Geometric Processing and Efficient Collision Detection for Neurovascular Cells

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the

Blavatnik School of Computer Science, Tel-Aviv University

by

Yoav Jacobson

This work has been carried out under the supervision of
Prof. Dan Halperin

July 2021

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Dan Halperin for guiding me in this journey. His experience and knowledge, and his always-helpful advises, were a crucial part in the success of this work. I've learned a lot while working on this thesis, and mostly from him.

This work could not have been done without Prof. Pablo Blinder, Hagai Har-Gil and Omri Tomer from Sagol School of Neuroscience, who were great colleagues to work with on this work. By bringing different expertises from distinct science fields, while challenging and improving each other's work, I believe we created something together which couldn't have been done by any side on its own.

A special thanks to Efi Fogel from the Computational Geometry and Robotics lab, whose help with providing CGAL alpha-shapes support was most welcome and much appreciated.

I would also like to emphasis the love and support I have received from my family, my friends and my loving wife while working on this thesis.

Abstract

The analysis of neuronal structure and its relation to function has become a fundamental pillar of neuroscience since its earliest days, with the underlying premise that morphological properties can modulate neuronal computations. It is often the case that the rich three-dimensional structure of neurons is quantified by tools developed in fields other than neuroscience, such as graph theory or computational geometry; nevertheless, some of the more advanced tools developed in these fields have not yet been made accessible to the neuroscience community. Here we present NCD, Neural Collision Detection, a library providing high-level interfaces to collision-detection routines and alpha-shape calculations, as well as statistical analysis and visualization for 3D objects, with the aim to lower the entry barrier for neuroscientists into these worlds. Our work here also demonstrates a variety of use cases for the library and exemplary analysis and visualization, which were carried out with it on real neuronal and vascular data.

The library at the heart of this work, NCD, uses state-of-the-art computational geometric techniques and algorithms to perform collision detection at scale, with major focus on performance and accuracy. We designed an elaborate pipeline, which starts from the complex components in the neurovascular system—the blood vessels and the neurons in the brain—and finishes with detailed analysis of collision data of the different components, including correlation between the likelihood of a collision at some points on the neuron and geometric properties of these points like graph distance and the critical values of their alpha-shapes.

Contents

1 Introduction

Neuroscience has strong roots in many scientific fields, including biology, medicine, psychology and computer science. Despite significant progress in all of those fields, there are still many mysteries left to unravel about the brain. It consists of huge and complicated neural and vascular networks, which work together to get a proper function. Understanding the relations between these two networks is a key element to understand how the brain works.

In this work, we present NCD, Neural Collision Detection, an efficient software library for simulating neurovascular micro-circuitry structure using realistic vascular networks and cellular morphologies. We use advanced computational geometric techniques to simulate collision-free settings for neurons and blood vessels, and study them to get insights about the relations between them.

The main goal of our work is to investigate the relations between neurons and blood vessels, and for that we need to know the exact placements of the neurons, in relation to the blood vessels surrounding them. However, existing works focus on either the vasculature [9, 30] or the neurons [42], but not both. When we approached this at the beginning of our work, there was no database available which captured both the neurons and the blood vessels, and there were no statistic frameworks describing the physical interactions between them. Because of that, we decided to simulate the juxtaposition in the best way we can. To achieve a faithful simulation, we rely on the fact that the neurons cannot intersect with the blood vessels, but must lie in between them. Thus, we can place the neurons in many possible positions and orientations (obeying various biological constrains), and select for further consideration only those that incur minimal number of contact points. To this end, we need to have a fast and reliable method to test for collisions between the objects—and this is the job of NCD, the computational pipeline that we have developed.

To deal with the highly complicated and rich objects, and test a satisfying amount

of different settings in reasonable time, we had to invest time and effort in optimizing our pipeline while keeping its accuracy. We used FCL, the Flexible Collision Library [40], an open source C++ library, which efficiently answers collision detection queries, as well as carried out other optimizations, which will be described below, to achieve the desired performance.

Contribution Our main contribution is NCD, the library described above and its underlying algorithmic framework, which are presented in detail in Section 3. NCD is an open source Python and C++ package, which can be freely downloaded from our repository [27]. In this work, we demonstrate several possible uses of NCD, like visualize collision probabilities on top of the neuron and comparing different neurons to one another, and invite the readers to use NCD capabilities for their own research. The article describing this work is available at bioRxiv [25].

Thesis Outline We begin by providing a background on two central algorithmic topics used in our work. In Section 2.1 we discuss the problem of collision detection, by providing several definitions, going through different variants of the problem, presenting several known algorithms to tackle it from different angles and surveying several of the widely used libraries for collision detection. We conclude this section by showing a few examples of how collision detection is used in life sciences, and some related work in that domain. In Section 2.2 we present alpha shapes, a useful tool, which we use to determine how "concealed" a point is in the neuron, and how it affects its likelihood to collide with the blood vessels. We provide definitions to alpha shapes and critical alpha value of a point, and survey leading software libraries used to calculate alpha shapes.

In Section 3 we go into details about the computational pipeline of our system and its architecture. We describe the different components and stages in the pipeline, while providing background and the reasons for our choices. Then, we discuss the tools we used during the development phase to ease the process and to verify the correctness of our code, including visualization of the cells and a toy example we created for NCD.

Next, in Section 4, we focus on the biological side of the work—we present the biological background to the work, and provide some exemplary experimental results, using real neurovascular data. As this work focuses more on the framework than the actual results, this part serves as an invitation to the reader to continue and explore the library capabilities.

We conclude the work in Section 5, where we reflect on our development process, draw conclusions from our experience during the development, and pose some possible directions for future work.

2 Background

We review two topics, which are central to the thesis. In Section 2.1 we survey algorithms and software for checking collision between three-dimensional objects. In Section 2.2 we discuss alpha shapes, which are a useful tool to determine how "buried" a point is in a given set of points.

2.1 Collision Detection

Collision detection is a fundamental subject of study in computational geometry and robotics, with applications in many areas, including motion planning, computer games, physics simulations, virtual reality, collision avoidance and computer graphics. The basic collision-detection problem is to test whether or not two objects intersect with each other. Naturally, there are a lot of variants to the problem, which we will discuss in Section 2.1.3. As a result, this is an intensively studied problem, many algorithms were presented in the literature, and many libraries exist, tackling the issue from various angles.

Though collision detection is relevant both in 2D and 3D, in this chapter we will focus on 3D collision detection, since this is both the case in our work and the more common and interesting case to study.

2.1.1 Objects Representation

A geometric object is a 3D entity with volume and boundary. In our context, a geometric object can be of any shape, convex or not, and is rigid, meaning its shape cannot be changed.

An object can be represented in several ways, depending on the way it was created and

how it is going to be used. The most commonly used representation is by a *triangle mesh*—a set of triangles in 3D space, connected by their common edges and vertices, which constitute the faces of the object [10]. A *polygon mesh* is the same, but using any polygon instead of only triangles. Another common representation is by a *point cloud*—a set of points in 3D space, which usually reside on the boundary of the object [45]. A similar way to represent an object is by a set of balls, defined by their centers and radii, whose union constitutes the object. This approach is less common, but is the one used to represent the original data in our work.

2.1.2 Position and Orientation

The *position* of an object is the position in $\mathbb{R}^3$ of some reference point of the object, usually its center. The position is defined by three coordinates, x, y, z, of that reference point.

The *orientation* of an object in $\mathbb{R}^3$ can be defined by three angles, α, β, γ. One possible interpretation of the three angles is that the object is rotated by γ degrees around the z-axis, then by β degrees around the y-axis and finally by α degrees around the x-axis; see [32, Chapter 3]. It is important to note that the order is significant—though other orders are valid methods, they yield different results. Another way to define an orientation is by a "rotation matrix"—a 3×3 matrix, by which the coordinates points of the object (represented as vectors) are multiplied, in order to apply the orientation [46]. This is usually the representation used by programs, since it is easy to calculate and avoids ambiguities.

The *placement* of an object comprises these two attributes together—position and orientation.

2.1.3 Variants

As such an important topic in many fields, the collision-detection problem has many variants, each poses different challenges and requires dedicated algorithms.

First, in most applications the objects are moving (or at least one of them is), and we continuously need to know if they collide. Though it is mostly possible to calculate it frame-by-frame, it is highly inefficient—the frames often change very little, and this behavior may be exploited to gain in efficiency. In addition, the frame-by-frame method is discrete, and might be less accurate in finding the exact time of collision, or might miss some collisions altogether if the movement is too fast and the objects are too thin.

Second, in many situations there are not only two, but rather n objects in the region in question, and we need to know if any two objects collide. An example is a video game, in which a typical scene contains many objects, and for realistic game experience, any collision should be known and dealt with in real time.

Third, sometimes knowing if two objects collide is insufficient and more information is required. For example, in cases where the objects are not rigid or we want to calculate some error score (a numerical measurement to how intensively the objects collide, when they are

not supposed to), we may want to know the penetration depth of the collision—the minimal distance that one of the objects needs to move, so they will not collide. In addition, the ability to make proximity queries is useful, meaning calculating when the distance between two objects is under some threshold. Lastly, the normal of the collision (the direction of the force that each object exerts on the other) is also critical when dealing with elastic collision, like in the simulation of the motion of billiard balls.

Forth, different representations of the objects may require different algorithms. Using different ways to represent the objects, like those described in Section 2.1.1 or others, it may be easier or harder to test for collision, and different algorithms are in use.

2.1.4 Algorithms

When testing for collision of two 3D objects represented by triangle meshes, a simple brute force approach works—go over all pairs of triangles, one from each object, and test for intersections. The two 3D objects collide if and only if any two triangles, one of each object, intersect. (Notice that this method assumes that neither object is contained inside the other—this is a common assumption in central existing collision-detection libraries, and needs to be addressed in settings where this assumption does not hold.) The brute force method is reliable and easy to implement. However, the time complexity of the brute force approach is $\mathcal{O}(n^2)$, where n is the number of triangles in the triangle meshes. When the objects are complex, it may take too much time and be infeasible. Because of that, and to accommodate the different variations of the problem, many more efficient algorithms for collision detection were developed and described in the literature, each addressing a different variant of the problem. In this section, we will survey a small but significant representative set of these algorithms.

A common and fast method to detect collisions in the basic case is using bounding volume hierarchies. This is a family of algorithms, which all have a pre-processing phase, in which a hierarchical tree of bounding volumes is created for each object, where the resolution increases with the depth of the tree. It means that the root stores a bounding volume that contains the whole object, each node on the next level contains approximately half of the triangle mesh of the object, until the leaves contain one triangle each. Then, when two objects are tested for collision, their trees are compared with each other. If the bounding boxes of two nodes, one node from each tree, intersect, then their children are compared with each other. If the triangles stored in two leaves intersect, then the two objects do and the intersection point is reported. The main difference between the algorithms in this family is the nature of its bounding volumes: these can be, for example, axis-aligned bounding boxes (AABB) [8], oriented bounding boxes (OBB) [24] or spheres [26]. The type of the bounding volumes affects some aspects of the algorithm, such as the complexity of collision queries between two nodes, the height of the tree, the flexibility of the tree for changes in the object (e.g., rotating the object) and the complexity of the pre-processing step.

There are two approaches when it comes to collision detection of moving objects—discrete and continuous. Discrete algorithms look for collisions in each time frame separately, while using information gathered in the previous iterations. Continuous algorithms, on the other

hand, calculate the exact time of the collision, which is mostly not the same as the exact time of a frame. Discrete algorithms are typically simpler since they do not need to consider the complex physics of the movement, which might be very complicated to describe accurately and is affected by many factors. However, on the down side, they might miss collisions of fast-moving objects, and they mostly detect the collision after it happened. Continuous algorithms are more accurate, but they need to predict the exact movement of each object, taking into account all of the variables controlling it.

Usually, continuous collision-detection algorithms use arbitrary in-between rigid motion. It means that the motion is approximated as a series of successive placements, determined at fixed points in time, and in between the motion is interpolated with some fixed rigid movement, in a constant velocity and direction. If the time frames are sufficiently small, the approximation error is negligible. This way, the motion is much easier to describe and more feasible to work with. For example, Redon et al. [43] present one such algorithm, which uses interval arithmetic and an extension of OBB for the continuous case.

When using discrete collision-detection algorithms for moving objects, naively applying static methods frame-by-frame is usually not efficient enough, since it misses the similarity between successive frames. One better approach is described in [33]. This algorithm works only on convex polyhedra, and keeps track of the pair of closest features on two bodies, which move in discrete, small steps. These queries are performed in roughly constant time in practice (after an initial pre-processing step), since the pair of features does not change frequently, and verifying this is done in constant time.

These approaches are good for two moving objects, but when there are k moving objects, usually another step is required, which is called the *broad phase*. Its purpose is to quickly reduce the $\mathcal{O}(k^2)$ pairs of objects which are candidates to collide, and eliminate pairs which are sufficiently far from one another and hence are not going to collide soon, so that the next phase—the narrow phase—can feasibly calculate the collisions with algorithms like those described above. There are many algorithms for the broad phase. In one such algorithm [14], the broad phase uses axis-aligned bounding boxes for each object, and sorts them in 3-space, by projecting the bounding boxes onto the x, y and z-axes, and sorting these projections (segments) on each of the axes, separately. If two such segments do not intersect on some axis, then the matching bounding boxes do not intersect, and thus the corresponding objects obviously do not; and if the bounding boxes do intersect, an exact collision detection is performed in the narrow phase. This way, assuming not all of the objects are very close, much less than $\mathcal{O}(k^2)$ exact computations are performed, and only nearby objects are subjected to exact collision detection.

Many algorithms, as those mentioned above, assume that the objects are rigid, and thus the pre-processing phase stays relevant and accurate the whole time. However, when the objects are deformable [31], a hierarchical bounding volume tree is still used, but updated every step, using several heuristics.

2.1.5 Libraries

Naturally, for a common problem with so many use cases, variants and algorithms, many libraries are available for performing collision detection. They differ in functionalities, program language and common use cases. In our work, we looked for a library to perform discrete collision detection in 3D between two complex objects (the neurons and the vasculature), with a major focus on reliability and performance. Eventually, we decided to use the Flexible Collision Library [40], FCL, for reasons that we will mention later. In this section, we will survey FCL and some other widely used libraries for collision detection.

FCL, the library we chose, is one of the frequently used libraries for collision detection and proximity computation. It is written in C++, is available as open source under BSD license and is still maintained by the community. The library supports several types of models as input, including triangle meshes and point clouds. In its core, it uses several types of hierarchical bounding box trees, like axis-aligned bounding boxes and oriented bounding boxes. When testing two objects for collision, the library provides the coordinates of the contact points as well as the penetration depth and the normal of the collision. Being written in C++, and using state-of-the-art algorithms for collision detection, FCL is one of the most efficient libraries.

Bullet [15] is another commonly used library in the field. Rigid collision-detection queries are supported in Bullet as part of its more sophisticated physics simulations, which support gravity simulation and rich visualization. Bullet is written in C++, but it also has python bindings [16]. Though this library is very well maintained and widely used, using it for mere collision detection is somewhat unnatural, and thus it fits less to our needs here.

ODE [47] is yet another C++ library, which provides physics Software Development Kit (SDK), and naturally includes a collision-detection engine. It is a direct competitor of Bullet, and as such they were compared in some studies. In [21], Bullet and ODE (along with some other physics SDKs) were compared, and this study found ODE to be faster than Bullet. However, a more recent study [11] found that Bullet provides better results. As both of the libraries are still maintained and developed, newer studies are needed to have more up to date comparison between them.

Even though C++ seems to be the most commonly used language for collision-detection libraries, some are written in other languages too. One such library is ncollide [5] which is written in Rust, and provides several types of collision queries for 2D and 3D objects. Another example is BEPUphysics [4], which is written fully in *C#* and provides efficient 3D physical simulation.

There are many more libraries for collision detection, and we surveyed some representative examples. When choosing a library to work with, there are many factors to take into consideration, such as performance, accuracy, programming language, ease of use, documentation and maintenance. We chose FCL, a leading library for collision detection, for its efficiency, seemingly good maintenance and being comparatively easy to use, doing exactly what we need.

2.1.6 Collision Detection in Life Sciences

In our work, we use collision-detection methods to answer biological questions. Though the way we do it is novel, we are not the first to bring the two together—collision detection and biology. Next, we briefly touch on some interesting applications of collision detection in the biology world.

Telesurgery, or remote surgery, is the ability to perform surgeries using robots, which makes surgeries safer and saves life. In [37], a framework was developed to provide haptic feedback for the surgeon, enhancing the surgical procedures. This framework uses collision detection to know when a haptic feedback is required, and uses FCL for the task. In [53], a fast and accurate algorithm for collision detection was presented, tailor-made for Robot-Assisted Laparoscopy, using OBBs.

Another interesting application can be found in neuroscience. If the brain could be programatically simulated, it could have created new and exciting way of studying it. Brain activity could have been closely examined and debugged, allowing us to understand it much better than we do today. However, the brain is too complex to achieve it in today's tools, and only simplified versions of the brain can be simulated, by using a lot of computation power to simulate every neuron and the connections between them. On top of that, a crucial part of brain development is studying by experience, and simulated brains need that too. NeoCortical Virtual Robot [29] is a framework that does just that, by creating a virtual world for the simulated brain to experience in. They also use OBBs for collision detection in the virtual world, to make it more realistic, a procedure required by almost every virtual reality system.

In addition to using collision detection to support biology, the other, more surprising direction also exists. Collision detection and collision avoidance are critical tasks for every living creature, from avoiding predators in the wild to safely crossing a road. Therefore, the ability to perform collision detection based on sensory data is developed in the brain, and studied in the neuroscience community. In [52], a specific neuron of locusts, responsible of collision detection, was studied and simulated. Its simulation was then successfully used by mobile robots to avoid collisions with cars. This is a great example of a biological mechanism that inspires algorithms, much like neural networks and genetic algorithms did in other areas.

2.1.7 Related Work in Life Sciences

Besides collision-detection, more computational geometry tools are used in a variety of life-science fields, and specifically in neuroscience. The neuron is a highly complex, three-dimensional structure, with a tight connection between its morphology and function, and therefore special computational geometry tools were created to study it. In [36], for example, a new algorithm was developed for comparing quantitatively the three-dimensional structure of different neurons, based on Hausdorff distance. It allows to analyze a group of neurons, and find which ones have similar morphology, which implies they have the same function in the brain. Another example can be found in [28], where a new way to analyze the structure of dendritic spines (a part of the neuron) was introduced, based on numerous computational

geometric tools, including converting voxel data to surface mesh, feature extraction from 3D mesh models and discrete differential-geometry operators. These methods were all used to extract descriptors of the spines, for a later stage machine-learning analysis.

2.2 Alpha Shapes

2.2.1 Definition

In computational geometry, α-shapes provide a way to capture the "shape" of a finite set of points, with the α parameter controlling how fine or crude the representation of the shape is. For a set S of points in the plane and a value α, the α-shape is a polygon that encloses all of the points in S, and its vertices are a subset of S. A point from S is a vertex of the α-shape, if and only if there exists a circle of radius α, where the point lies on its boundary, and no point from S is inside the circle. Similarly, a segment between two points from S is an edge of the α-shape, if and only if there exists a circle of radius α, where the two points lie on its boundary, and no point from S is inside the circle. For the purpose of α-shapes, a circle with radius $\alpha = \infty$ is defined to be a half-plane. In that case, the α-shape is actually the convex hull of S. The definition of α-shapes in 3D is analogous, with balls instead of circles and a half-space instead of a half-plane.

More formal definitions of α-shapes can be found in [19] for 2D, and in [20] for the 3D case. In these articles, Edelsbrunner et al first introduced α-shapes and discussed some of their properties.

2.2.2 Critical Alpha Value of a Point

For a given point in a finite set of points, α-shapes can be used to measure how "interior" or "exterior" the point is. When α is sufficiently big, the α-shape is identical to the convex hull of the points, meaning that only the extreme points (those lying on the convex hull) constitute the α-shape's vertices. As α decreases, more points become α-shape's vertices, until for sufficiently small values of α, all of them are. In other words, in some sense, the more interior a point is, α needs to be smaller in order for that point to show up on the surface of the α-shape.

We point out some properties for each point p in a set of points S. For a given α, Surface(p, α) is true if p is on the surface of the α-shape (being one of its vertices), and false if p is interior to it. It is important to note two attributes of Surface(p, α). First, for every $\alpha_1 > \alpha_2$, if Surface$(p, \alpha_1) = true$ than Surface$(p, \alpha_2) = true$. In other words, if p is on the surface of α_1-shape, it is on the surface of α_2-shape for every $\alpha_2 < \alpha_1$. Second, for every point p there exists some α such that Surface$(p, \alpha) = true$.

The second property we point out is the *Critical alpha value* of a point, $\alpha_0(p)$, which is the largest α such that Surface$(p, \alpha) = true$. Because of the attributes mentioned above, $\alpha_0(p)$ exists for every p, and for every $\alpha < \alpha_0$, Surface$(p, \alpha) = true$. Notice that if p is on

the convex hull of S, than $\alpha_0(p) = \infty$.

We use $\alpha_0(p)$ to measure how interior, or hidden, a point is in our data set—the smaller α_0 is, the more hidden a point is. We are interested in the property of being interior, since it might affect how likely a point is to be part of a collision in our experiments. A more detailed discussion about this issue can be found in Section 4.2.

2.2.3 Software Libraries

In our work, we used the Computational Geometry Algorithms Library (CGAL) [50], a state-of-the-art C++ computational geometry toolbox, which has a rich α-shapes module, due to its performance and rich functionality. We utilized CGAL's Python bindings, as described in Section 3.8.

However, as such an important and useful concept in computational geometry, there are many more libraries for working with α-shapes, in many programming languages. One such library is Alpha Shape Toolbox [1], which allows an easy creation of α-shapes in Python. Another package is alphashape3d [3], which is written in R, and allows the creation of α-shapes as well as functionality like calculating its volume and visualizing the α-shape. Matlab also offers methods to create α-shapes [2], without the need to use any external library.

3

Computational Pipeline and System's Architecture

At the heart of our work lies an elaborate system, which aims to answer biological questions regarding the relations between neurons and blood vessels in the brain. In this chapter, we describe the various components of the system, explain the rationale behind the algorithmic and implementation choices that we have made, discuss some of the difficulties that we have encountered along the way and draw conclusions from the design and development phases.

3.1 Neurons Orientations

As already described in Section 2.1.2, an orientation can be defined by three angles, α, β, γ, so the object is rotated by γ degrees around the z-axis, by β degrees around the y-axis and by α degrees around the x-axis. In order to find a placement of the neuron that is structurally feasible among the blood vessels, we test various orientations in each position for collision. Due to biological constrains, we limit the angles of the rotations so that $|\alpha| \leqslant 5°, |\beta| \leqslant 5°$, and γ can be any angle. In addition, we use only integer degrees with a resolution of 1°. These choices result in 43,560 different orientations for every position of the neuron.

3.2 Data Representation

In our work, we used one set of vascular data and eighteen neurons, all in the form of sets of balls, as described in Section 2.1.1. The vascular data represents blood vessels in a cuboid of 1053 by 987 by 1091 μm^3, and is composed of about 680,000 balls with different radii,

sampled along the blood vessels. The neurons are smaller—they differ in size, but all of them fit in that cube, and their representation consists of about 7,000 to 21,000 balls each.

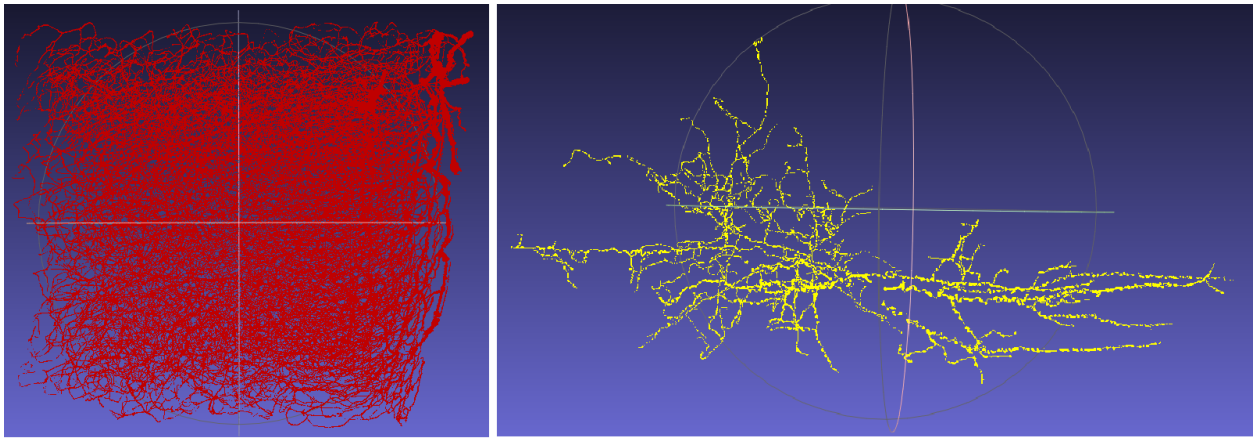

Figure 3.1: **A visualization of the blood vessels and a neuron used in out experiments.** (a) The blood vessels. (b) One of the eighteen neurons.

3.3 The Naive Solution

The first solution that was tried was using a brute-force approach. In this method, for each position and orientation the neuron is placed in, each ball of the neuron is tested against each ball of the vascular data. For every pair of such balls that are sufficiently close, i.e., the distance between them is smaller than the sum of their radii, a collision is reported. Not surprisingly, this approach, with time complexity of $\mathcal{O}(nk)$ (where n is the number of balls in the vascular data and k is the number of balls in the neuron), turned out to be too inefficient to yield significant results in acceptable times. Even though some improvements could be made to this approach, we quickly abandoned it and looked for more sophisticated methods to perform the task at hand.

3.4 Pipeline Description

As described in Section 2.1.3, many libraries for collision detection exist, each with its own advantages. We chose to use FCL, and built our pipeline around it. In this subsection, we walk through the different stages in the pipeline and explain how they work. The main features are illustrated in Figure 3.2.

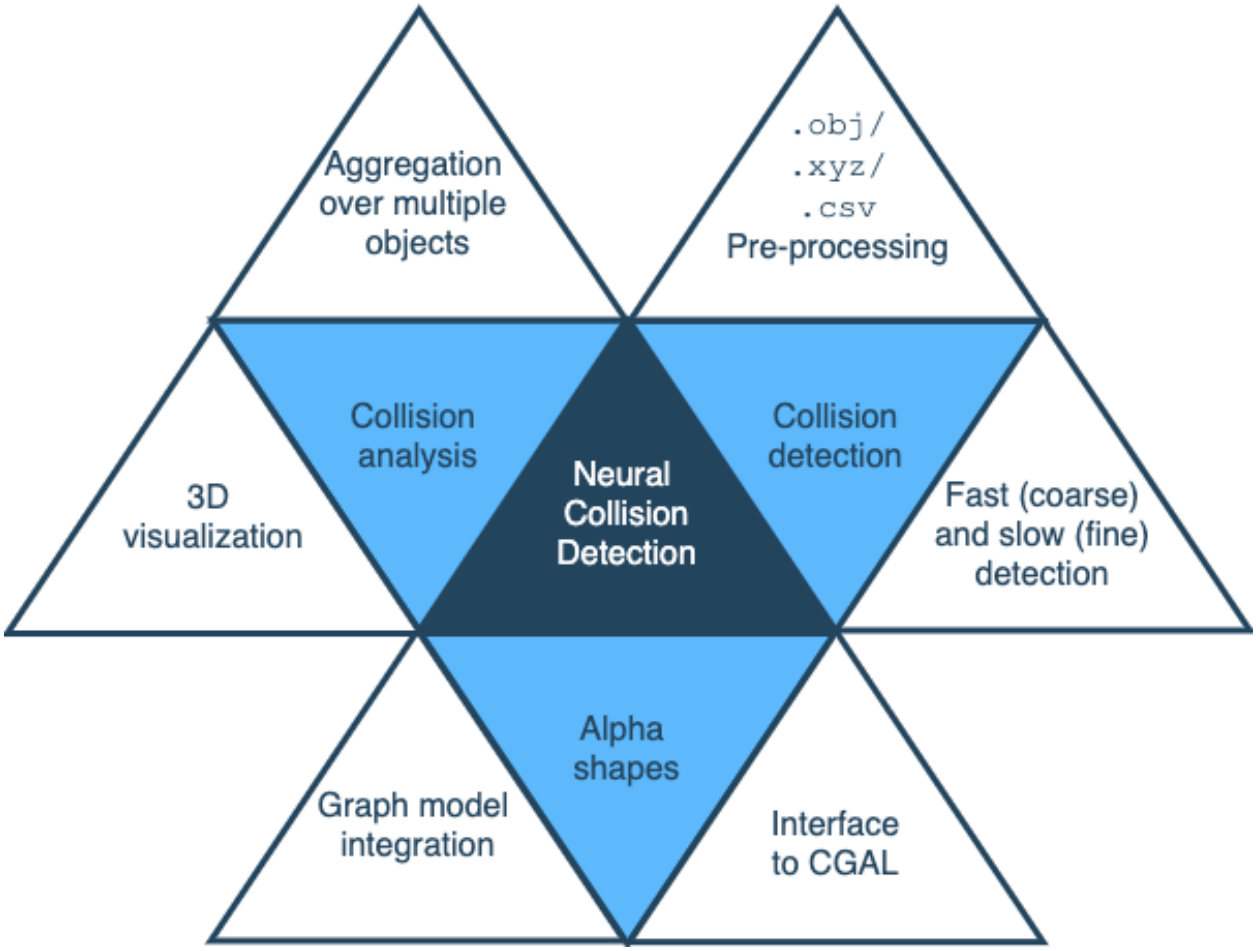

Figure 3.2: **Features Overview.**

3.4.1 Raw Data to Triangle Meshes

We start the process with raw vascular and neural data, where each object is represented as a set of balls as described above. The first step is to transform them into triangle meshes, so we can use FCL to calculate the collisions and use common visualization programs to examine the objects.

To transform a single object into a triangle mesh, we calculate its height, width and depth, and create an occupancy-grid with those dimensions. Then, we use Matlab's isosurface() function to transform it into a triangle mesh. It gives us two csv files (one for vertex and one for triangles), which we combine to a single obj file, which is readable by most visualization programs, like MeshLab [13].

The most time-consuming step is the call to isosurface function. Its time complexity is $\mathcal{O}(n)$, where n is the number of cells, which in the case of the blood vessels gets to 10^9. However, since it happens only once for each object, we did not invest time in optimizing it.

It is important to note that the output of this step—the triangle meshes—are much more

complicated than the original representation. While there are about 680,000 balls in the original vasculature representation, there are roughly 17,000,000 triangles in the vascular triangle mesh. The ratio is lower but still significant for the neurons—7,000 balls are transformed into 50,000 triangles, while 21,000 balls are transformed into 144,000 triangles. This resolution is dictated by Matlab's isosurface() function. We could have added a simplification step to reduce these numbers, but we chose not to do it, as discussed in 3.7. It means that while we are able to use more sophisticated algorithms with the triangle meshes, they are larger and heavier to work with. Nevertheless, it is still much more efficient than using the naive approach on the original data.

3.4.2 Triangle Meshes to Basic Collision Data

Once we have triangle meshes of the blood vessels and the neurons, we can calculate how intensively they collide in different placements, meaning how many contact points they have and where they are located. A contact point is a point which resides on two triangles, one from each object. It means that the two objects may either touch or intersect each other at this point. Note that since the triangles are always on the boundaries of the objects, the contact points will also always be there.

First, we had to define what placements of neurons are relevant for our simulation. Inside the bounding box of the blood-vessels region, we took a set of hundreds of points known to be centers of neurons, but for which it is unknown which neuron should be there and in what orientation. A valid placement for a neuron is one that the center of the neuron is one of those points, and the neuron is rotated by some α, β, γ, as described in Section 3.1. In addition, we eliminate placements in which part of the neuron goes out of the blood vessels region.

Then, we use the main component in our pipeline—NCD (Neural Collision Detection), which is responsible for the actual collision detection. At a high level, NCD works in the following manner:

- The program receives a single neuron, a single vascular set and a list of possible centers for the neuron.
- For each center, the program calculates the collisions between the two objects, in every valid orientation of the neuron. For each such placement, the program stores the number of contact points, and possibly the contact points themselves.
- For each center, the program outputs the top ten placements with minimal number of contact points, as "best placements". Only those placements will be used later in our pipeline.

Since this step is very time consuming, and since we have a large data set—we test many positions of different neurons, we consider many orientations and the objects are very complex, made of millions of triangles, much effort has been invested along the way to speed up the process. First, we used FCL for the collision calculation, which is known for its efficiency.

Second, we calculate the collisions in parallel, since the computations for the different orientations do not depend on the results of one another. Third, we use "batch mode", which means that we calculate the collisions for several placements of the neuron in one run, and then the pre-processing stage—the creation of the OBB-tree models by FCL—is performed only once. Using these measures, we managed to make this step feasible, though still rather expensive.

As described in [24], the time complexity of the pre-processing stage—building the OBB-tree models—is $\mathcal{O}(n \log^2 n)$, where n is the total number of triangles in both of the models. A single collision test between a neuron and the blood vessels is more complex to analyze, since in the general case, the number of collision tests between two bounding volumes and between two triangles is highly dependent on the shape and placement of the objects. In our case, where a neuron has far fewer triangles than the blood vessels and is contained inside the blood vessels region, the typical situation is that each one of the neuron's bounding boxes is compared to a constant number of boxes bounding the blood vessels, but not the other way around. It means that if a neuron is composed of k triangles, and hence its OBB-tree is composed of $\mathcal{O}(k)$ bounding boxes, there will be $\mathcal{O}(k)$ collision tests between the neuron and the blood vessels.

3.4.3 Basic Collision Data to More Accurate Collision Data

The results from FCL are very helpful to estimate whether a placement has many contact points or a few, but we were not fully satisfied with them for the following reasons. In our experience, some of the contact points received from FCL were duplicate, and some were only on one of the objects. Because of that, and since the triangle meshes are only an approximation of the original data, we added another step to calculate directly the number and locations of the contact points. We do so by taking the "best placements" as found by the previous step, and use brute-force counting of contact points, similar to the method described in Section 3.3. The brute-force step is performed on the original data—the sets of balls—and so is not affected by any processing we performed on the data up till that point.

This brute-force step is not very efficient (as described in Section 3.3), but since we do it only on relatively few placements (up to 10 per center, instead of 43,560 per center), it is not a bottleneck in the process.

3.4.4 Accurate Collision Data to Meaningful Insights

Following the previous step, we know for each position in the list what placements are "best placements", the number of contact points and their locations. To have meaningful insights, we need to aggregate all of the accumulated data. Though there are many ways to do it, the most natural way seems to be as follows: for each neuron, we cast all the contact points of the best placements on the neuron in its base state (with no rotation), and see where the neuron tends to meet the blood vessels. We do so by iterating over all the results from the previous step, rotating back the locations of the contact points to the neuron at its base

state, summing up the number of contact points for each point on the neuron, and dividing by the number of results we used. The result we get is the *collision probability* of the point on the neuron. This way, we are able to look at statistical phenomena at a large scale, aggregating a large data set into a single data set per neuron.

3.5 Improvements Made along the Way

The pipeline above, though presented as a whole, was created in an iterative process, as improvements were made when needed. Most of the changes were made to improve performance, as we were using NCD ever more intensively, took it to its limit and needed even better performance.

In the beginning, NCD received a neuron object, a vasculature object and a single position, and returned a full report of the contact-points count of each valid orientation. Though it was a significant improvement over the brute-force method, it quickly became insufficient. The first improvement was to utilize parallelization, since different orientations are independent of one another. Naturally, introducing threads improved the speed significantly, and the pre-processing stage—the creation of the OBB-tree models—became the new bottle neck. Then, we introduced a new running mode called "batch mode", in which many placements are processed in a single run of the program, and the models are created only once. When the running time was sufficiently short, we could stop using simplification, as described in Section 3.7.

After running NCD on more and more data, its output—the table with the collision data—became too large and more difficult to work with, since it contained information on tens of thousands of orientations for each position of the neuron. To mitigate that, we filtered the results to contain only the information regarding the "minimal placements", those with minimal number of contact points, which are really the placements we are interested in. This simple change reduced the size of the output by several orders of magnitudes, and was very helpful.

When we continued to use NCD and run it on real data, we strove to increase the accuracy of the results. Some contact points reported by NCD appeared twice and some were close to the actual contact point but not exact. In addition to the fact that we worked on processed data and not the original data, we observed that the results were not absolutely correct, though we verified they were close to the truth. Because of that, we added the step described in Section 3.4.3, to benefit from both worlds—on the one hand, we utilized the robustness and efficiency of FCL in the filtering phase, and on the other hand, the extra step we added helped us be confident that the results we use are as accurate as possible.

3.6 Integration with FCL

3.6.1 Model Type

We examined two of the model types supported by FCL. The first is AABB, Axis Aligned Bounding Box, where each edge of the bounding boxes is parallel to one of the axes. The second model type is OBB, Oriented Bounding Box [24], where the algorithm tries to find a best fit box of arbitrary orientation. As a result, when using an AABB model the collision tests are faster, but the model needs to be modified when the object rotates [34]. Since we heavily use rotations in our process, we decided to use OBB models.

3.6.2 FCL Contact Points

Examining the contact points returned by FCL, we could not figure out their exact nature, since some of them were duplicate, and some of them were only on the surface of one object. Because of that, we decided to use FCL results only as hints, as explained in Section 3.4.3.

3.7 Simplification

Simplification of a triangle mesh means using less triangles to represent the same object [10]. While it often causes the model to be less accurate, it can significantly reduce its complexity and the running time and memory consumption of the program using it.

In the beginning, we used an optional simplification phase to reduce the total running time of our computations, using Fast Quadric Mesh Simplification [22]. Though it was very helpful in the development phase, eventually we decided not to use the simplification phase, and our results reported in the thesis were not computed on simplified object, to avoid the degradation in the accuracy of the results.

3.8 Alpha Shapes

A novel feature of our library is its bindings to the Computational Geometry Algorithms Library (CGAL) [50], a state-of-the-art C++ computational geometry toolbox. We use CGAL's α-shapes module to enhance the graph-structure of the 3D shape with information of its *critical alpha value* at every point, as defined in Section 2.2.2. This was done using the new Python bindings for the necessary parts of CGAL, written by Efi Fogel from the TAU CG Lab.

3.9 Visualization

In this work, we developed a fairly involved pipeline, aiming to support a variety of possible empirical results. Throughout the development process we put special emphasis on verifying the correctness of our results, and we perpetually tested our procedures in several ways. In this section, we describe how we used visualization to verify the correctness of our pipeline, and in the next section we will show how we used a simple toy example for program checking.

The primary tool we had for program checking is visualization—transforming the numbers and object files into shapes we can see and examine, which increased the confidence we had in the correctness of our results and actually allowed us to spot a few bugs. We used visualization in several steps along the way. First, when we transformed the original data into triangle meshes, we used MeshLab to view them side by side with the balls (as point cloud), and saw they matched. Second, when we started using FCL, we used simple shapes like cubes and pyramids in different placements, and checked if we see collisions in their visualization, when FCL reported they collide. Third, we used visualization to verify the correctness of our program, by checking if its recommended placements with minimal number of contact points are actually true, by picking a few such results and verifying them visually.

3.10 Toy Example

Another way to verify that NCD works correctly was experimenting with a toy example to test the program in a simple, controlled environment. We focused on two key aspects of the program:

- Finding a placement with minimal collisions among many options.
- Rotating an object in order to find the best orientation for a given position.

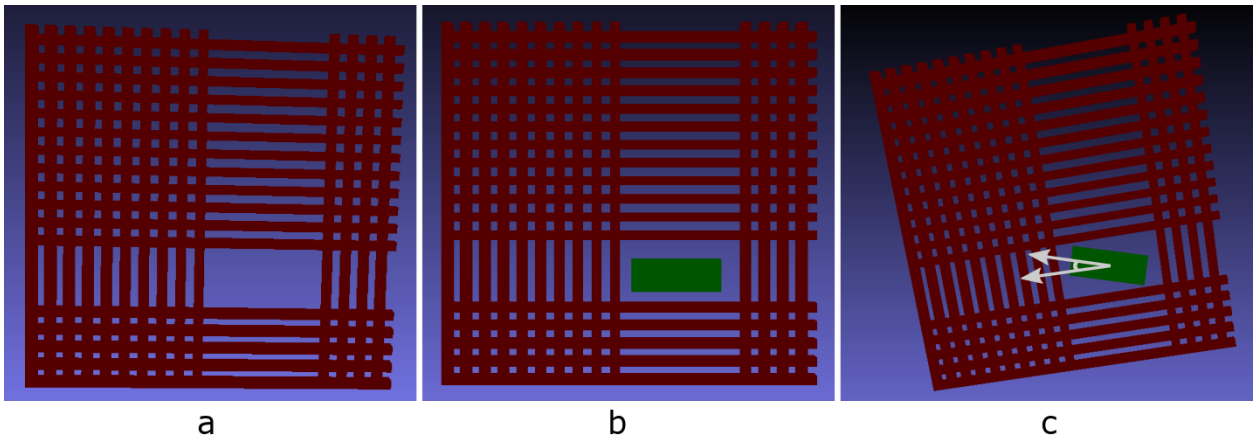

Figure 3.3: **2D Simplified Setup.** (a) The grid G, (b) G and B in an axis-aligned orientation, (c) G and B rotated by different angles.

3.10.1 Setup

There are two objects in the toy example—a box B and a three-dimensional grid G of thin, long boxes, with some missing boxes leaving an empty space slightly larger than B, in a random position. We conducted two experiments, as explained below.

3.10.2 First Experiment

In the first experiment, we wanted to verify that NCD can report only placements with no collisions, among many options. We disabled the rotation element in NCD, and sampled many random points within the volume of G, as possible locations for B. We ran NCD on that input, and examined the reported placements with no collisions, annotated by R. We expected the Minkowski sum (the addition of each point of an object with each point of another object) of the convex hull of R and B to fit in the hole in G, and the more sample points we have, the more the Minkowski sum will tend to the shape of the hole. As Figure 3.4 shows, as we increased the number of sampled points, we got that the Minkowski sum tends to the empty area, and at any rate does not exceed it.

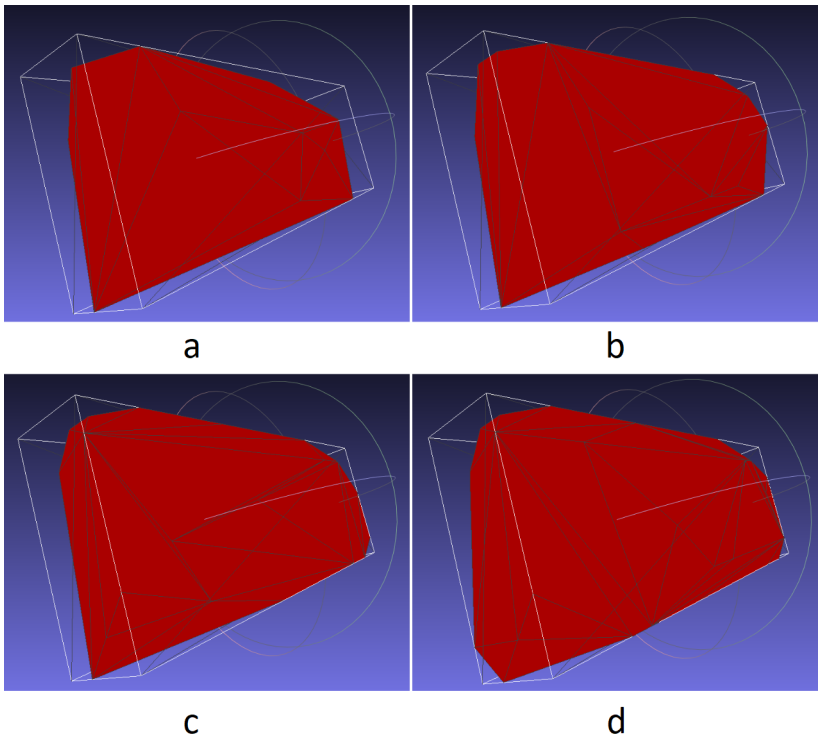

Figure 3.4: **Results of the First Experiment.** The convex hull of R with increasing number of centers. (a) 5,000 points, (b) 10,000 points, (c) 20,000 points and (d) 30,000 points. The convergence to the surrounding box shows that NCD is able to successfully identify the positions with no collision.

3.10.3 Second Experiment

In the second experiment, we tested the rotation functionality of our program. We applied a random orientation to G, and sampled many random points within its volume. After running NCD, we iterated over all of the collision-free placements, and recorded the differences between the reported rotation angles and the orientation of G, and calculated the distribution for each axis separately. We expect the distribution to be normal, with the mean close to 0. Indeed, Figure 3.5 shows that we get a normal distribution, with a mean of -0.53, and a peak at 0.

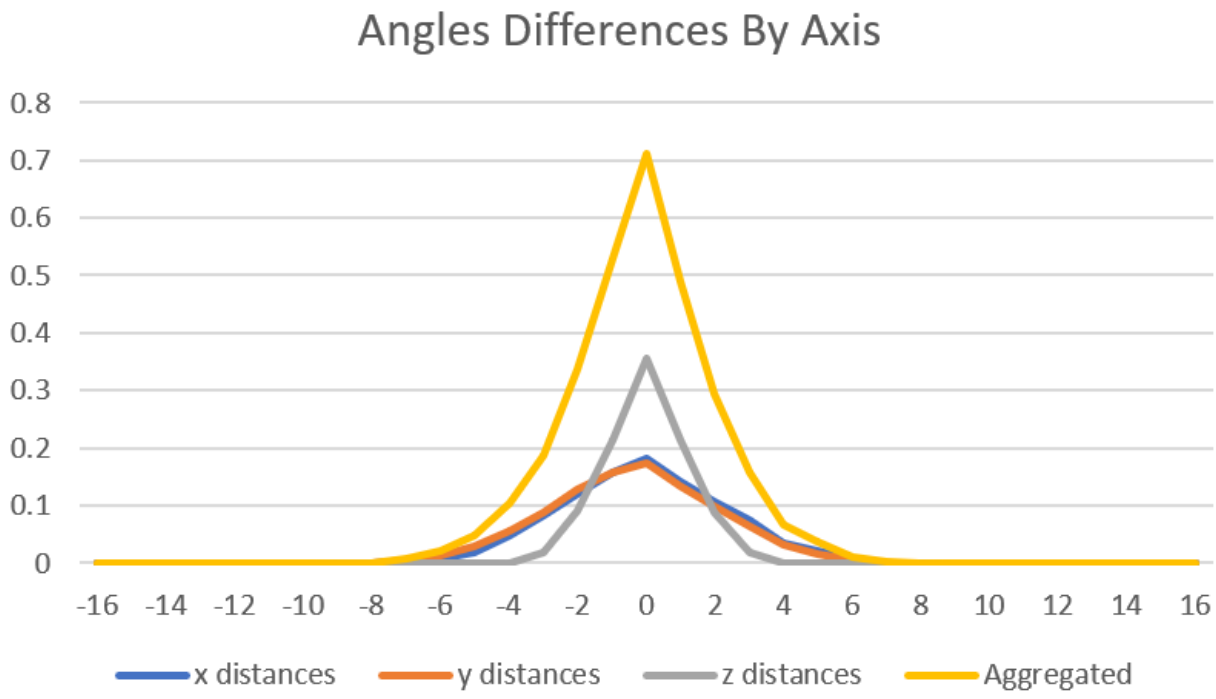

Figure 3.5: **Results of the Second Experiment.** Distribution of angles differences. The normal distribution with mean close to 0 confirms our hypothesis, and shows that NCD is able to find collision-free placements by rotating the object and find the optimal orientation.

4 Results

This chapter is dedicated to the biological aspect of the work. In Section 4.1 we explain the biology reasoning for the necessity of our solution, and point to the lack of existing solutions. In Section 4.2 we show some experimental results using NCD with real neurovascular data. These results demonstrate the various capabilities of our system, including data aggregation and visualization, combining neuron's properties with collision probabilities data (as defined in Section 3.4.4) and alpha-shapes integration.

4.1 Biology Background

The unique morphology of types and sub-types of neurons in the mammalian brain has drawn researchers to extensively investigate their structure and its relation to function [38, 42]. The neurons have an elaborate network of connections between each other via neuron-to-neuron synapses, but they also continuously communicate with other cell types that surround them, including glial cells and those composing the neurovascular unit [12]. As a result, it is expected that the neuron's geometric properties will also be affected, and could even be modulated, by the physical interactions with the cells in its surroundings [6, 39, 41], and vice versa.

This complex 3D environment may be investigated from a variety of methodological approaches and angles. One such approach emphasizes geometric interactions and uses computational geometry as its framework of reference [28, 49]. With this point of view neurons, for example, can be thought of as three-dimensional rigid body, which are often represented as graphs, with their soma (the cell body of the neuron) being the parent node which branches out to different sub-graphs that describe the axon (a branch of the neuron responsible to transmit information from the soma to other cells) and dendrites (branches of

the neuron responsible to receive information from other cells and propagate it to the soma). This structure is useful when seeking information that is closely tied to the morphological properties of the neuron and its surroundings, such as conductance along a branch and neuronal fiber distance between two spines, and can also be helpful when modeling the interaction of a pair of neighboring neurons. As a concrete example, Kashiwagi and colleagues [28] used computational-geometry-based methods to correlate evolving spine morphology with learning in an awake animal, and were able to find differences in the morphological structure between different strains of mice.

Questions such as the ones detailed above require designated *in silico* frameworks, either commercial or open-source; see [17] as well as Neurolucida by MicroBrightFields, Inc., Colchester, VT, USA. A prime example of a missing feature in existing frameworks is the gap between the single object level and macro-scale analysis. Several existing frameworks have focused on the single-cell level, letting their users analyze and visualize neurons, vasculature and other cell types in the brain with incredible precision [7, 35]. Other frameworks provide an API that assists with multicellular networks and modeling of cortical columns, providing answers to questions that relate to encoding and decoding of information in the brain [18, 44]. The lack of computational environments that target cell-to-cell interactions and single-to-many relationships, motivated us to develop NCD as a means to bridge this gap.

4.2 Experimental Results

4.2.1 Materials

Eighteen three-dimensional models of VIP-expressing GABAergic neurons [23], which form the majority of data from [42], were kindly supplied to us by the authors. These barrel-cortex neurons were imaged in the original study using structured illumination, with the raw data being post-processed by Neurolucida to form the final model of the cell. The authors of [42] also identified the exact cortical layer of each of the neurons using DAPI staining (a technique that uses a special fluorescent stain that binds to specific regions in the DNA, to research cells in the microscopic level).

A cortical vascular network spanning 1053 by 987 by 1091 μm^3 from [9] was used. The network was skeletonized and its graph properties were extracted using the software described in [51]. The positions of the neuronal soma were also recorded and used in our computational pipeline. Both the neurons and the vasculature were given in the form of sets of balls, as technically described in Section 3.2.

4.2.2 Exemplary Visualization and Analysis of Collision Data

Figure 4.1 and Figure 4.2 show the capabilities of NCD to integrate collision and morphological data together via its underlying graph structure. We tested here for the collision

of the neurons obtained from [42] with vascular data from [9] and analyzed the resulting distributions.

First, in Figure 4.1 we see NCD-facilitated 3D renderings of neurons and their collision data. The two neurons shown, from layer II/III and from layer IV in the neocortex, were overlayed with their collision probabilities, with the size of the disc in correlation to the magnitude. The original 3D image of the neuron is also shown in the inset, and the colors code the axon (green) and dendrites (orange). The dark sphere in the insets marks the location of the cell's body. These renderings may inform the viewer about disparities in collision probabilities between parts of the objects and can enable a more detailed analysis of the collision distribution.

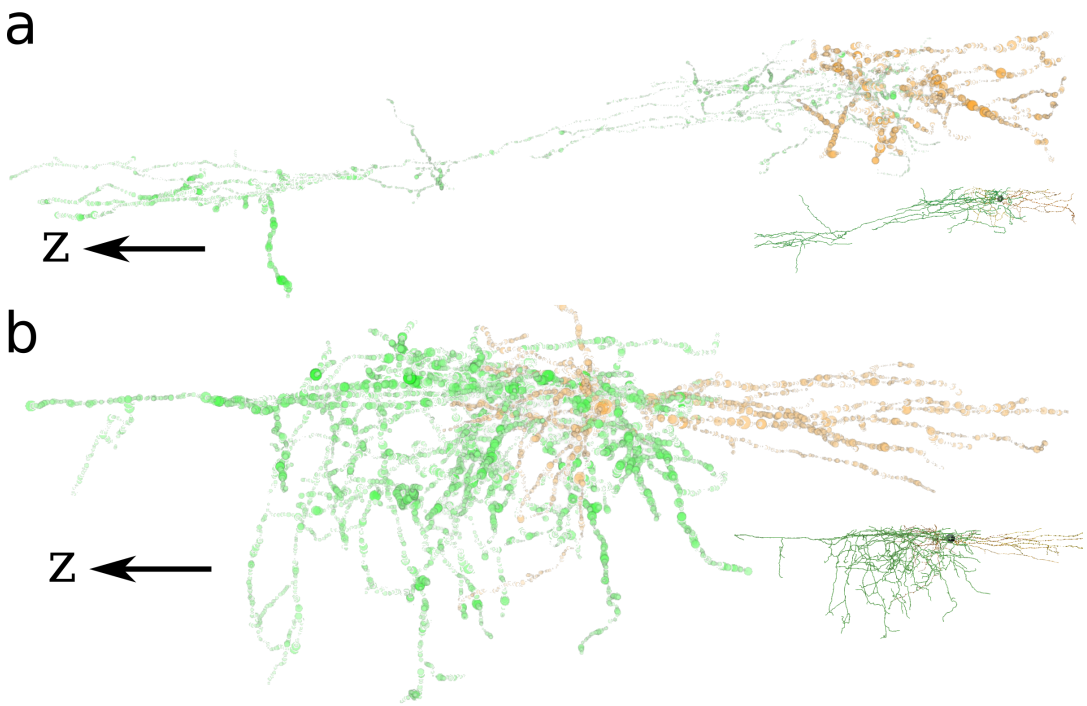

Figure 4.1: Exemplary visualization of neuron collisions using NCD. 3D models of two representative cortical neurons from layer II\III (a) and layer V (b). The two distinct morphologies also generate a different collision distribution, as seen in the central part of neuron (a) which rarely collides with the surrounding vasculature. The z direction points toward the white matter (ventrally). The length of the arrow at (a) is 100 μm, while at (b) the length of the arrow is 60 μm. The napari application [48] was used for 3D renderings of the morphological structures and collision data. The insets show the neurons without collision data but with the same color coding, and with a dark sphere signaling the soma's position.

Second, in Figure 4.2 we utilize the underlying graph structure to evaluate the distribution of collision probabilities for a single neuron and for groups of neurons. Panel (a) showcases the chance for a collision as we step further away from the soma for both axon (green) and dendrites (orange). The distributions are slightly shifted along the x-axis due to the different

lengths of the axon and dendrites, but remain quite similar. The bottom part of the figure shows aggregations of that same data from neurons in layer II\III and (b) the neurons from other layers. This division highlights the dramatic change in morphology between the two subgroups, which is also visible in Figure 4.1, and how it affects the collision probability distribution.

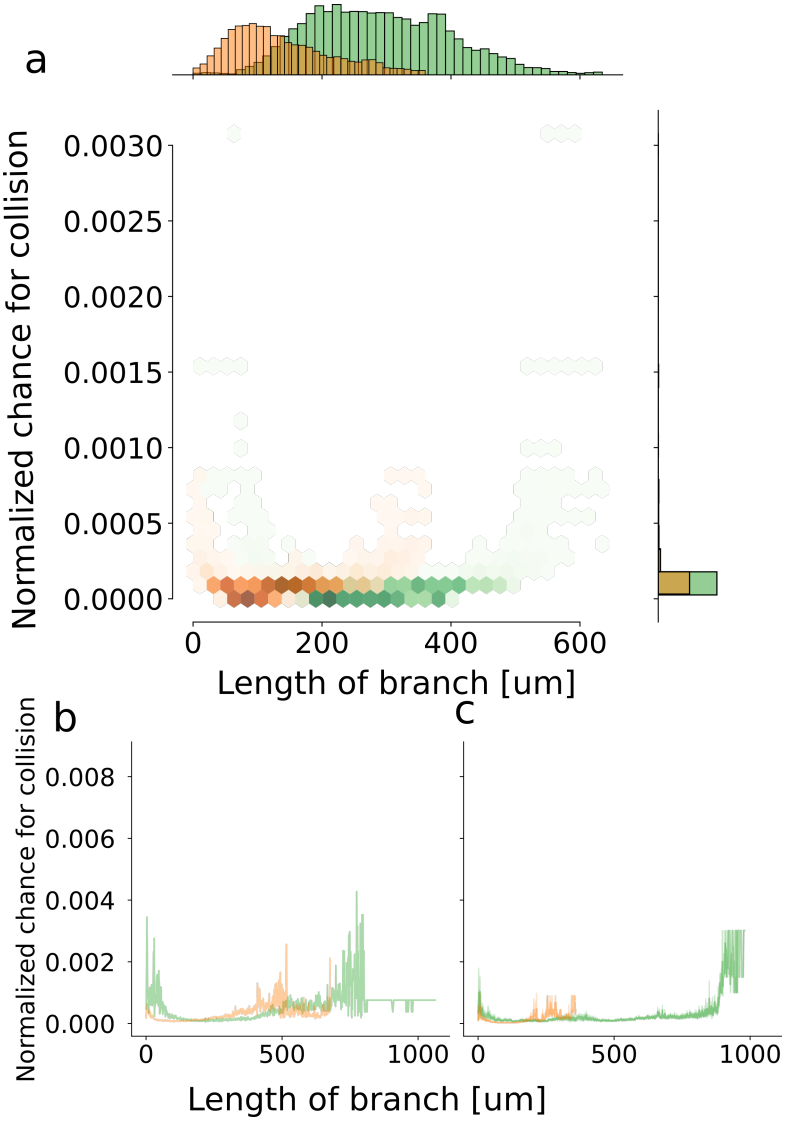

Figure 4.2: Collision probability as function of distance from the soma. (a) Example "hexbin" density plot showcasing the collision distribution of dendritic (orange) and axonal (green) population for the single neuron shown in the inset. (b-c) Layer II\III (b) collision distribution for the axonal and dendritic population as opposed to layer V neurons (c). The dendrites of layer II\III neurons are generally more collision prone, while their axons show an increased collision probability at the middle and extreme parts.

4.2.3 Exemplary Analysis of Alpha Shapes

As noted, NCD is able to calculate the alpha-shape spectrum for each node along the neuron using newly-generated Python bindings to the relevant components of the CGAL library [1]. Further, NCD will also calculate the critical alpha value per point, as explained in Section 2.2.2.

Figure 4.3 shows the neurons from Figure 4.1 with the critical alpha radius overlayed onto their corresponding morphology. The insets show close-ups for specific areas in the neuron, which exhibit a significant difference between the alpha values of nearby axons and dendrites, suggesting that the morphological difference between these two nearby areas might bear biological significance. The largest radius in the image corresponds to the maximal allowed radius by the surrounding vasculature.

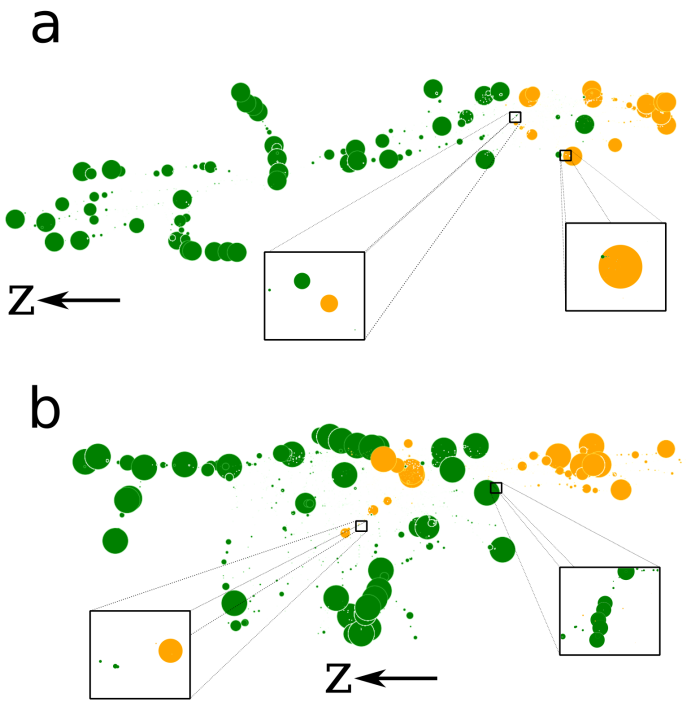

Figure 4.3: Exemplary 3D visualization of alpha values for the neurons shown in Figure 4.1. (a-b) The alpha values were integrated into the neuronal graph structure so that the size of the disc correlates to the critical alpha value of that point. The highlighted regions show how proximate regions of dendrites (orange) and/or axons (green) can have very different corresponding critical alpha values. The z direction points toward the white matter (ventrally). The length of the arrow at (a) is 100 μm, while at (b) the length of the arrow is 60 μm. The napari software was used for 3D rendering.

[1] Efi Fogel, Personal communication.

On top of the visualization layer, NCD also provides a large array of functions and scripts to analyze the alpha-shape critical values in relation to the other quantities that were introduced before. These include, for example, correlations between the alpha-shape values and the distance of each node to the soma. In addition, NCD also compares the collision probability of each node to its alpha value and to other quantities (Figure 4.4).

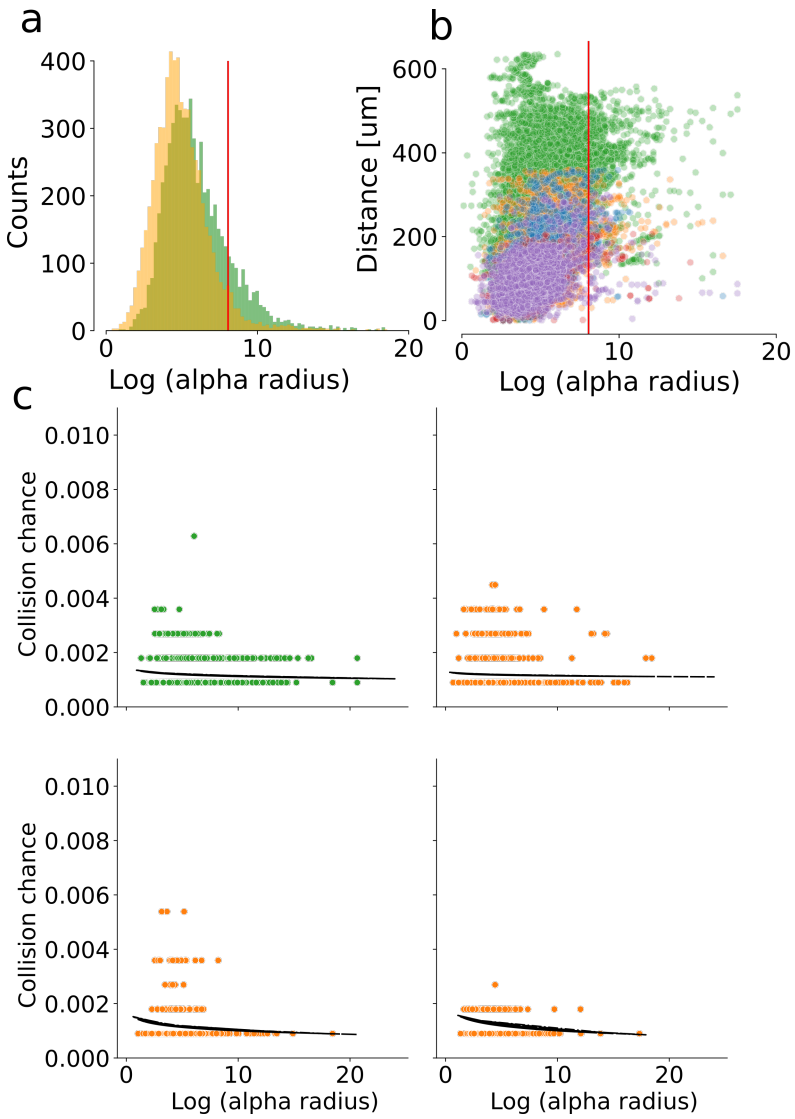

Figure 4.4: Alpha shape analysis examples using built-in NCD tools. (a) Critical alpha value distribution for a single neuron. The green histogram is for points on the axon while the orange depicts all dendrites. (b) The distribution of the topological distances from the soma as a function of the alpha values, with the respective trend lines. Green points are from the axon, while the rest of the points are colored by the specific dendritic tree they belong to. (c) Collision probability as a function of the critical alpha value for each point on the neuron, colored by neurite type. Data from the first two panels is from neuron (a) of Figure 4.1.

As seen in Figure 4.4, NCD facilitates a variety of analysis methods with its built-in

functions. The neurons' alpha-shape spectrum is pre-computed using novel CGAL bindings which were developed by Efi Fogel per our request, and are publicly available. Then, this data can be further processed by functions in NCD's `alpha_shapes` module. These methods include alpha-values distribution and correlation of the critical alpha values to quantities such as topological distance from the soma (the length of the shortest path on the neuron graph, from the soma to the point) and collision probability. In Figure 4.4, the vertical red line points to the maximal vascular critical alpha value, which we use as the theoretical limit for the neurons' alpha values once they are positioned inside the vasculature.

5 Conclusion

5.1 The Development Process

This work shows yet again that working on large data sets is difficult. Indeed, this challenge arises in many problems and situations across almost all fields of science and technology. In our work, this challenge was twofold—we worked with highly complicated objects, and we needed to run our collision tests in a lot of different settings, to get enough reliable data. We addressed these difficulties in several ways. First, in the development phase, we used smaller data sets, like toy examples and simplified versions of the real neurons and vasculature. Second, we utilized visualization in several steps of the work, since many times it proved to be the best way to grasp and handle a large data set. Third, we iteratively improved our solution, as we encountered more and more challenges when working with the real data. This way of agile development—being open to change and adjust the solution upon new demands—is an important approach in today's software development, especially when the unknown is significant. Fourth, we continuously checked ourselves and verified that the code is correct, so when we moved to larger data sets, we had strong confidence in our program and the correctness of the results.

Another advantage of being agile was that the task definition changed slightly along the way. Our team consists of neuroscientists as well as computer scientists, and as each of us brings a different expertise, and the goal was to investigate the unknown, reaching the final conclusions and results could not have been fully planned from the beginning. As we progressed in our work and new ideas came up, they were researched and implemented. An example for that is using alpha-shapes as a measure of concealment, which was thought of and added only halfway through the work.

The fact that we come from different fields and bring different skills and expertise was

a huge advantage in this work. These days, many scientific projects rely on experience and knowledge gathered from two or more fields, to achieve novel and exciting results, which could not have been achieved with the tools from one field alone. In this aspect, our work is not different—combining a deep understanding of the brain on one hand, and state-of-the-art computational geometry tools and algorithms on the other hand, was crucial to the development and successful usage of NCD.

Regarding the library we chose to put in the heart of our program, FCL, I am divided whether we made the right choice. On the one hand, the library is indeed rightfully considered the state-of-the-art in collision computation and is wildly used, and its performance for our needs has been fairly good. On the other hand, it was more difficult to work with than expected—parts of its documentation is somewhat succinct, the installation process is not trivial and understanding its semantics was not always easy. At the end, we produced meaningful results with good performance by relying on FCL, so I believe it was the right decision.

5.2 Future Work

Our work can be extended and advanced in two major directions—improving the pipeline, and using it in different ways to produce more results.

The pipeline can be improved in various ways. First, the triangular mesh representation can be enhanced. With sophisticated enough algorithms, it may consist of fewer triangles and be easier to work with, while not losing from its accuracy and possibly representing the actual objects even better. Second, for a given position, we test the neuron for collisions many times separately with different but similar orientations. It is plausible that with a new technique, it can be performed with a single continuous collision detection test, which might be more accurate and improve the running time of the test.

As demonstrated by some exemplary results that were produced by running the pipeline, the search for new insights is not exhausted, as new hypotheses can be proposed and tested using our software. We used mainly alpha-shapes and topological distance from the soma as properties of a point on the neuron. However, more properties—biological and geometric—can be used and tested against the collision probability of a given point. In addition, we focused on points with high likelihood of collision on the neuron, but it is possible that by shifting the focus to the blood vessels, more phenomena can be revealed.

Abstract

The analysis of neuronal structure and its relation to function has been a pillar of neuroscience since its earliest days, with the underlying premise that morphological properties can modulate neuronal computations. It is often the case that the rich three-dimensional structure of neurons is quantified by tools developed in fields other than neuroscience, such as graph theory or computational geometry; nevertheless, some of the more advanced tools developed in these fields have not yet been made accessible to the neuroscience community. Here we present NCD, Neural Collision Detection, a library providing high-level interfaces to collision-detection routines and alpha-shape calculation, as well as statistical analysis and visualization for 3D objects, with the aim to lower the entry barrier for neuroscientists into these worlds. Our work here also demonstrates a variety of use cases for the library and exemplary analysis and visualization, which were carried out with it on real neuronal and vascular data.

The library at the heart of this work, NCD, uses state-of-the-art computational geometric techniques and algorithms to perform collision detection at scale, with major focus on performance and accuracy. We designed an elaborate pipeline, which starts from the complex components in the neurovascular system—the blood vessels and the neurons in the brain—and finishes with detailed analysis of collision data of the different components, including correlation between the likelihood of a collision at some points on the neuron and geometric properties of these points like graph distance and the critical values of their alpha-shapes.

Bibliography

[1] Alpha shape toolbox. https://alphashape.readthedocs.io/en/latest/. Accessed: 2021-03-12.

[2] alphaShape - Matlab. https://www.mathworks.com/help/matlab/ref/alphashape.html. Accessed: 2021-03-12.

[3] alphashape3d: Implementation of the 3D alpha-shape for the reconstruction of 3D sets from a point cloud. https://cran.r-project.org/web/packages/alphashape3d/index.html. Accessed: 2021-03-12.

[4] BEPUphysics, 3D rigid body physics engine. https://www.bepuentertainment.com/. Accessed: 2021-01-09.

[5] ncollide, 2D and 3D collision detection library. https://www.ncollide.org. Accessed: 2021-01-09.

[6] J. Bastian and J. Nguyenkim. Dendritic modulation of burst-like firing in sensory neurons. *Journal of Neurophysiology*, 85(1):10–22, 2001. PMID: 11152701.

[7] A. S. Bates, J. D. Manton, S. R. Jagannathan, M. Costa, P. Schlegel, T. Rohlfing, and G. S. Jefferis. The natverse, a versatile toolbox for combining and analysing neuroanatomical data. *Elife*, 9:e53350, 2020.

[8] G. v. d. Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of graphics tools*, 2(4):1–13, 1997.

[9] P. Blinder, P. S. Tsai, J. P. Kaufhold, P. M. Knutsen, H. Suhl, and D. Kleinfeld. The cortical angiome: an interconnected vascular network with noncolumnar patterns of blood flow. *Nature neuroscience*, 16(7):889–97, jul 2013.

[10] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Lévy. *Polygon Mesh Processing*. A K Peters, 2010.

[11] I. Bzhikhatlov and S. Perepelkina. Research of robot model behaviour depending on model parameters using physic engines bullet physics and ODE. In *2017 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, pages 1–4. IEEE, 2017.

[12] B. Cauli, X.-K. Tong, A. Rancillac, N. Serluca, B. Lambolez, J. Rossier, and E. Hamel. Cortical GABA interneurons in neurovascular coupling: Relays for subcortical vasoactive pathways. *Journal of Neuroscience*, 24(41):8940–8949, 2004.

[13] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference*, volume 2008, pages 129–136, 2008.

[14] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff, 1995.

[15] E. Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, page 1. 2015.

[16] E. Coumans and Y. Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2019.

[17] H. Cuntz, F. Forstner, A. Borst, and M. Häusser. One rule to grow them all: a general theory of neuronal branching and its practical application. *PLoS Comput Biol*, 6(8):e1000877, 2010.

[18] J. Dyhrfjeld-Johnsen, J. Maier, D. Schubert, J. Staiger, H. Luhmann, K. Stephan, and R. Kötter. Cocodat: a database system for organizing and selecting quantitative data on single neurons and neuronal microcircuitry. *Journal of Neuroscience Methods*, 141(2):291–308, 2005.

[19] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Transactions on information theory*, 29(4):551–559, 1983.

[20] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics (TOG)*, 13(1):43–72, 1994.

[21] T. Erez, Y. Tassa, and E. Todorov. Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE, 2015.

[22] M. Garland and P. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Visualization '98*, October 1998.

[23] Y. Gonchar, Q. Wang, and A. Burkhalter. Multiple distinct subtypes of GABAergic neurons in mouse visual cortex identified by triple immunostaining. *Frontiers in Neuroanatomy*, 2:3, 2008.

[24] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, 1996.

[25] H. Har-Gil, Y. Jacobson, A. Proenneke, J. F. Staiger, O. Tomer, D. Halperin, and P. Blinder. Neural collision detection: an open source library to study the three-dimensional interactions of neurons and other tree-like structures. https://www.biorxiv.org/content/10.1101/2021.07.20.452894v1, 2021.

[26] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210, 1996.

[27] Y. Jacobson, H. Har-Gil, D. Halperin, and P. Blinder. Neural collision detection. https://github.com/PBLab/neural_collision_detection, 2021.

[28] Y. Kashiwagi, T. Higashi, K. Obashi, Y. Sato, N. H. Komiyama, S. G. Grant, and S. Okabe. Computational geometry analysis of dendritic spines by structured illumination microscopy. *Nature communications*, 10(1):1–14, 2019.

[29] T. J. Kelly. Neocortical virtual robot: A framework to allow simulated brains to interact with a virtual reality environment, 2015.

[30] C. Kirst, S. Skriabine, A. Vieites-Prado, T. Topilko, P. Bertin, G. Gerschenfeld, F. Verny, P. Topilko, N. Michalski, M. Tessier-Lavigne, and N. Renier. Mapping the fine-scale organization and plasticity of the brain vasculature. *Cell*, 180(4):780 – 795.e25, 2020.

[31] T. Larsson and T. Akenine-Möller. Collision detection for continuously deforming bodies. In *Eurographics*, 2001.

[32] S. M. LaValle. Planning algorithms. *Cambridge University Press*, 2006.

[33] M. Lin and J. Canny. A fast algorithm for incremental distance calculation. *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 1008–1014 vol.2, 1991.

[34] M. C. Lin, D. Manocha, and Y. J. Kim. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, chapter 39, pages 1029–1056. Chapman & Hall/CRC, 2018.

[35] J. Lu. Neuronal tracing for connectomic studies. *Neuroinformatics*, 9(2-3):159–166, 2011.

[36] A. Mizrahi, E. Ben-Ner, M. J. Katz, K. Kedem, J. G. Glusman, and F. Libersat. Comparative analysis of dendritic architecture of identified neurons using the hausdorff distance metric. *Journal of comparative neurology*, 422(3):415–428, 2000.

[37] A. Munawar and G. Fischer. A surgical robot teleoperation framework for providing haptic feedback incorporating virtual environment-based guidance. *Frontiers in Robotics and AI*, 3:47, 2016.

[38] U. Nägerl, N. Eberhorn, S. B. Cambridge, and T. Bonhoeffer. Bidirectional activity-dependent morphological plasticity in hippocampal neurons. *Neuron*, 44(5):759 – 767, 2004.

[39] N. Ofer and O. Shefi. Axonal geometry as a tool for modulating firing patterns. *Applied Mathematical Modelling*, 40(4):3175 – 3184, 2016.

[40] J. Pan, S. Chitta, and D. Manocha. FCL: A general purpose library for collision and proximity queries. In *2012 IEEE International Conference on Robotics and Automation*, pages 3859–3866, May 2012.

[41] V. C. Piatti, M. G. Davies-Sala, M. S. Espósito, L. A. Mongiat, M. F. Trinchero, and A. F. Schinder. The timing for neuronal maturation in the adult hippocampus is modulated by local network activity. *Journal of Neuroscience*, 31(21):7715–7728, 2011.

[42] A. Prönneke, B. Scheuer, R. J. Wagener, M. Möck, M. Witte, and J. F. Staiger. Characterizing VIP neurons in the barrel cortex of VIPcre/tdTomato mice reveals layer-specific differences. *Cerebral Cortex*, 25(12):4854–4868, 09 2015.

[43] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies. In *Computer graphics forum*, volume 21, pages 279–287. Wiley Online Library, 2002.

[44] Blue Brain Project. MorphIO. https://github.com/BlueBrain/MorphIO/, 2021.

[45] R. B. Rusu and S. Cousins. 3D is here: Point cloud library (PCL). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, 2011.

[46] G. G. Slabaugh. Computing Euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999.

[47] R. Smith et al. Open dynamics engine. https://www.ode.org, 2001-2021.

[48] N. Sofroniew, T. Lambert, K. Evans, J. Nunez-Iglesias, K. Yamauchi, A. C. Solak, P. Winston, G. Bokota, ziyangczi, G. Buckley, T. Tung, D. D. Pop, Hector, J. Freeman, M. Bussonnier, P. Boone, L. Royer, H. Har-Gil, A. R. Lowe, M. Kittisopikul, S. Axelrod, A. Rokem, Bryant, C. Gohlke, J. Kiggins, M. Huang, P. Vemuri, R. Dunham, T. Manz, and V. Hilsenstein. napari/napari: 0.4.0, Oct. 2020.

[49] S. A. Swanger, X. Yao, C. Gross, and G. J. Bassell. Automated 4D analysis of dendritic spine morphology: applications to stimulus-induced spine remodeling and pharmacological rescue in a disease model. *Molecular Brain*, 4(1):38, Oct 2011.

[50] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.1 edition, 2020.

[51] P. S. Tsai, J. P. Kaufhold, P. Blinder, B. Friedman, P. J. Drew, H. J. Karten, P. D. Lyden, and D. Kleinfeld. Correlations of neuronal and microvascular densities in murine cortex revealed by direct counting and colocalization of nuclei and vessels. *Journal of Neuroscience*, 29(46):14553–14570, 2009.

[52] S. Yue, F. C. Rind, M. S. Keil, J. Cuadri, and R. Stafford. A bio-inspired visual collision detection mechanism for cars: Optimisation of a model of a locust neuron to a novel environment. *Neurocomputing*, 69(13):1591 – 1598, 2006. Blind Source Separation and Independent Component Analysis.

[53] Z. Zhang, Y. Xin, B. Liu, W. X. Li, K. H. Lee, C. F. Ng, D. Stoyanov, R. C. Cheung, and K. W. Kwok. FPGA-based high-performance collision detection: an enabling technique for image-guided robotic surgery. *Frontiers in Robotics and AI*, 3:51, 2016.

תקציר

ניתוח מבנה הנוירונים שבמוח, והקשר של מבנה זה לפעולת הנוירונים היה אחד הנושאים המרכזיים בחקר המוח עוד מימיו הראשונים של התחום, מתוך הבנה שתכונות מורפולוגיות משפיעות ומווסתות את פעולת הנוירונים. במקרים רבים המבנה התלת מימדי המורכב של הנוירונים מאופיין על ידי כלים שפותחו בתחומים שאינם מדעי המוח, כמו תיאוריית הגרפים או גיאומטריה חישובית; עם זאת, כמה מהכלים המתקדמים ביותר בתחומים אלה עוד לא הונגשו לחוקרי מדעי המוח. בעבודה זו אנחנו מציגים את הספריה Neural Collision Detection, NCD, המספקת ממשקים לזיהוי התנגשויות וחישובי צורות אלפא, וכן ניתוחים סטטיסטיים והצגה חזותית של אובייקטים תלת מימדיים, בדגש על נוירונים וכלי דם במוח, במטרה להוריד את חסם הכניסה לעולמות אלה עבור חוקרי מדעי המוח. בעבודה זו, אנו גם מדגימים מספר שימושים לספרייה, כולל ניתוחי סטטיסטיים והמחשות ויזואליות, שהתבצעו תוך שימוש במודלים המייצגים נוירונים וכלי דם אמיתיים.

הספרייה שבלב העבודה, NCD, משתמשת בשיטות ואלגוריתמים הנמצאים בחזית העשייה של גיאומטריה חישובית, כדי לבצע בדיקת התנגשויות בקנה מידה רחב, תוך התמקדות משמעותית בביצועים ודיוק. בעבודה זו פיתחנו מערכת בעלת רב שלבית, שמקבלת כקלט אובייקטים מורכבים מהמערכת הנוירו-וסקולרית - נוירונים וכלי דם מהמוח - ומסיימת בניתוח מעמיק של נתוני ההתנגשויות של המרכיבים השונים זה בזה, וכן מספקת קורלציות בין ההסתברויות להתנגשות בנקודות מסויימות על הנוירון ותכונות גיאומטריות של נקודות אלה, כמו מרחק על גרף הנוירון וערכים קריטיים של צורות האלפא שלהם.

עיבוד גיאומטרי וזיהוי התנגשויות יעיל בתאים נוירו-וסקולרים

חיבור זה מוגש כחלק ממילוי הדרישות לקבלת
התואר "מוסמך במדעים" (M.Sc.) בבית הספר למדעי המחשב
באוניברסיטת תל-אביב

ע"י

יואב יעקבסון

העבודה הוכנה באוניברסיטת תל-אביב
בהדרכתו של פרופסור דן הלפרין
תמוז התשפ"א