circuit). All tests have been rendered at 640x480 resolution to stress the geometry processing stage over pixel shading.

#### 6.1 Efficacy

The efficacy is the ratio between the amount of geometry dispatched for rendering and the minimum possible amount which is needed. The efficacy of both algorithms is bounded by the capsule skeleton, however this limit cannot be reached. For the skeleton octree algorithm, this is because capsules are culled using a coarser partition (the octree). For the GPU algorithm, it is due to the simplifications of the capsule-view frustum test. The efficacy of a skeleton octree of depth 6 and the GPU algorithm is illustrated in Figure 6 by means of 3 shots of a 100-neuron circuit taken at different distances. The images show areas outside the view frustum which includes the redundant geometry being processed. This experiment shows how the GPU algorithm reduces more effectively the amount of geometry to process and that the approximation of the intersection test does not seem to have a negative impact.

#### 6.2 Runtime performance

The frame times of our algorithms compared to OBB-tree culling are shown in figure 7. The bars indicate average wall clock CPU inter-frame, cull, draw dispatch times (stacked bottom up) for 50 frames of a camera rotation around 1,000, 2,000 and 5,000 neuron circuits using different viewport areas to simulate sort-first partitions. We can see that GPU culling and skeleton octrees outperform OBB-trees in all cases. Between these two algorithms, GPU culling performs better when its better culling efficacy compensates the overhead of kernel dispatching. This overhead becomes problematic when the viewport is reduced and when the number of neurons is increased, effectively limiting the speed ups. We can also see the impact of the CUDA kernel and primitive construction by comparing the red and green bars. From these results we can expect a reasonable scaling of sort-first parallelization up to 8 nodes, from that point the scalability starts to look sub-linear.

Tests with unique and translucent geometry are shown in image 8. We can see that the benefits of our techniques are exaggerated when unique geometries per neuron are used as long as the geometry fits into GPU memory (1K case). For the 2K case we see the CUDA culling algorithm competing for resources with the rendering. In the translucent renderings, the frame time is dominated by the multiple passes performed in the draw stage. Here reducing the viewport has clear benefits, but this is because the rendering is pixel shading limited.

Tests with pseudo-cylinders instead of detailed meshes are shown in figure 9; these tests have been run with the same parameters for creating the capsule skeletons as before. In the opaque tests we can see than the reduced geometry workload of this level of detail makes the algorithms behave much worse that with meshes; the speed up of the best algorithm is comparable to that of the simple
AABB test (even with a still camera), probably pointing out to a bottleneck caused by low primitive count per draw call (however, increasing capsule sizes degrades efficacy, so the solution is not straight forward). It is interesting to see the the skeleton octree perform worse at large viewport areas than the CUDA algorithm. On the translucent tests we can see that even for this small viewport (640 × 480), the rendering becomes constrained by pixel shading (observe how the algorithms compare to simple AABB test).

7 Conclusions

The interactive visualization of detailed cortical circuit simulations is a computationally complex task. This paper has addressed a part of this problem, which is the view frustum culling of cortical circuits in order to facilitate the interactive navigation within these circuits and its sort-first parallel rendering.

Two algorithms tailored to this problem have been presented. We have shown that both algorithms outperform OBB-trees in our cortical scenes thanks to a combination of a good degree of cull efficacy and the minimization of primitive fragmentation. At the same time, the algorithms preserve object independence, which allows easy interactive modification of their rendering attributes. Both algorithms look promising in a sort-first parallel setup if good load balancing is achieved. When the geometrical workload is reduced, both techniques become less effective; additional parameter tuning needs to be performed to improve the performance in this case.

The GPU-based approach has additional interesting features. The data structures are fast to create and its performance matches the performance of the skeleton octree for many cases. This method is also suitable for scenes with rigid-body animations and its better culling efficiency accelerates the rendering when the culling results can be reused for several rendering passes. The playback of computer simulations of neuronal circuits with a still camera or multi-pass rendering algorithms such as dual-depth peeling can benefit from this fact.

Future research lines and developments in view frustum culling include the parallelization of the cull stage in distributed memory environments to improve the scalability, and the pipelining of the cull and draw stages. Using recent OpenGL extensions (e.g., GL_AMD_multi_draw_indirect) it is also feasible to generate the primitive ranges to render without reading back the cull results back to the CPU.

Regarding the visualization problem, these algorithms are already integrated within a production application called RTNeuron, which is used for debugging and media production for presentations. Ongoing work includes load balancing improvements for parallel rendering and the wrapping of the rendering engine as a Python library in order to embed a Python interpreter in the tool to allow interactive customization and faster GUI development. The combinations of these developments is expected to facilitate the understanding of the simulations and foster scientific discovery.

Acknowledgements

This work has been funded by the Spanish Ministry of Education and Science (Cajal Blue Brain project and grant TIN2010-21289-C02-01&02) and the Blue Brain Project.

References

Unique geometries
Translucent pseudo-cylinders

32 difference compared to figure 7 for 1K neurons, but for 2K the rendering time is modulated by the cross section of the branches. At the right, opaque meshes using a unique geometry per neuron; in this case, there is no circuits using only a subset of the algorithms from figure 7. At the left pseudo-translucency rendered with dual depth-peeling, (alpha channel

Figure 8: CPU frame times for unique and translucent meshes Average frame times for 50 frames of the whole camera rotation around 3 circuits using only a subset of the algorithms from figure 7. At the left pseudo-translucency rendered with dual depth-peeling, (alpha channel modulated by the cross section of the branches). At the right, opaque meshes using a unique geometry per neuron; in this case, there is no difference compared to figure 7 for 1K neurons, but for 2K the rendering time is \(2 \times 3\) higher (5K data missing due to memory constraints).

1K neurons
2K neurons
5K neurons
1K neurons
2K neurons
5K neurons

Figure 9: CPU times for pseudo-cylinders Average frame times for 50 frames of the camera rotation around the full circuits using the pseudo-cylinders LOD. The color coding and labelling is the same as above. Notice how the translucent renderings are dominated by the draw time. Segment counts are 4.1M, 8.3M and 21.2M for each circuit.


Average frame times for 50 frames of the camera rotation around 3 circuits using only a subset of the algorithms from figure 7. At the left pseudo-translucency rendered with dual depth-peeling, (alpha channel modulated by the cross section of the branches). At the right, opaque meshes using a unique geometry per neuron; in this case, there is no difference compared to figure 7 for 1K neurons, but for 2K the rendering time is \(2 \times 3\) higher (5K data missing due to memory constraints).