TU WIEN Informatics

Echtzeit-Kollisionserkennung und Behandlung von Molekülen für Animation und Modellierung

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Johannes Pauschenwein
Matrikelnummer 01427350

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Tobias Klein

Wien, 11. Jänner 2022

Johannes Pauschenwein

Eduard Gröller

TU WIEN Informatics

Real-Time Collision Detection and Handling of Molecules for Animation and Modelling

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Johannes Pauschenwein
Registration Number 01427350

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Tobias Klein

Vienna, 11th January, 2022

Johannes Pauschenwein

Eduard Gröller

Erklärung zur Verfassung der Arbeit

Johannes Pauschenwein

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Jänner 2022

Johannes Pauschenwein

Danksagung

Ich möchte mich bei meinen Eltern und Freunden bedanken, dich mich während meines Studiums unterstützt haben. Besonderer Dank geht an meine beiden Betreuer Tobias Klein, der mir dieses spannende Thema für meine Bachelorarbeit ermöglichte und mir mit Rat und Tat beistand, und an Eduard Gröller, dessen Unterstützung und Geduld beim Schreiben der Arbeit viel geholfen haben.

Acknowledgements

I would like to thank my parents and friends who supported me during my studies. Special thanks go to my two supervisors, Tobias Klein, who made this exciting topic possible for my bachelor thesis and who supported me with word and deed, and to Eduard Gröller, whose support and patience, when writing the thesis, helped a lot.

Kurzfassung

Die molekulare Welt ist ein spannendes Wissenschaftsgebiet. Mit dem Fortschritt der Technik ist es endlich möglich, die Anordnungen der einzelnen Atome in Molekülen zu entdecken. Mit diesen Daten können komplexe Systeme wie Viren, Bakterien und Zellen, die aus Millionen von Atomen bestehen, modelliert werden. Solche Systeme werden oft als biologische mesoskalige Strukturen bezeichnet und es sind viele Visualisierungstechniken entstanden, um sie zu modellieren. Dies ist keine leichte Aufgabe, da solche Systeme sehr komplex sind und viel Berechnungszeit zum Rendern benötigen. Ein wesentlicher Bestandteil der Visualisierung ist die Kollisionserkennung und -auflösung, da sich Moleküle auch in der realen Welt nicht überlappen. Wir haben den "Fast Fixed-Radius Nearest Neighbors" Algorithmus der von Hoetzlein entworfen wurde, reimplementiert, um alle kollidierenden Atome zu finden, Kräfte zur Auflösung ihrer Kollision berechnet und auf die Moleküle angewendet. Um diese Methode zu testen, konstruierten wir ein Modell einer Lipiddoppelschicht und fanden heraus, dass es möglich war, Kollisionen mit bis zu vier Millionen Atomen in Echtzeit aufzulösen. Wir haben weitere Verbesserungen unseres Algorithmus überprüft und sind überzeugt, dass er als effiziente Kollisionserkennung für mesoskalige Umgebungen verwendet werden kann.

Abstract

The molecular world is an exciting field of science. Since technology has advanced it is finally possible to discover the arrangements of the single atoms in molecules. With this data, complex systems like viruses, bacteria and cells, made out of millions of atoms can be modeled. Such systems are often called biological mesoscale structures and many visualization techniques have been developed to model them. This is not an easy task as such systems are very complex and need a lot of processing time to render. An essential part of the visualization is the collision detection and resolving as in the real world molecules do not overlap. We reimplemented the Fast Fixed-Radius Nearest Neighbors algorithm that was designed by Hoetzlein to find all colliding pairs of atoms, calculated forces which are then applied on the molecules in order to resolve the overlaps. To test this method we constructed a model of a lipid bilayer and found that it was able to resolve collisions with up to four million atoms in real time. We reviewed further improvements of our algorithm and are positive that it can be used as a efficient collision detection for mesoscale environments.

Contents

CHAPTER 1

Introduction

Molecules are smaller than the wavelength of light, hence even with light microscopy, it is not possible to see them. Methods like x-ray crystallography, NMR spectroscopy, electron microscopy or atomic force microscopy help us to discover the arrangement of atoms in a molecule [Goo09]. With data collected by these techniques it is possible to model complex systems, such as viruses, bacteria and cells, made out of millions of molecules at atomic resolution. Such systems are often called biological mesoscale structures, because they are bridging the molecular (nanoscale) and the cellular biology. A visual representation of such models helps scientists to better understand the inner working of cells and to communicate findings to peers. Such visualizations were traditionally created by scientific illustrators using 3D modeling tools or are hand drawn, see Figure 1.1. Such visualizations are cumbersome to create, require vast domain knowledge, and sometimes lack the ability to interactively explore the mesoscale model.

To overcome these issues CellPACK [JAAA+15] describes the scene on a higher level of abstraction using so called "recipes". They gather data, mainly from cellular tomography, and derive a recipe to automatically generate a virtual model. This approach allows a broad audience from a diverse background to interact and create mesoscale models. The main drawback of this method is that the model creation takes long, depending on the complexity of the model, ranging from minutes to several hours. Changing parameters, like the number of proteins inside a virus, require a recreation of the model, which makes this method so time consuming.

A more experimental friendly environment could help scientists to communicate new domain knowledge or could be used for hypothesis validation. Klein et al. present cellVIEW [KAK+17], a tool which makes it possible to model the biological mesoscale at atomic resolution while allowing the user to interactively change parameters and explore the model. They do so by using a novel set of GPU (Graphic Processing Unit) algorithms that form a base for the fast generation of such structures. To speed up the generation process they make use of the high parallelism of the GPU that allows thousands of

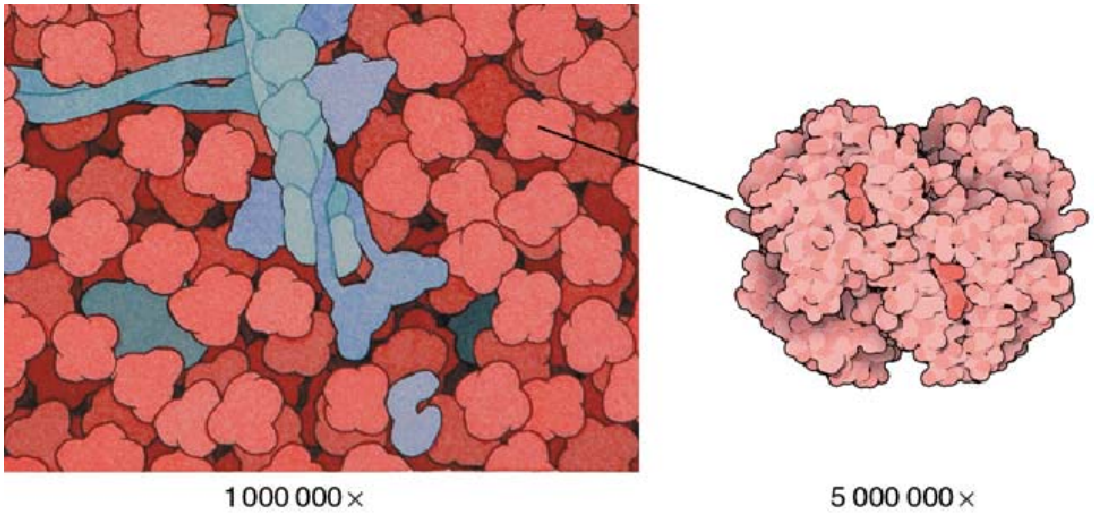

Figure 1.1: (Right) Computer generated hemoglobin, where each atom is distinguishable. Magnified by 1.000.000 (Left), hand drawn section of a red blood cell, magnified by 5.000.000 [Goo09].

threads to work on a task simultaneously. Klein et al.´s main bottleneck was the collision detection for molecules. To increase realism, overlapping molecules should be resolved. They use a brute force method implemented on the GPU, but for the mesoscale structure of a lipid bilayer issues arise as this method is too slow.

A lipid is a small longish molecule that builds together with other lipids the biggest structures inside a cell. When they come in contact with water they arrange themselves into a water impermeable layer that isolates the inside of the cell to its environment. A lipid bilayer consists of two layers of parallel orientated lipids where the tails of the two layers face each other. The lipids are oriented normal to the layer [Goo09], see Figure 1.2. Such structures often consist of millions of atoms. In comparison to proteins, which are dissolved in water and have lots of empty space around them, lipids are tightly packed and form a dense layer of molecules. To resolve collisions of so many closely spaced molecules, a highly specialized algorithm implemented on the GPU, to facilitate its high parallelism, is needed.

In this work we discuss such an algorithm, namely the Fast Fixed-Radius Nearest Neighbor (FFRNN) algorithm presented by Hoetzlein [Hoe14]. First we explore other methods in collision detection generally. Then we look into methods specialized for mesoscale visualization. We explain the FFRNN algorithm, describe how and why it works and how we can apply it to our molecule simulation. At the end we discuss and evaluate our approach and explore further improvements.

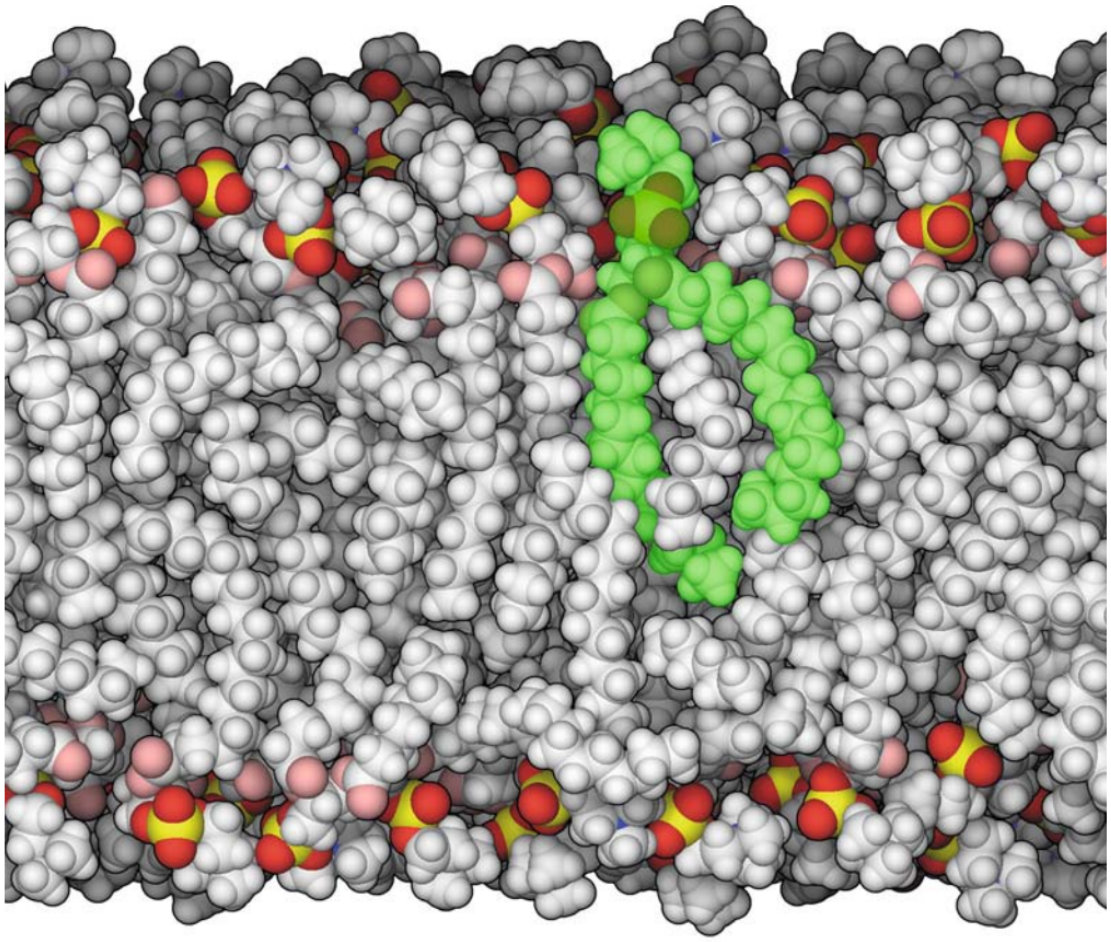

Figure 1.2: A cutout of a lipid bilayer. A single lipid molecule is highlighted in green. The Lipids are tightly packed and form a dynamic lipid bilayer [Goo09].

CHAPTER 2

Related Work

Collision detection is a large field of research in computer science. Many different algorithms exist, each with its own advantages and drawbacks. In this section, we first describe a selection of collision detection algorithms in Section 2.1 and then describe other mesoscale visualization tools for molecules and their approach for collision detection in Section 2.2.

2.1 Collision Detection

The simplest algorithm to find colliding bodies is to check all possible pairs, if they overlap. This brute force method is not feasible as its complexity is of $O(n^2)$ where n is the number of bodies. To reduce the number of checks, many collision detection algorithms consist of a phase where collision culling is performed to reduce the number of pairwise checks. This phase is called broad phase. The broad phase may have produced false positives, therefore the narrow phase starts, where the pairs of objects in proximity are checked more closely, if they collide.

A common algorithm for broad phase collision detection is the Sweep and Prune algorithm, firstly described by Cohen et al. [CLMP95]. For each body, an Axis Aligned Bounding Box (AABB) is defined. These AABB are projected on the x, y and z axis, resulting in intervals on these axes, see Figure 2.1. A collision is only possible, if and only if these intervals are overlapping on all three axes. Cohen et al. used three sorted lists, one for each axis, containing the endpoints of the projected AABBs, to quickly determine overlapping AABBs. These lists are sorted using *insertion sort*. Cohen et al. argued that for environments, where the coherence is preserved, this sorting is possible in $O(n)$ time, because *insertion sort* works well for previously sorted lists. In addition to sorting, the change of collision status for the intervals needs to be updated for each axis in each time frame. This is also possible in linear time due to coherence, which results in an overall complexity of $O(n + s)$ for this algorithm, where n is the number of bodies and s is the

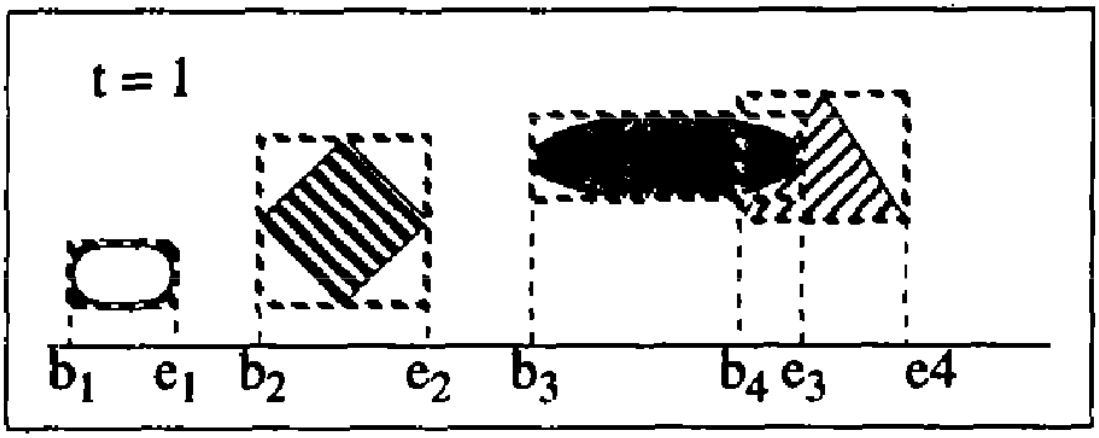

Figure 2.1: Axis Aligned Bounding Boxes and their intervals on a single axis. *b*: begin of bounding box, *e*: end of the bounding box. The begin of the bounding box four overlaps with the end of bounding box three. Such an overlap is registered in a list for this axis [CLMP95].

number of changes of collision status. This broad phase results in a list of candidate bodies, which may overlap.

More recently, Liu et al. presents a parallel version of the Sweep and Prune algorithm [LHLK10]. They perform Sweep and Prune simultaneously for many objects using the many threads of the GPU. They also do a Principle Component Analysis (PCA) to find the ideal axis to sweep.

Other collision detection algorithms rely on space partitioning, where the space is subdivided into smaller regions. Bodies in different regions can be pruned from the list of possibly colliding objects. The main structures used for spatial subdivision are *Uniform Grid, Quadtree, Kd-Tree* and *BSP Tree*.

Collision detection is often coupled with an appropriate response to the collision. This response is dependent on the general environment, for instance physical simulation, see Baraff et al. [Bar92], or molecular modeling, see Turk [Tur89]. For a physical simulation of colliding bodies, exact contact determination of those bodies is needed to compute an adequate response. This is where the narrow phase starts. Cohen et al.[CLMP95] use in their $I-Collide$ system the Lin-Canny collision detection algorithm, which tracks closest points between pairs of convex polytopes [Lin93]. To find those closest points, they use Voronoi regions that partition the space around polytopes.

2.2 Visualization Tools for Mesoscale Environments

The mesoscale represents an intermediate scale between molecular (nanoscale) and cellular biology. On the nanoscale, proteins, nucleic acids, lipids and polysaccharides are built

out of atoms and on the mesoscale, these molecules are assembled into more complex sub-cellular models. The mesoscale representation can help scientist to better understand the inner working of cells.

Mesoscale models are enormous, consisting of millions of atoms. Thus, they require specialized, highly-optimized methods to be rendered efficiently. Klein et al. [KAK+17] present the first approach of modeling the mesoscale within an interactive visual environment. According to Klein et al. [KAK+17], the workflow for modeling scenes consists of three basic steps:

1. *Scene organization*
2. *Recipe definition*
3. *Population of the model*

The first step defines compartments described as 3D meshes. The second step compiles the recipe that describes, which molecules make up which compartment. It also describes interactions with other molecules and constraints to their positions. The third step is the computational expensive part, where, according to the recipe, the scene is populated with molecules and fibrous structures.

To populate the membrane with lipids, an extension to the Wang tiling algorithm [FL05] is used to prevent visible repetition patterns, see Figure 2.2. First, four small patches, in form of a diamond, are extracted from the input texture. The input texture they use for biological membranes is a planar bilayer consisting of numerous, closely positioned lipids which do not overlap. The patches are combined to form a larger diamond patch and a rectangular patch is extracted from the center of the larger patch. This rectangular patch results in overlapping lipids which are detected and removed. The remaining holes are filled with non-colliding lipids. The collision detection uses two proxy geometries. The higher level geometry is the bounding sphere of the molecule and is used to detect potential collisions. The lower level geometry is made of a small number of spheres that approximates a molecule, see Figure 2.3. The spheres are calculated with a GPU-based K-means clustering algorithm [FRCC08]. The lower level geometry is used to verify collisions and to calculate reaction forces to resolve collisions. The calculation of the forces is loosely based on standard rigid body dynamics [Bar97]. For each pair of overlapping molecules, the overlapping spheres of the proxy geometries are used to calculate reaction forces. These forces are accumulated for each molecule as a linear and angular force. When all molecules are processed, the forces determine the new positions. Thereby it can happen that the molecules leave their assigned compartment. A springforce that pulls the molecule back to its position is applied to prevent that. The collision detection and force calculation is computed on the GPU.

A very complex part of a cell is the DNA. DNA has been traditionally modeled with random walks, which was slow because of the sequential nature of the algorithm [AJ11].

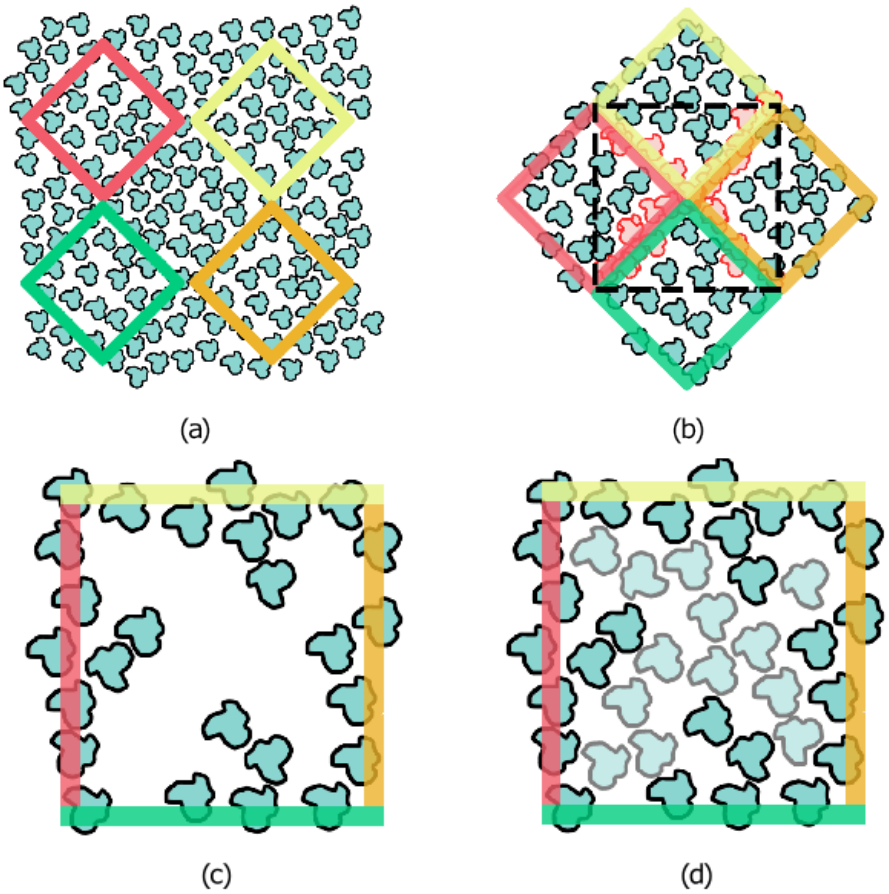

Figure 2.2: Wang tiling algorithm. (a) Extraction from input texture. (b) Combination to a larger texture and extraction of a rectangular texture. (c) Deletion of overlapping lipids. (d) Holes filling [KAK+17]

.

A random walk produces a set of points where each point is dependent on its predecessor. Klein et al. [KMA+19] propose a new approach to model such a structure in parallel on the GPU, which significantly speeds up the process. They first used a random walk with a large step size, to generate a coarse-grained backbone of the fibrous structure. With a parallel implementation of the midpoint displacement algorithm [Man82] [FFC82] they generate a more fine grained structure. The midpoint displacement method takes two spheres and produces a midpoint, which is displaced perpendicular to the line segment connecting the two initial spheres by a random amount. This process is repeated until a desired level of detail is reached. In each iteration the displacement magnitude is reduced. To not slow down the process, overlaps between spheres are disregarded. Therefore, a collision detection and resolving algorithm is applied to solve overlapping spheres after the structure is generated. Klein et al. use a fixed nearest neighbor search as described by Hoetzlein [Hoe14] to find overlapping spheres. This is the same algorithm as used in this

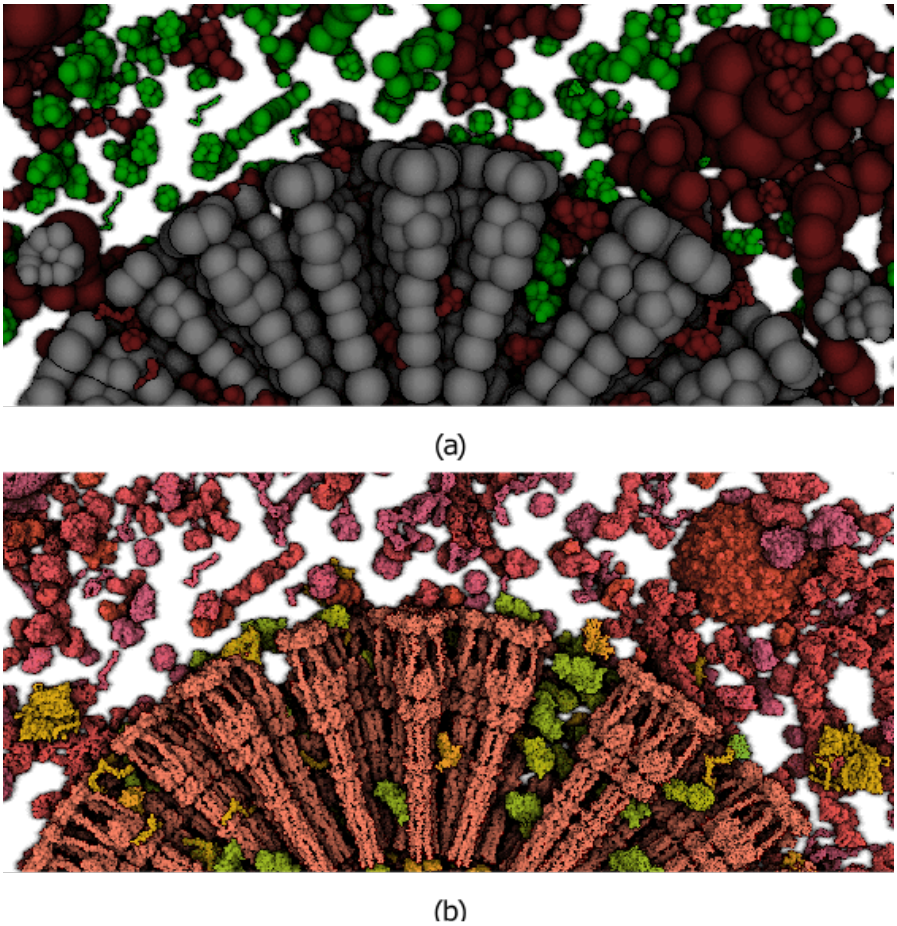

Figure 2.3: (a) Proxy geometry generated by the K-means clustering algorithm, used for collision detection and resolving. 17 Spheres were used to approximate the molecules. Grey are static molecules, red are colliding molecules and green are not colliding molecules. (b) Corresponding ingredients in atomic resolution [KAK+17].

work and is further described in Section 3. When overlapping pairs of spheres are found, a combination of a *repulsion* and a *recover* force is applied to resolve the collision. The *repulsion* force pushes the spheres apart while the *recover* force pulls them back in such a way that it adheres structural constraints. DNA is usually enclosed inside a membrane. During the collision resolving step, spheres can leave the enclosure. To prevent this, another force is applied that pushes the spheres back inside their compartment. This process is repeated until all collisions and constrains are resolved or a stop criterion is reached. To model the final DNA, several processing steps are applied on the spheres. Only then each sphere is replaced with a more detailed DNA building block.

Nguyen et al. [NSK+21] present a new technique to construct scientifically accurate mesoscale models. They use a rule-based modeling strategy that can utilize evidence

collected from microscopy data. Rules can also be specified by the modeler using his domain knowledge. The creation of a model begins by processing the rules sequentially. While the system populates the scene with molecules, a collision handling algorithm prohibits overlaps with other molecules. Nguyen et al. used an octree with four levels of subdivision to speed up the process. For every element a bounding sphere is assigned, that can be scaled by the user to better approximate the object. When a new candidate element is created, all leaf nodes of the octree, that intersect with the bounding sphere of the element, are fetched. If there are no collisions with the fetched elements, the candidate element is created. Otherwise the element is not created. This process can result in rules that never finish. Therefore, a counter is implemented that increments for each consecutive detected collision. If this counter reaches a threshold, the rule terminates.

A different approach is used by Gardner et al. [GAF+21] [GAB+18] to visualize mesoscale environments. Instead of 3D models they use 2D sprites that can be interactively added. To simulate a 3D world they use two background layers that are also populated with molecules, but in a darker shade. For collision detection they use Box2D, a well known 2D physics engine for games [Cat21]. Figure 2.4 shows CellPAINT and its user interface.

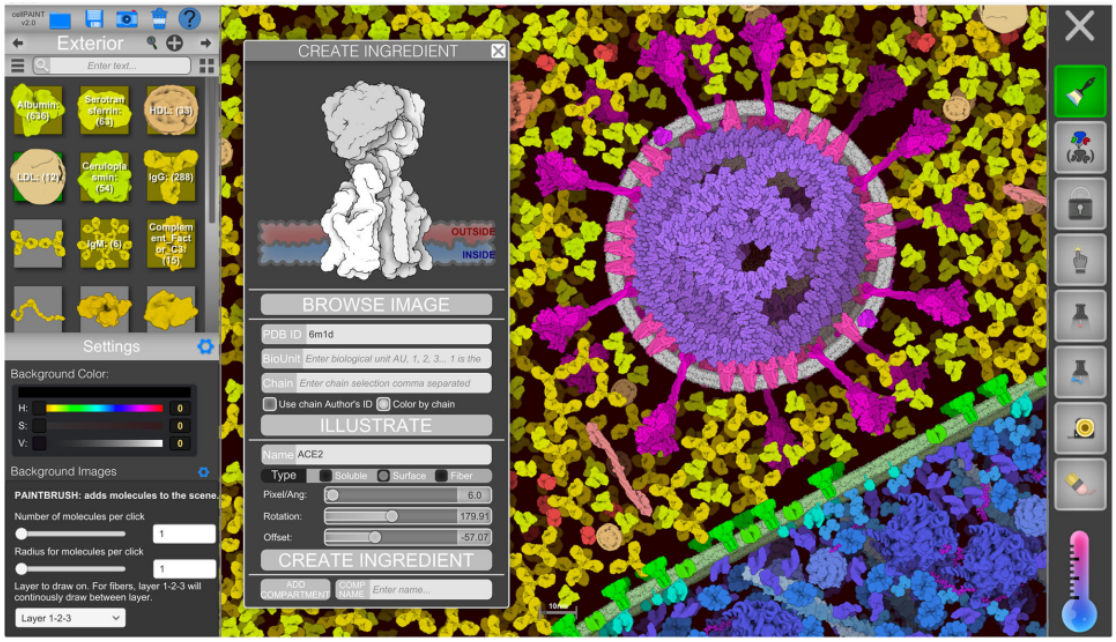

Figure 2.4: CellPAINT and its user interface. In this scene a simple eukaryotic cell, blood plasma and a coronavirus can be seen. The user creates a new ingredient based on atome coordinates from a Protein Data Bank [GAF+21].

CHAPTER 3

Methods

The main subject of this thesis is the real time collision detection and resolving of rigid bodies in the context of molecules. Collision detection of millions of bodies was previously solved with the FFRNN algorithm by Hoetzlein [Hoe14] on the GPU. The Fixed-Radius Nearest Neighbors problem is well known in computer science and has long been studied [Ben75]. For this bachelor thesis we consider the problem only in 3D Euclidean space. Given a set of Points P in 3D and a radius r, the Fixed-Radius Nearest Neighbors problem is defined as finding a set of all neighbors $N \subseteq P$ of a given query point $q \in P$ which are within the distance r. In this chapter we explain the FFRNN algorithm in detail and then we show how it is applied on a collision detection for molecules.

3.1 Fast Fixed-Radius Nearest Neighbors

Hoetzlein builds a uniform grid to partition the space instead of using brute force to search all possible pairs. Then the particles are placed into bins (cells) according to their spatial position. As the bin size is chosen in a way that it is at least as big or bigger than the radius, only neighboring bins need to be searched. This reduces the complexity to $O(k * N)$ where k is the number of particles in neighboring cells and N is the number of all particles. To find all neighbors of a particle, we query all particles that are in the same bin and all particles that are in neighboring bins. Such queries are slow on the GPU because these particles are laid out in memory randomly and would result in scattered reads, see Figure 3.1. To have ideal and fast coherent reads it is necessary to sort the particles by bins. Hoetzlein uses *Counting Sort* as sorting algorithm.

Counting Sort first creates a helper array by counting how many particles are in each bin and then computes a *Prefix Sum* of this array. A *Prefix Sum* of an array A is also an array S where each entry $S(x)$ is the sum of the entry $A(x)$ and all previous entries of $A(< x)$, see Figure 3.2. This counting of particles by bin and the calculation of the prefix sum can be done very efficiently in parallel on the GPU, see Section 4. Now sorting is

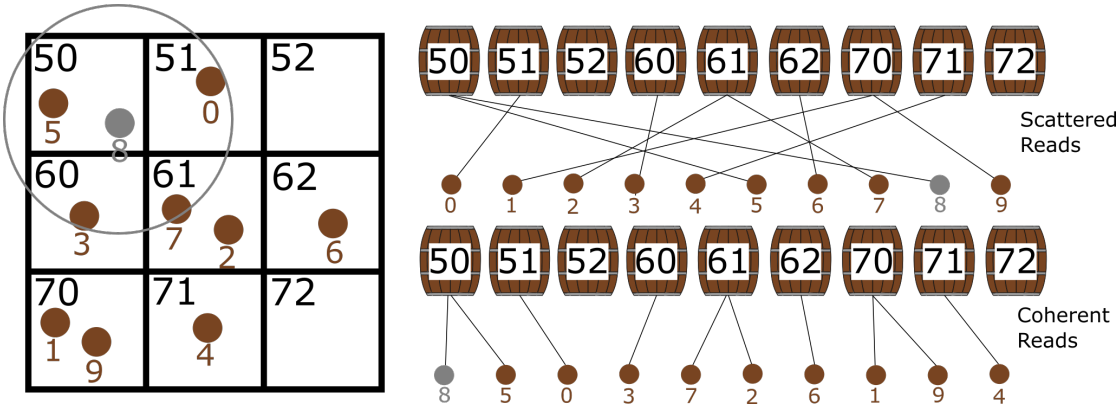

Figure 3.1: This Figure shows a grid and the corresponding binning process. First without sorting the points resulting in slow scattered reads, then with sorting by bins resulting in fast coherent reads. The search radius should not be greater then the cell size, otherwise more cells need to be searched.

only a matter of binning the particles into the correct bin. With the *Prefix Sum* array it is known where to put the particles in memory. After a particle is binned, the *Prefix Sum* array is updated by decrementing the corresponding entry. Note that the order of particles inside a bin is irrelevant. This binning process can also be implemented in parallel on the GPU, see Section 4. Now, to get all neighbors of a query point q of P the points of the bin containing q and all points of the neighboring bins are queried, then particles that are not within the radius are discarded.

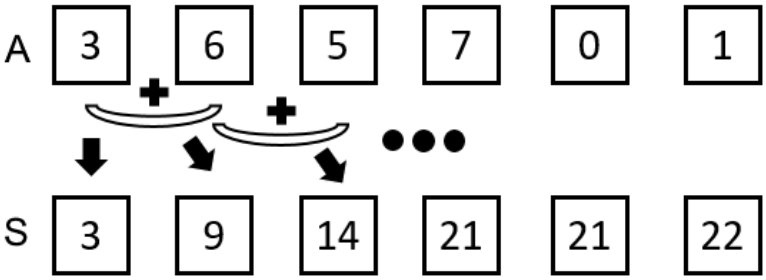

Figure 3.2: Prefix Sum of an array with random values. A is the initial array, S is the array with the Prefix Sum result.

3.2 Fast Fixed-Radius Nearest Neighbors on Molecules

The FFRNN can be used in many different applications. Hoetzlein [Hoe14] used it for a *Smoothed Particle Hydrodynamics* simulation. We show how FFRNN is used to detect and solve collisions of molecules.

In this application we implement a *Rigid Body Simulation*. That means bodies are not

deformable. The aim is to detect overlapping bodies. Therefore the radius, in which the search happens, should be the radius of the bounding sphere of the bodies. The FFRNN algorithm works, as its name suggests, only on a fixed radius. Nevertheless, molecules come in many different shapes and size. Defining a single large bounding sphere for all the different molecules would be impractical, because smaller molecules would be overestimated, for concave shaped molecules a sphere is not descriptive enough and large molecules would be underestimated, see Figure 3.3. The solution to this is to go down on the atomistic level. In the marion framework, which is used for this thesis, molecules are made of atoms, which are represented as spheres of the same size. Defining the search radius as the radius of the atoms, gives us an exact collision detection that detects overlapping atoms. However, atoms belonging to the same molecule do always overlap and are allowed to do so. These "collisions" must be excluded from the collision detection, see Figure 3.4.

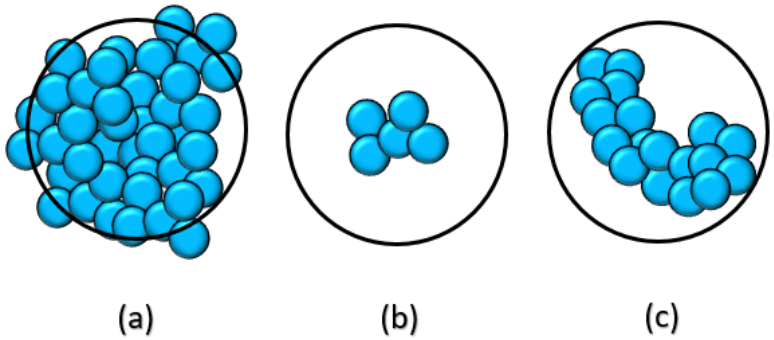

Figure 3.3: The problem of same sized bounding spheres for different molecules. (a) The bounding sphere underestimates the molecule, (b) the bounding sphere overestimates the molecule, (c) the bounding sphere is not descriptive for concavely shaped molecules.

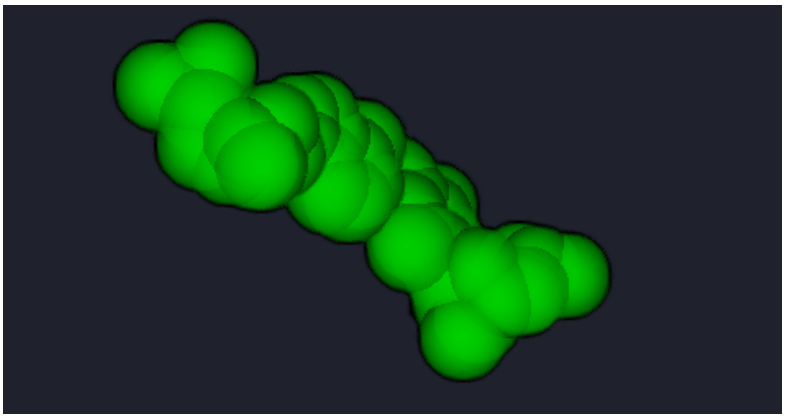

Figure 3.4: A single molecule. The overlapping of atoms within a molecule is allowed. These overlaps are disregarded by the algorithm.

In the same step, when we do the neighbor search, we also calculate the forces that push

the molecules apart. For each colliding pair of atoms we calculate a linear force, that moves the molecules apart and an angular force that rotates the molecules apart. These calculations are loosely based on rigid body dynamics as described by Baraff [Bar92] and are described in more detail in Section 4. The calculated forces are accumulated for each molecule. After the forces of all colliding pairs are calculated, the molecules are moved and rotated according to these forces.

As some molecules have a specific place inside a scene, e.g. lipids are arranged along the hull on a sphere pointing into the center of the sphere, they are not allowed to rotate or move across the scene unregulated. Therefore, we calculate a spring force and a spring torque for each molecule, that depends on its position and orientation. Similar to the collision resolving process we calculate an offset and a rotation out of these spring forces and move and rotate the molecules accordingly. In this way the molecules move back and forth over several iterations until a resting position is found or a stop criterion is reached.

The scene for testing the application arranges molecules along the hull of a sphere into two layers, pointing towards the center, see Figure 3.5. The editor is described in Figure 3.6. The number of molecules and the radius of the sphere can be changed independently and interactively using an editor menu. As the surface of a sphere is given by its radius, the number of molecules that fit on the surface is limited. If the number of molecules exceeds the available space, the algorithm will never find a resting position for each molecule. The user is responsible to find a suitable combination of number of molecules and radius of the sphere by trial and error. The user does not get a warning to indicate impossible parameter settings.

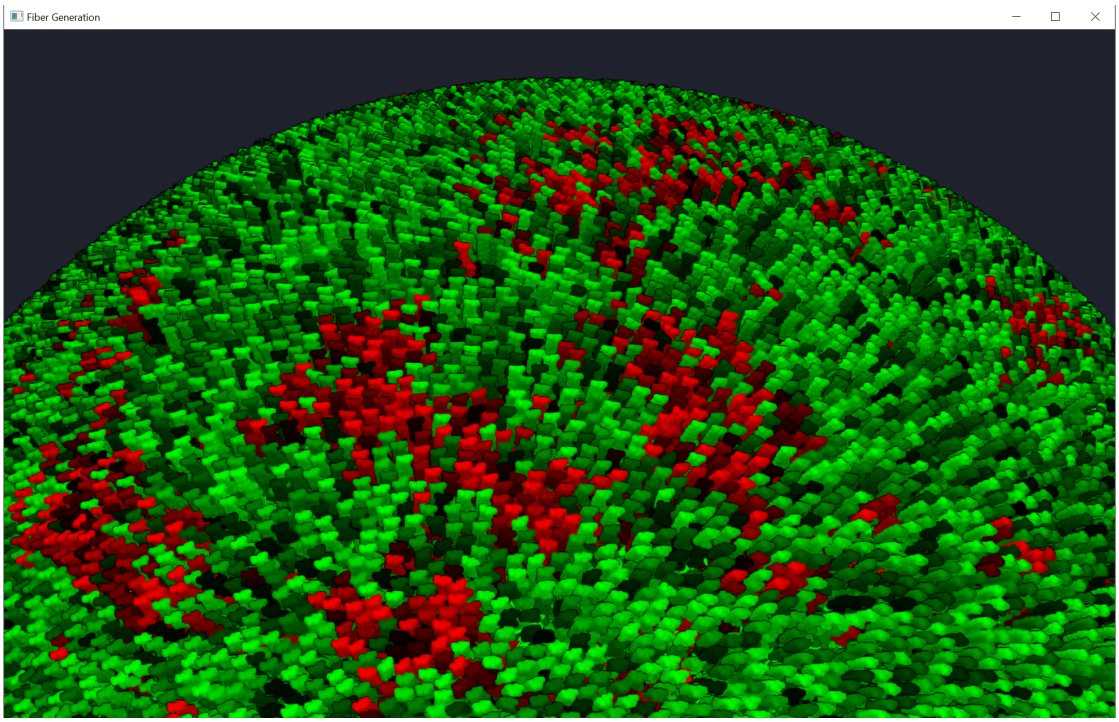

Figure 3.5: Screenshoot of the application. On the main screen the lipid bilayer model can be seen, consisting of 2.000.000 atoms. A green molecule indicates a non colliding state, a red molecule indicates a colliding state. The different shades of color have no meaning.

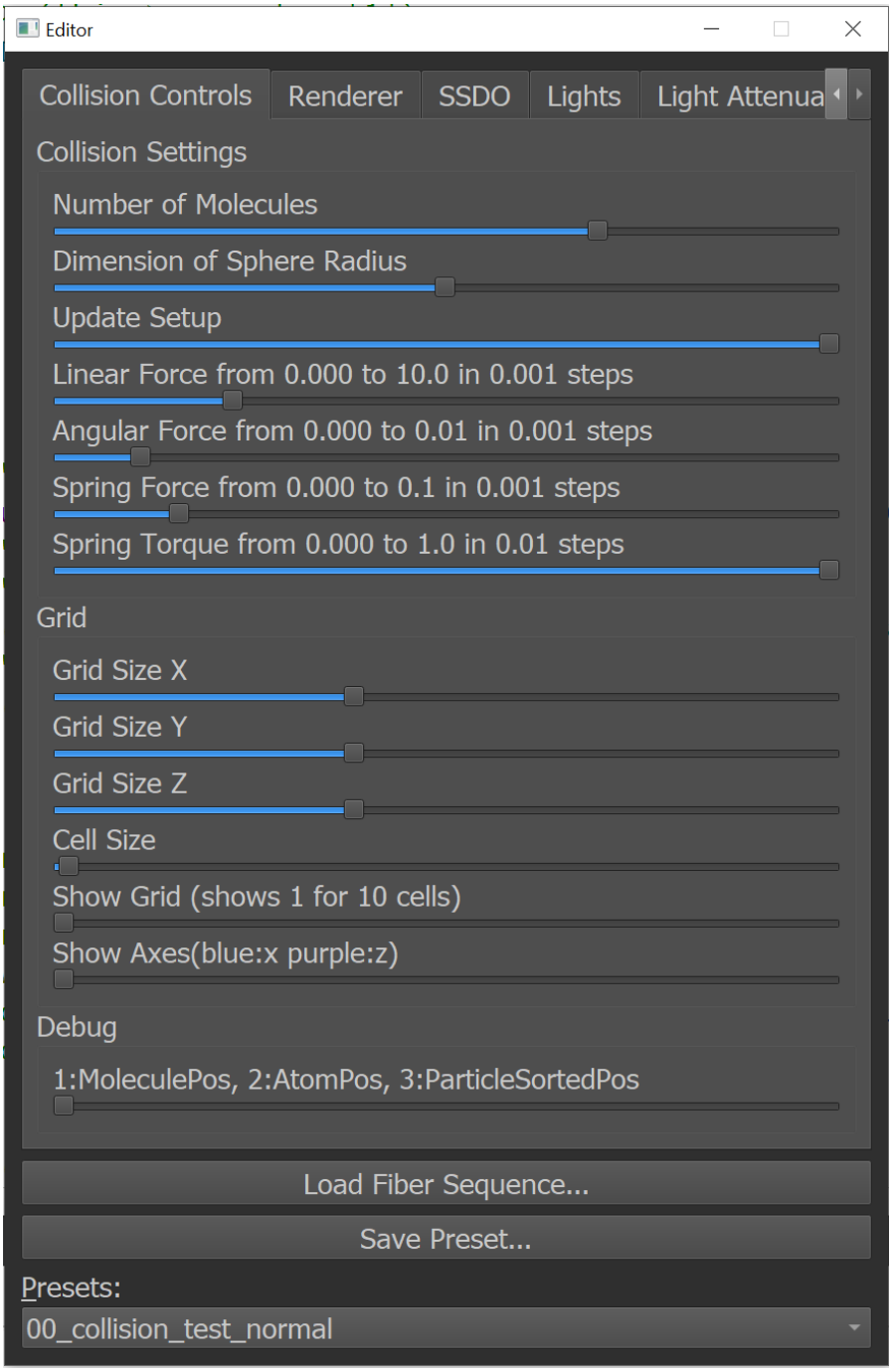

Figure 3.6: Screenshoot of the application´s editor. Various parameters can be adjusted in real time e.g. the number of molecules, the radius of the lipid bilayer sphere, integration step size for each type of force, grid size etc. It is also possible to save parameter settings as presets to load the settings in another session.

CHAPTER 4

Implementation Details

In this chapter we describe the implementation of the application in more detail. First, we describe, how molecules are generated and how to get the position of their atoms. Then we explain, how the grid is created and populated with atoms. Finally, we show how we calculate forces and torques and how they result in a displacement of molecules. Figure 4.1 shows an overview of the algorithm pipeline.

4.1 Molecule Placement

As mentioned in Chapter 3, molecules are made of atoms. Scenes can contain millions of molecules. Therefore GPU instancing is used to accelerate the population process. GPU instancing means that a single drawcall is used to render multiple similar objects, which reduces the overhead between CPU and GPU communication. As molecules are only a collection of differently arranged atoms, it is possible to define each molecule type once and send this information to the GPU. In addition, the molecules´ positions and orientations in global space are also sent to the GPU, so the GPU knows where to render the molecules. To place the molecules uniformly on a sphere, a sampling algorithm is used. Sampling a sphere is not trivial, as simple sampling can result in points clustered around the poles of the sphere, see Figure 4.2. For example, a simple sampling method is using uniformly distributed polar coordinates and then transforming them to Cartesian coordinates as seen in Algorithm 4.1. This method does not produce uniform samples on a sphere.

Distributing the molecules uniformly around the sphere is important, because the molecules should be as tightly packed as possible. Having clusters at the poles and a sparse distribution around the equator decreases the maximal number of molecules. A better way to sample a sphere is with the inversion method and polar coordinates. This method warps the polar coordinates, using the inverse of the cumulative distribution function (cdf) of ϕ (the polar angle) to produce a uniform distribution after transforming

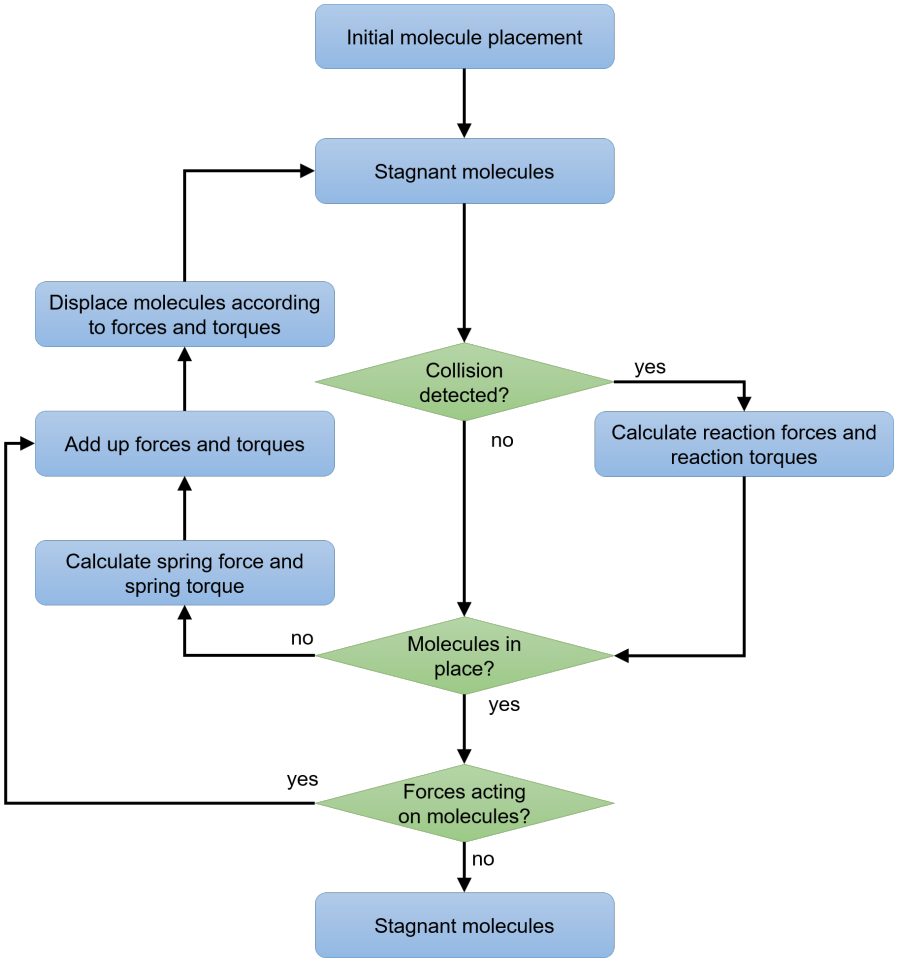

Figure 4.1: Overview of algorithm pipeline. First the molecules are placed inside the scene. If a collision is detected, reaction forces/torques are calculated per atom. If molecules are not in place, spring forces/torques are calculated per molecule. These forces are summed up so they can act per molecule. According to the forces the molecules are displaced. If no molecule is colliding and all molecules are placed correctly, the algorithm found a resting position.

to Cartesian coordinates, see Algorithm 4.2. The algorithms 4.1 and 4.2 give coordinates for the unit sphere. To create a sphere of any size, the coordinates must be multiplied by the desired radius.

The positions of all atoms of a molecule are easily calculated by querying the molecule type, rotating its atom positions by the molecule's orientation and adding the molecule's position in global space.

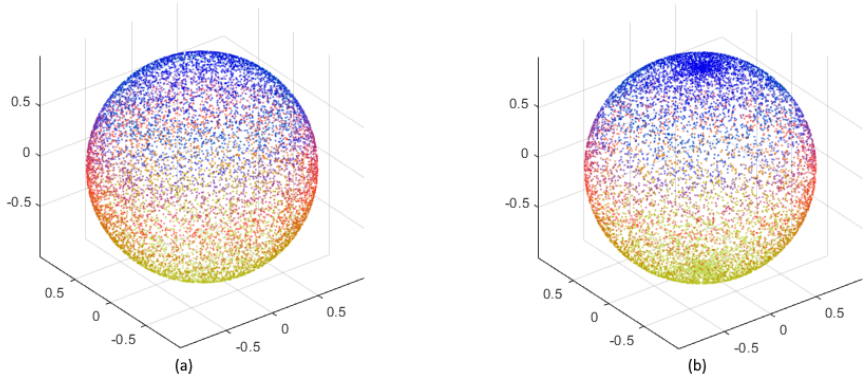

Figure 4.2: Sampling spheres. (a) A uniformly sampled sphere with the inversion method described in Algorithm 4.2. (b) A sphere sampled with the simple method described in Algorithm 4.1 with clusters around the poles.

Algorithm 4.1: Simple sampling resulting in clustered points around poles

Input: Two random floats $r_1\, r_2$ between [0, 1)
Output: A position Vector $\vec{p} \in \mathbb{R}^3$ in Cartesian coordinates
`// uniformly distributed polar coordinates`
1 $\theta = 2 * \pi * r_1$;
2 $\phi = \pi * r_2$;
3 $p_x = sin(\phi) * cos(\theta)$;
4 $p_y = sin(\phi) * sin(\theta)$;
5 $p_z = cos(\phi)$;
6 **return** $\vec{p}$;

4.2 Grid Creation

The uniform grid has to be big enough that every molecule is inside, otherwise the collision detection will not work properly as the molecules outside the grid are disregarded. The grid is described by the cell size as a float and a 3D integer vector that defines how many cells there are for each axis. The 3D grid can be flattened to a 1D array, where each entry represents a bin (cell) that counts the number of atoms. The size of this array can be calculated by the extent of the grid. The array size has to be the extent in x direction multiplied by the extent in y direction multiplied by the extent in z direction. Given a point in 3D it is possible to find its position in the array by first calculating its index on the x, y and z axis and then mapping these positions to 1D, see Algorithm 4.3. Now it is possible to populate the grid and increase the counter of a cell for each atom. This can be done in parallel on the GPU. To prohibit concurrent access on a shared memory *atomic addition* is used to update the cell counter. As the process is executed in parallel by multiple threads, without *atomic operations* it could happen that two threads read the same value simultaneously, increment it and write it back to memory, resulting in

Algorithm 4.2: Inversion method sampling resulting in uniformly sampled points

Input: Two random floats $r_1\, r_2$ between [0, 1)
Output: A position Vector $\vec{p} \in \mathbb{R}^3$ in Cartesian coordinates
`// warped polar coordinates`
1 $\theta = 2 * \pi * r_1$;
2 $\phi = acos(2 * r_2 - 1)$;
3 $p_x = sin(\phi) * cos(\theta)$;
4 $p_y = sin(\phi) * sin(\theta)$;
5 $p_z = cos(\phi)$;
6 **return** $\vec{p}$;

one lost operation and a wrong result. This is also known as the lost update problem. *Atomic operations* lock a value so no other thread can access it, performs its operation and writes the result back into memory and only then the value is freed again.

Algorithm 4.3: Calculate grid index of flattened 3D grid

Input: Atom position in 3D $\vec{p}$
Output: Atom's cell index in flattened array *index*
1 $x_{gridMax} = cellsize$;
2 $y_{gridMax} = cellsize$;
3 $z_{gridMax} = cellsize$;
`// Calculate index on each axis`
4 $x_{index} = \lfloor (x_{gridMax} + p_x)/cellsize \rfloor$;
5 $y_{index} = \lfloor (y_{gridMax} + p_y)/cellsize \rfloor$;
6 $z_{index} = \lfloor (z_{gridMax} + p_z)/cellsize \rfloor$;
7 $index = x_{index} + y_{index} * gridSize_x + z_{index} * gridSize_x * gridSize_y$;
`// Map to 1D`
8 **return** *index;*

The cell size is an important parameter for this algorithm. Choosing the cell size too big leads to too many atoms inside a single cell, which means the neighbor search takes longer. On the other hand, if choosing the cell size too small the search radius would span over multiple cells resulting in more cells needed to be searched. The grid dimension and cell size can be changed interactively, which makes it a lot easier to find optimal values. In this application a cell size of the size of a single atom results in the best performance.

After the grid is populated, the atoms are sorted by bins, with the help of a prefix sum array as mention in Section 3.1. There are many algorithms that efficiently compute a prefix sum on the GPU. Explaining such algorithms is rather complex and is not the focus of this work. The reader is referred to the book GPU Gems 3 [HSO07].

4.3 Calculating Forces

When the algorithm detects a colliding pair of atoms we compute a *linear force* and a *torque*. These forces are written inside *Shader Storage Buffers*. All the calculations happen on the GPU. We refer to the atom that is currently processed as left hand side atom (LHSA) and the atom, with which it collides, as right hand side atom (RHSA), see Algorithm 4.4. The direction of the force is the normalized difference between the position vector of LHSA and RHSA. As both atoms are spheres this direction corresponds exactly to the contact point of both atoms. To get the strength of the force we calculate the reciprocal of the sum of the squared distance between LHSA and RHSA and the squared local position of LHSA.

$$strenght = \frac{1}{squaredDistance + squaredLocalPositionLHSA} \tag{4.1}$$

That means the strength decreases when LHSA and RHSA slightly overlap, because of the reciprocal of the squared distance and it also decreases when the LHSA is farther away from the center of the molecule, because of the reciprocal of the squared relative position, see Figure 4.3. On the other hand the strength increases when LHSA and RHSA overlap more and when LHSA is near the center of the molecule. We multiply the normalized direction with this calculated force and save it for the LHSA into the force buffer. This calculation is a mere heuristic to approximate real world physics.

The reason we want to decrease the strength with the distance to the molecule's center is, that we also want to rotate the molecule. Sometimes it is much more efficient to rotate molecules out of a collision state instead of pushing them away. It is also more physically plausible to introduce rotational forces. Therefore, we also calculate torques for each atom. A torque can be seen as the rotational equivalent of a linear force. Intuitively a torque is the resulting rotation of a force acting on a lever, it is represented as an unnormalized 3D vector. It is computed by the cross product of the position vector and the force vector. In this application the position vector is the relative position of the LHSA as this is the vector that points from the origin of the molecule to the atom where the force is applied. The force vector is the normalized difference between the position vector of LHSA and RHSA (direction from RHSA to LHSA). The cross product gives us a vector that is perpendicular to those two vectors. The torque increases with the distance of LHSA to the molecule´s center, according to the principle of leverage: the greater the lever the greater the torque. The normalized force only influences the torque's axis as the force´s magnitude is always one. At the end we save the torque for this LHSA into the torque buffer.

After we have calculated the linear forces and torques for each atom of each molecule we can proceed. As mentioned in Chapter 3 we want to hold the molecules in a specific place and orientation. They are allowed to move and rotate, but not too far. We also want to define how far they are allowed to move and rotate. For this we calculate a *spring force* and a *spring torque* for each molecule that pulls the molecule back to its place, similar to a spring, see Algorithm 4.5. As we want to model a lipid bilayer in the form of a sphere

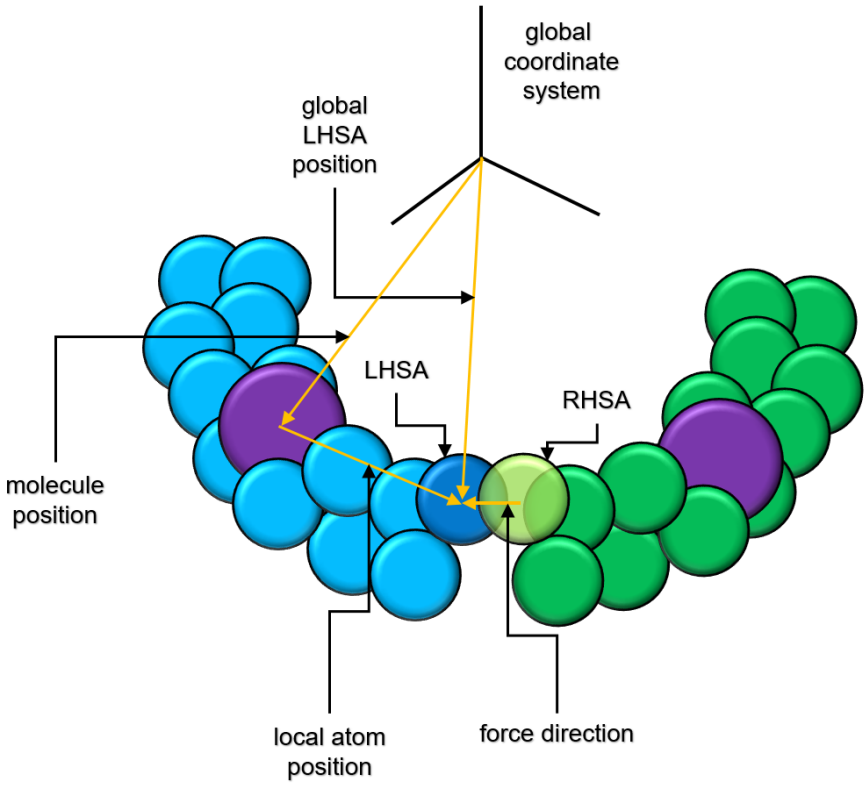

Figure 4.3: Variables visualized. We see two colliding molecules. The big purple sphere represents the molecules position in global space. The dark blue sphere represents the atom that is currently processed (LHSA) and the light green sphere represents the atom with which it collides (RHSA).

as described in Chapter 1, we know the radius of the sphere and therefore the required distance of the molecule to the center. If a molecule is outside the sphere, we calculate a vector that points from the molecule position to the center of the sphere. If a molecule is inside the sphere, we calculate a vector that points from the center of the sphere to the molecule. This vector is then saved into the spring force buffer.

Calculating the spring torque is not as straight forward as calculating linear forces. To form the lipid bilayer, we want the molecule's up vector to point to the center of the sphere. The sphere's center lies in the origin of the coordinate system. In this application the up vector of a molecule is defined as $\vec{up} = (0, 1, 0)$. We need a rotation that rotates the *current orientation* to the *target orientation*. The *current orientation* is the up vector rotated by the molecules rotation. The *target orientation* is the direction vector that points from the center of the sphere to the molecule, which is equivalent to the normalized position vector of the molecule. To formulate it like a lever/force-relationship we can interpret the current orientation as a position vector and the target orientation as a force

Algorithm 4.4: Reaction Force Calculation

Input: List of atoms, Grid
Output: Force and torque that act on an atom

```
for each atom LHSA in list in parallel do
    for each neighboring atom RHSA in Grid in sequence do
        direction = normalize(LHSA − RHSA);
        strenght = 1 / (squaredDistance+squaredLocalPositionLHSA);
        linearForce = direction ∗ strenght ∗ integrationStepForce;
        torque = cross(localPositionLHSA, direction) ∗ integrationStepTorque;
    end
end
return force and torque;
```

that acts onto the position vector pushing it in the target orientation. We calculate the rotation from *current orientation* to the *target orientation* as axis angle. The axis is calculated by taking the normalized cross product from the *current orientation* and the *target orientation*. The angle is calculated by taking the arcus cosinus from the dot product of the *current orientation* and the *target orientation*. If the angle of this rotation is smaller than a threshold, it means the molecule was already in the desired orientation and we add no torque to the spring torque buffer. Otherwise we take the axis of the rotation and multiply it with the angle, this gives us the torque that is needed to rotate the molecule to the desired orientation.

In addition we scale all the forces and torques, we just described beforehand, with a constant factor. This factor represents the integration step size and is for each type of force different and adjustable by the user. A bigger step size means the corresponding forces have a bigger influence on the displacement of the molecules in each frame. The step size greatly influences the collision resolving, as large values would lead to a farther displacement, and eventually push molecules into one another. On the other hand, a small value reduces the displacement and the resolving of collisions takes additional frames. Generally it is recommended that the integration step size should be small for a scene with a very dense distribution of molecules e.g. lipid bilayer, and large for scene with a sparse distribution of molecules e.g. cytoplasm. Making this factor adjustable gives more control to the user and an adequate preset is also available.

As we have calculated the *linear forces* and *torques* for each atom, we need to accumulate them for each molecule. We also add the *spring forces* and *spring torques* to the same buffers. Because we do the operations also in parallel on the GPU we need to make sure the additions do not operate concurrently on the same value. Therefore, we also use atomic operations to add up the forces and torques of each molecule. Adding up torques first seems odd as we add up axes, but because all torques pivot around the molecules center, which is also its center of gravity, we can simply add them up [Bar97]. Unfortunately,

Algorithm 4.5: springForce Calculation

Input: List of Molecules
Output: Spring Force and Spring Torque that act on a molecule

for *each Molecule* $\vec{position}$ *in list in parallel* **do**
    // Calculate spring force
    $epsilon1$ = leeway for the molecule´s position;
    $epsilon2$ = leeway for the molecule´s orientation;
    $radius$ = radius of the sphere;
    $distance$ = distance from center of the sphere to the molecule;
    if $distance > radius + epsilon1$ **then**
        $\vec{springForce} = -\vec{position} * IntegretionstepSpringForce$;
    else if $distance < radius - epsilon1$ **then**
        $\vec{springForce} = \vec{position} * IntegretionstepSpringForce$;
    // Calculate spring torque
    $currentOrientation = \vec{up}$ rotated by molecule´s rotation;
    $targetOrientation = normalize(\vec{position})$;
    $\vec{torqueAxis} = normalize(cross(targetOrientation, currentOrientation))$;
    $angle = acos(dot(currentOrientation, targetOrientation))$;
    if $(180/\pi) * angle > epsilon2$ **then**
        $\vec{springTorque} = torqueAxis * angle * IntegrationstepSpringTorque$;
end
return *springForce and springTorque;*

these forces and torques come in floats and at the time of writing this thesis only Nvidia cards support atomic operation on floats using the *GL_NV_shader_atomic_float* extension [CWB12]. We take the downside and go with the support of only Nvidia graphics cards users. When all forces and torques are added up, we can continue and apply the forces and torques on the molecules.

The linear force is just added to the position of the molecule. The torque must first be converted to a quaternion, because the current rotation of the molecule is also expressed as a quaternion. To rotate the molecule we need to multiply the quaternion converted from the torque with the quaternion representing the current rotation of the molecule. After all forces and torques are applied, one iteration is done.

CHAPTER 5

Evaluation

In this chapter we evaluate the algorithm's efficiency by comparing its execution time to a brute force implementation. The FFRNN algorithm has a complexity of $O(k * N)$ where N is the number of particles and k is the number of particles in neighboring cells. Domain specific costs need to be added, for instance the computation of forces, the accumulation of these forces for each molecule and the application of these forces on the molecules. To provide a ground of comparison, a brute force algorithm was implemented on the GPU that uses the same collision resolving calculations but to find colliding pairs it checks all other atoms in the scene. Such an algorithm has a complexity of $O(N^2)$. Figure 5.1 shows the difference in execution time on a semi-y-log graph, where the y axis is logarithmic and the x axis is linear.

In Table 5.1 the exact times the algorithm needs for one execution and the corresponding number of atoms are listed. For 100.000 atoms upwards the brute force algorithm can not be considered real time anymore, as the time to compute one execution takes already 280 ms. 280 ms corresponds to a frame rate of 3.5 Frames Per Second (FPS). It can be seen that the FFRNN algorithm is far more performant than the brute force algorithm.

Atoms	**FFRNN**	**Brute Force on GPU**
50 000	9 ms	61 ms
100 000	10 ms	280 ms
200 000	11 ms	977 ms
400 000	16 ms	4.530 ms
1 000 000	28 ms	25.392 ms
2 000 000	44 ms	102.604 ms
4 000 000	78 ms	599.236 ms

Table 5.1: The table shows the execution time of the FFRNN and brute force algorithm. At 100.000 atoms the brute force algorithm can not be considered real time anymore.

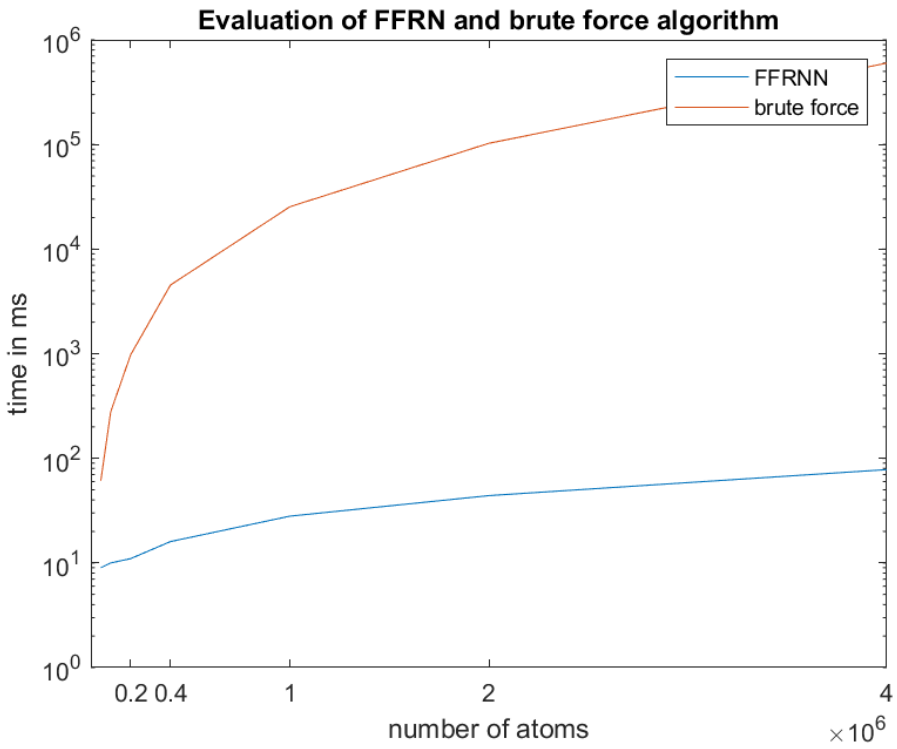

Figure 5.1: Comparison between the FFRNN and the brute force algorithm. It can be seen that the brute force algorithm increases quadratically and the FFRNN linearly. Tested on an Nvidia GeForce GTX 970.

To take a closer look, the execution time of each compute shader is measured, see Table 5.2. These times were measured with 3.000.000 atoms inside the scene on an Nvidia Geforce GTX 970. It can be seen that the prefix sum and the sorting of the atoms only take little time while most of the time is needed for the collision detection. Note that the collision detection includes also the computation of the *linear force* and *torque* as described in Section 4.3.

Steps	**time**	
Insert Atoms	1 ms	
Prefix Sum	8 ms	
Sort Particles	3 ms	
Sum	**12 ms**	Preprocessing Steps
Detect Collisions	40 ms	
Calculate Spring Force	0 ms	
Sum Forces	7 ms	
Apply Forces	1 ms	
Sum	**48 ms**	One iteration

Table 5.2: Execution times for each step of the FFRNN algorithm for 3.000.000 atoms on an Nvidia Geforce GTX 970.

CHAPTER 6

Discussion and Future Work

In this chapter we address diverse shortcomings of our work and how they can be solved in the future. First we look into the upper limit of atoms inside a scene and how we can increase that number. Then we investigate the algorithm for opportunities to improve its performance. One of the criteria of this application is that it needs to be in real time. With 4.000.000 atoms the algorithm needs 78 ms for a single iteration. This corresponds to a frame rate of around 13 FPS, which is arguably on the lower end of a real time application. But the collision detection is not the only process in the application that needs processing time. Rendering and shading of such a high number of bodies decreases the FPS even further. On the other hand, when all the collisions are resolved, the algorithm stops and frees a lot of processing time, that means FPS drops would occur only shortly at the start of the application. Nevertheless, managing an increasing number of atoms inside a scene is still a valuable endeavor.

6.1 Proxy Geometry with K-Means Clustering

As described in Chapter 2, Klein et al. use a low level proxy geometry constructed with a K-means algorithm that approximates the geometry of a molecule with fewer but bigger spheres, see Figure 2.3 [KAK+17] [FRCC08]. This proxy geometry overestimates the real geometry only a little and is not noticeable for the common user. It is still possible to render the scene on the atomic level, but for the collision detection we could use this proxy geometry. Klein et al. use only 17 spheres to approximate a single molecule. Especially for large molecules consisting of thousands of atoms this would mean a significant performance boost. The problem which arises with this approach is that the spheres of the proxy geometry differ in size for differently sized molecules. Yet the FFRNN algorithm only works on spheres of the same size and some modifications are needed to make different sized spheres possible. One problem is that the cell size needs to be adjusted as bigger spheres could extent over several cells and not just the

neighboring ones. Also the radius, in which the neighbor search happens, needs to account for differently sized spheres. These problems can surely be solved but are left for future work.

6.2 Slow Atomic Operations

In Table 5.2 row seven, we see that adding up forces takes seven ms. This is rather long and can be expected to increase, when the number of atoms per molecule increases, as more threads want to write concurrently into one position in memory. To prevent lost updates, we use atomic additions. Atomic additions prevent lost updates, but also slow down the computation the more threads want to access a value concurrently. In our case we accumulate the forces of each atom of a molecule and write it in one place. Hence, the number of threads that want to concurrently access the same value, is as big as the number of atoms in a molecule. Generally, molecules can have thousands of atoms meaning using atomic operations is not a feasible way for some applications. Another downside of using atomic operations is that it uses the *GL_NV_shader_atomic_float* extension [CWB12] that, at the time of writing this thesis, only Nvidia graphics cards support. Finding another approach to accumulate the forces that act on a molecule could increase performance and also provide support for a larger audience. It should be noted that atomic operations are not necessarily slow, we used them in the grid population process described in Section 4.2 too. But in the grid creation process, the number of concurrently accessed values is small, because we accumulate the number of atoms inside a cell and this number is also small.

CHAPTER 7

Conclusion

We discussed the FFRNN algorithm and built a collision detection and resolving algorithm on top of it that works for mesoscale environments. With our method we are able to compute and resolve overlaps of molecules on an atomic level with millions of atoms in a physically plausible way. We utilize the high parallelism of the GPU to detect colliding pairs of atoms and calculate forces that efficiently resolve collisions and also provides means to preserve molecules' position and orientation to an extend.

To evaluate our approach we implemented a simple brute force algorithm to provide a ground of comparison with other algorithms. Our method outperforms the brute force approach by several orders of magnitude. With the application that resulted of this work it is possible to construct test environments interactively. We are able to load different molecules and arrange them into a sphere similar to a lipid bilayer. To populate the outer hull of the sphere we use the inversion method to uniformly distribute the molecules. Thereby we reach a very dense distribution that provides a good starting point for our method.

The processing time of the algorithm increases linearly with increasing number of atoms. At some point the algorithm takes too long to process all atoms of a large scene. We discussed a way to improve the algorithm´s performance further by introducing proxy geometry that has all ready been implemented by Klein et al. [KAK+17].

Our method is fast and resolves collisions correctly. Further development can increase performance and may eliminate dependencies on Nvidia graphics cards. Overall we showed that our method is applicable in mesoscale scenarios and can be expected to perform very well on large scenes.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AJ11] Hellen Altendorf and Dominique Jeulin. Random-walk-based stochastic modeling of three-dimensional fiber systems. *Physical Review E*, 83(4):041804, 2011.

[Bar92] David Baraff. *Dynamic simulation of nonpenetrating rigid bodies*. PhD thesis, Cornell University, 1992.

[Bar97] David Baraff. An introduction to physically based modeling: rigid body simulation i—unconstrained rigid body dynamics. *SIGGRAPH course notes*, 82, 1997.

[Ben75] Jon L. Bentley. Survey of techniques for fixed radius near neighbor searching. Technical report, Stanford Linear Accelerator Center, Calif.(USA), 1975.

[Cat21] Erin Catto. Box2d. `https://box2d.org/`, 2021. Accessed: 10.01.2022.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav 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.

[CWB12] Cyril Crassin, Eric Werness, and Jeff Bolz. Nv-shader-atomic-float. `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_atomic_float.txt`, 2012. Accessed: 10.01.2022.

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[FL05] Chi-Wing Fu and Man-Kang Leung. Texture tiling on arbitrary topological surfaces using wang tiles. In *Rendering Techniques*, pages 99–104. Citeseer, 2005.

[FRCC08] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H. Campbell. A parallel implementation of k-means clustering on gpus. In *Parallel and Distributed Processing Techniques and Applications*, volume 13, pages 212–312, 2008.

[GAB+18] Adam Gardner, Ludovic Autin, Brett Barbaro, Arthur Olson, and David S. Goodsell. Cellpaint: Interactive illustration of dynamic mesoscale cellular environments. *IEEE Comput Graph Appl*, 38(6):51–66, 2018.

[GAF+21] Adam Gardner, Ludovic Autin, Daniel Fuentes, Martina Maritan, Benjamin A. Barad, Michaela Medina, Arthur J. Olson, Danielle A. Grotjahn, and David S. Goodsell. Cellpaint: Turnkey illustration of molecular cell biology. *Frontiers in Bioinformatics*, 1:7, 2021.

[Goo09] David S. Goodsell. *The machinery of life*. Springer Science & Business Media, 2009.

[Hoe14] Rama C. Hoetzlein. Fast fixed-radius nearest neighbors: interactive million-particle fluids. In *GPU Technology Conference*, volume 18, 2014.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda, gpu gems 3, 2007.

[JAAA+15] Graham T. Johnson, Ludovic Autin, Mostafa Al-Alusi, David S. Goodsell, Michel F. Sanner, and Arthur J. Olson. cellpack: a virtual mesoscope to model and visualize structural systems biology. *Nature methods*, 12(1):85–91, 2015.

[KAK+17] Tobias Klein, Ludovic Autin, Barbora Kozlíková, David S. Goodsell, Arthur Olson, Eduard Gröller, and Ivan Viola. Instant construction and visualization of crowded biological environments. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):862–872, 2017.

[KMA+19] Tobias Klein, Peter Mindek, Ludovic Autin, David S. Goodsell, Arthur J. Olson, Eduard Gröller, and Ivan Viola. Parallel generation and visualization of bacterial genome structures. In *Computer Graphics Forum*, volume 38, pages 57–68. Wiley Online Library, 2019.

[LHLK10] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. Real-time collision culling of a million bodies on graphics processing units. *ACM Transactions on Graphics (TOG)*, 29(6):1–8, 2010.

[Lin93] Ming Chieh Lin. *Efficient collision detection for animation and robotics*. PhD thesis, 1993.

[Man82] Benoit B. Mandelbrot. *The fractal geometry of nature*, volume 1. WH Freeman New York, 1982.

[NSK+21] Ngan Nguyen, Ondřej Strnad, Tobias Klein, Deng Luo, Ruwayda Alharbi, Peter Wonka, Martina Maritan, Peter Mindek, Ludovic Autin, David S. Goodsell, and Ivan Viola. Modeling in the time of covid-19: Statistical and rule-based mesoscale models. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):722–732, 2021.

[Tur89] Greg Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, 1989.