UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DELAUNAY TRIANGULATIONS FOR MOVING POINTS FIXED RADIUS NEAR NEIGHBORS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

HEINICH SAID PORRO SUFAN

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
JÉRÉMY BARBAY
CRISTÓBAL NAVARRO GUERRERO

Este trabajo ha sido parcialmente financiado por el Proyecto Fondecyt N°1181506

SANTIAGO DE CHILE
2021

Resumen

En simulaciones de fluidos basadas en el método smoothed particle hydrodynamics (SPH), la operación para encontrar los vecinos dado un cierto umbral fijo de cada partícula es la operación más costosa que debe realizarse en cada iteración de la simulación. A este problema lo llamamos Moving Points Fixed Radius Nearest Neighbors (MPFRNN). El mismo problema, cuando las partículas no se mueven, se llama Fixed Radius Nearest Neighbors (FRNN).

El principal aporte de este trabajo es la descripción y evaluación en escenarios de simulación sin preservar el volumen excluido de cada partícula (donde las partículas pueden atravesarse entre sí), de un algoritmo nuevo que resuelve el problema MPFRNN mediante triangulaciones de Delaunay. El análisis teórico sugiere que este enfoque es más rápido que las soluciones del estado del arte dadas ciertas condiciones. Este trabajo también explora las dificultades de paralelizar este algoritmo en la GPU.

En el algoritmo secuencial descrito, la triangulación de Delaunay inicial se construye a partir de un conjunto de n vértices utilizando el algoritmo Jump&Walk. Luego, se mantiene eliminando y luego moviendo y reinsertando todos los vértices en la triangulación. Esto se mejora en el caso de los vértices que no se mueven fuera de sus cavidades (que es el polígono formado por los vértices alrededor de un vértice), puesto que la triangulación se puede arreglar localmente mediante flips. Finalmente, el FRNN se calcula realizando una búsqueda en profundidad (DFS) en la triangulación, teniendo especial consideración en no realizar cálculos de distancia duplicados, utilizando sólo $O(n)$ memoria adicional.

Los datos experimentales muestran que el algoritmo secuencial propuesto es más lento que un algoritmo de referencia en la mayoría de los escenarios probados. Aún así, el algoritmo propuesto tiene un tiempo de actualización comparable, que corresponde al tiempo que lleva actualizar la triangulación de Delaunay de los puntos móviles. Esto se puede utilizar en trabajos futuros para desarrollar una mejor estructura de datos secuenciales para resolver el problema MPFRNN.

Abstract

In Smoothed Particle Hydrodynamics (SPH) based fluids simulations, the operation to find the neighbors given a certain fixed threshold of each particle is the most costly operation that has to be performed at each iteration of the simulation. We call this problem Moving Points Fixed Radius Nearest Neighbors (MPFRNN). On a static setting, where the particles do not move, this problem is called Fixed Radius Nearest Neighbors (FRNN).

The main contribution of this work is the description and evaluation in simulation scenarios without preserving the excluded volume of every particle (where particles can go through each other), of a novel algorithm that solves the MPFRNN problem using Delaunay triangulations. Theoretical analysis suggests that this approach is faster than state of the art solutions under certain conditions. This work also explores the difficulties in parallelizing this algorithm on the GPU.

In the described sequential algorithm, the initial Delaunay triangulation is built from a set of n vertices using the Jump&Walk algorithm. It is then maintained deleting and then moving and reinserting all the vertices in the triangulation. This is improved in the case of vertices that do not move outside its cavities (which is the polygon formed by the vertices around a vertex), where the triangulation can be locally fixed only using flips. Finally, the FRNN is calculated performing a Depth First Search (DFS) on the triangulation, taking special consideration of not performing duplicated distance computations, using within $O(n)$ additional memory.

Experimental data shows that the proposed sequential algorithm is outperformed by a baseline algorithm in most of the scenarios tested. Still, the proposed algorithm has a comparable update time, which corresponds to the time it takes to update the Delaunay triangulation of the moving points. This can be used in future work to develop a better sequential data structure to solve the MPFRNN problem.

A Paula y a su mamá.

Agradecimientos

Gracias a la profesora Nancy Hitschfeld por tomarse el tiempo de conversar conmigo de más que sólo la tesis. Por su confianza, su paciencia y guía. Al Profesor Benôit Crespin por sus consejos y ayuda. A Francisco Carter y Cristobal Navarro, por su ayuda con todo lo relacionado a programación en GPU, y por tomarse el tiempo de explicarme personalmente una y otra vez su trabajo. A Hang Si por tomarse el tiempo de escuchar y aportar ideas para este trabajo. Gracias también a los miembros de la comisión por tomarse el tiempo de leer y corregir esta tesis.

Gracias al proyecto Fondecyt N° 1181506 por permitirme desarrollar esta tesis sin tener que preocuparme por mis finanzas personales en tiempos tan complicados.

Mis abuelos, tíos, tías, primos, primas, hermanos, papá y mamá. Todos fueron parte en menor o mayor medida de mi paso por la universidad, y les estoy eternamente agradecido. En particular le estoy muy agradecido a la familia que tengo en Santiago, por haberme ayudado a acostumbrarme a la vida capitalina, y a mis hermanos por tomarse el tiempo de escuchar y entender los problemas que iba teniendo a medida que desarrollaba mi tesis.

Finalmente, muchas gracias a Paula y a su mamá por alojarme en su casa por tantos meses durante esta cuarentena. Sin su apoyo incondicional, tanto emocional como material, esta tesis no podría haberse realizado.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Research Questions . . . 2
- 1.3 Hypothesis . . . 2
- 1.4 Objectives . . . 2
  - 1.4.1 General Objectives . . . 2
  - 1.4.2 Specific Objectives . . . 2
- 1.5 Methodology . . . 2
- 1.6 Structure of this work . . . 3

2 Theoretical Background **4**
- 2.1 Computer geometry fundamentals . . . 4
  - 2.1.1 Common geometric tests . . . 4
  - 2.1.2 Precision and Robustness . . . 6
- 2.2 Triangle meshes . . . 7
  - 2.2.1 Triangulation of a set of points . . . 8
  - 2.2.2 Flipping edges . . . 8
  - 2.2.3 Data structures . . . 8
  - 2.2.4 Delaunay triangulations . . . 9
- 2.3 GPU Computing concepts . . . 10
  - 2.3.1 GPU architecture . . . 10
  - 2.3.2 GPU programming . . . 11

3 Related Work **14**
- 3.1 Cell Linked Lists . . . 14
- 3.2 Related work on Delaunay triangulations . . . 15
- 3.3 Precision problems in computer geometry . . . 16

4 Proposed Algorithm **18**
- 4.1 Sequential Kinetic Delaunay triangulation for small displacements . . . 19
  - 4.1.1 Data structure . . . 19
  - 4.1.2 Construction . . . 19
  - 4.1.3 Moving Points Delaunay triangulation . . . 22
  - 4.1.4 Moving Points Fixed radius neighbors . . . 23
- 4.2 Exploring Parallel Moving Points Delaunay triangulation . . . 24
  - 4.2.1 Keeping the moving points Delaunay triangulation . . . 24

4.2.2 FRNN in GPU using the Delaunay triangulation . . . 26

5 Results 28
5.1 Experimental setup . . . 28
5.1.1 Shaking points . . . 28
5.1.2 Fluid simulation . . . 29
5.2 Shaking points simulation results . . . 30
5.3 Falling column fluid simulation results . . . 33
5.4 Fluid simulation without gravity results . . . 38
5.5 Analysis . . . 42
5.5.1 Shaking Points . . . 42
5.5.2 Fluid simulation . . . 42

6 Conclusions 44
6.1 Conclusions . . . 44
6.2 Future Work . . . 45

Bibliography 47

Appendix A Code appendix 50
A.1 Cell Linked Lists code . . . 50
A.2 Fluid simulation code . . . 53

List of Tables

5.1 Shaker simulation variables used. 31
5.2 Variables set in different settings of the falling column fluid simulations. . . . 33
5.3 Different settings of the initial random position fluid simulation without gravity. 38

List of Figures

1.1 FRNN of a particle, and Delaunay triangulation of the point set 1

2.1 Area of the cross product. 5
2.2 Segments intersection. 6
2.3 Summary of the geometric tests. 7
2.4 Graphic representation of the SplitTriangle function. 8
2.5 Comparison of physical space utilization between CPU and GPU architectures. 11
2.6 Automatic scalability in CUDA model. 11
2.7 CUDA Heterogeneous programming. 12
2.8 CUDA memory hierarchy. 13

3.1 Graphic representation of Cell linked lists of a 2D point set. 15

4.1 Information stored in the used data structure. 19
4.2 Flip between two triangles 22
4.3 Triangle fix and Delaunay condition checking. 25
4.4 Fixing a triangulation when a particle mover further away, performing 2 flips. 26
4.5 Special case fix. 26
4.6 Special case of the algorithm described by Carter et al. 27

5.1 Double density relaxation. 29
5.2 Initial configurations of fluid simulations. 30
5.3 Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 5,000 particles and different velocities. 31
5.4 Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 10,000 particles and different velocities. 32
5.5 Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 100,000 particles and different velocities. 33
5.6 Falling column case 0 graphs. 34
5.7 Falling column case 1 graphs. 34
5.8 Falling column case 2 graphs. 35
5.9 Falling column case 3 graphs. 35
5.10 Falling column case 4 graphs. 35
5.11 Falling column case 5 graphs. 36
5.12 Falling column case 6 graphs. 36
5.13 Falling column case 7 graphs. 36
5.14 Falling column case 8 graphs. 37

5.15 Falling column case 9 graphs. 37
5.16 Falling column case 10 graphs. 37
5.17 Fluid simulation without gravity case 0 graphs. 38
5.18 Fluid simulation without gravity case 1 graphs. 39
5.19 Fluid simulation without gravity case 2 graphs. 39
5.20 Fluid simulation without gravity case 3 graphs. 39
5.21 Fluid simulation without gravity case 4 graphs. 40
5.22 Fluid simulation without gravity case 5 graphs. 40
5.23 Fluid simulation without gravity case 6 graphs. 40
5.24 Fluid simulation without gravity case 7 graphs. 41
5.25 Fluid simulation without gravity case 8 graphs. 41
5.26 Fluid simulation without gravity case 9 graphs. 41
5.27 Fluid simulation without gravity case 10 graphs. 42

Chapter 1

Introduction

1.1 Motivation

In Smoothed Particle Hydrodynamics (SPH) based fluids simulations, the operation to find the neighbors given a certain fixed threshold of each particle is the most costly operation that has to be performed at each iteration of the simulation. We will call this problem Moving Points Fixed Radius Nearest Neighbors (MPFRNN), because we want to compute the fixed radius near neighbors (FRNN) of every point in a point set, while they are moving.

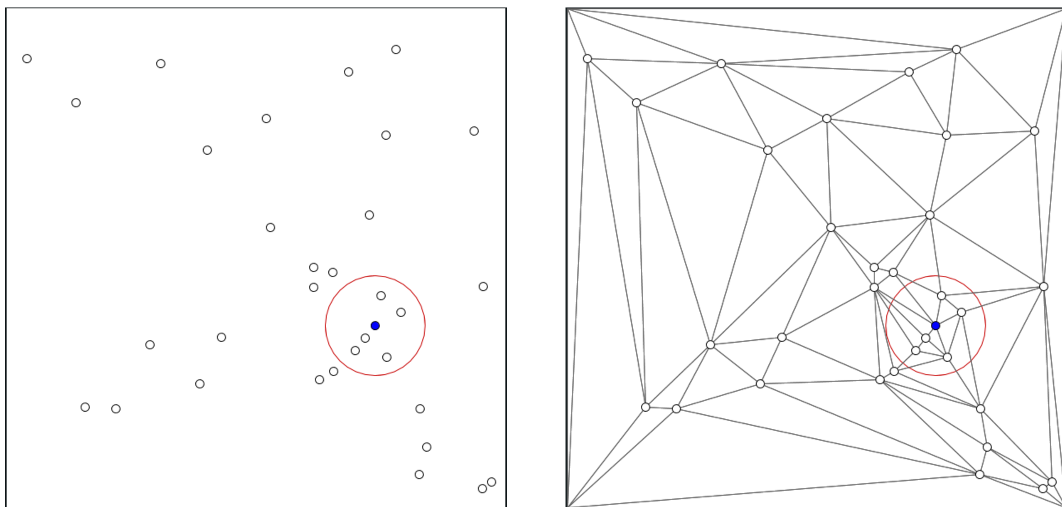


Figure 1.1: FRNN of a particle, and Delaunay triangulation of the point set

This work focuses on the development of a new data structure to solve MPFRNN using Delaunay triangulations in simulation settings without preserving the excluded volume of every particle (where particles can pass through each other), exploring the scope of this algorithm and showing that this approach is parallelizable in GPU. It also aims to improve the time this process takes in simulation scenarios where particles move slowly, or in which there are too many particles within the FRNN of every particle.

1.2 Research Questions

- Is a Delaunay based data structure a competitive approach (on time and space usage) to solve MPFRNN in CPU based simulations compared to other approaches based on Cell Linked Lists?
- If we define the density of a particle simulation as the mean FRNN size, and the speed of the simulation as the average speed of the particles on it. Does the proposed method compute faster the FRNN of each particle than other methods on slower and denser simulations?
- If it is true that the proposed method is faster than cell based ones only in slower or denser particle regions. What is the threshold of that benefit? How dense or slow does a set of particles have to be to take advantage of this approach?
- What is the scope of the proposed approach? Which simulation scenarios benefit the most from it?

1.3 Hypothesis

A Delaunay approach will have a better performance than all the other state of the art methods for solving MPFRNN in an interactive setting where particles move slowly or have a dense neighborhood.

1.4 Objectives

1.4.1 General Objectives

Evaluate what is the scope of a sequential algorithm based in Delaunay triangulations to calculate MPFRNN in different simulation settings. We also explore a parallel version of this algorithm, studying what difficulties exist with this parallelization.

1.4.2 Specific Objectives

1. Design, implement and evaluate a sequential algorithm based on Delaunay triangulations to calculate MPFRNN in simulation settings.
2. Compare and show that the designed algorithm is faster than other sequential solutions to the problem in some scenarios, and determine what those scenarios are.
3. Develop a library to test the algorithm.
4. Explore a parallel version of this algorithm, studying what the difficulties of this parallelization are.

1.5 Methodology

1. Review which methods are being used nowadays in order to solve the MPFRNN problem.

2. Design a sequential algorithm based on Delaunay triangulations to solve the MPFRNN problem.
3. Develop a C++ implementation of the proposed algorithm and a baseline algorithm.
4. Explore a parallel version of the proposed algorithm, studying what the difficulties of this parallelization are.
5. Define sets of experiments with different settings of particle based simulations varying in particle velocity, particle density and FRNN radius, in order to determine the influence of those factors on the performance of the algorithm.
6. Use said sets to compare the proposed sequential approach with other the baseline algorithm in terms of update and query times at every time step of each simulation.

1.6 Structure of this work

The organization of this thesis is the following:

- Chapter 2 is a brief explanation of the theoretical background needed to understand the problem, state of the art solutions, and the proposed algorithm. It starts reviewing fundamental computer geometry topics needed to understand Delaunay triangulations and its subproblems, and a summary on parallel computing and the CUDA GPU programming model.
- Chapter 3 is a state-of-the-art review on solving the MPFRNN problem, and also a description of the related work on Delaunay triangulations.
- Chapter 4 is a description of the designed algorithm. In this chapter we will describe the proposed sequential algorithm, explore a parallel version of it and discuss its difficulties. In order to do that, we will show how a Delaunay triangulation is built and then maintained while its vertices move (sequentially and in parallel) at every time step, and then we will show how we use these data structures to calculate the FRNN of the particles.
- Chapter 5 is a description and analysis of the experiments performed.
- Chapter 6 is a summary of every conclusion made in this work.

Chapter 2

Theoretical Background

In this chapter we review the necessary foundations to understand how we solved MPFRNN using Delaunay triangulations. First, we define some concepts in Computer Geometry and triangular meshes that are necessary to define and compute a Delaunay triangulation. Finally, we give some insights in parallel computing and GPU programming, to give the foundations on which we will explore the parallel version of this algorithm.

2.1 Computer geometry fundamentals

Computer geometry is the subfield of computer science devoted to the study of problems that can be stated in terms of geometry. In this section we do not intend to describe the whole field, but to summarize the necessary results needed in order to define and compute the Delaunay triangulation of a set of points. In this thesis, we will only deal with 2D geometry and 2D triangulations, but most of the definitions, theorems and algorithms described in this section also apply to higher dimensions (all of them, except the flipping of the triangle meshes). For a more detailed introduction to the topic, we recommend the following sources [25, 26, 8].

2.1.1 Common geometric tests

There is a set of geometric tests, or predicates, in the field of computer geometry that commonly appear in branching steps of various algorithms. In practice that means that the correct evaluation of these tests is of critical importance in the correct behavior of the algorithms.

Two classic geometric tests of special importance in the context of this work are *Orient2D* and *InCircle*.

Given 3 points, a, b and c, *Orient2D* is defined as:

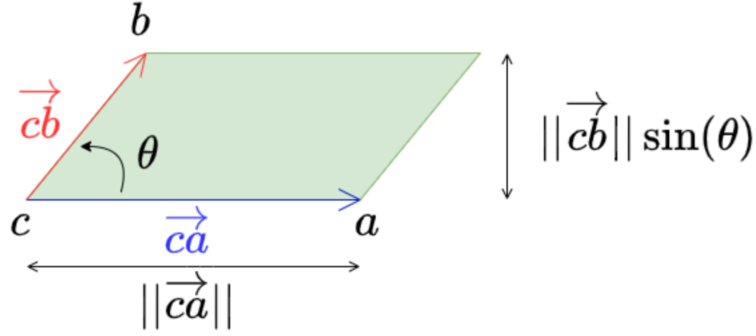


Figure 2.1: Area of the cross product.

$$Orient2D(a,b,c) = \begin{cases} 1 & \text{if } c \text{ lie to the left of the semiplane defined by the line } \overline{ab} \\ 0 & \text{if } c \text{ lie on the line } \overline{ab} \\ -1 & \text{if } c \text{ lie to the right of the semiplane defined by the line } \overline{ab} \end{cases} \quad (2.1)$$

Orient2D can be easily calculated using the fact that the cross product indicates if 3 points are in CCW order or not, solving one of the following equivalent matrix determinants of the vectors $\overrightarrow{ca}$ and $\overrightarrow{cb}$:

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \quad (2.2)$$

If the determinant is positive, then the points are ordered in CCW order, if it is 0 the points are collinear and if it is negative the points are in non CCW order. That way we can check those cases to define the *orient2d* test.

This fact is illustrated in figure 2.1. If θ was measured in clockwise order (non CCW order), the area would be negative.

One common example of how to use this geometric test is the calculation of the intersection of two line segments. Let us say that we want to check whether of not the segments $S(a, b)$ and $S(c, \mathrm{d})$ intersect properly with each other (the intersection point does not coincide with any of the four extremities points a, b, c or d). We can simply check if *orient2d(a,b,d)* has a different sign that *orient2d(a,b,c)*, while *orient2d(c,d,a)* has a different sign that *orient2d(c,d,b)*, and all of them are not 0. The motivation behind this is to check if the points d and c are in the opposite semiplanes defined for the line passing through $S(a, b)$, and also the same for the points a and b about the segment $S(c, \mathrm{d})$.

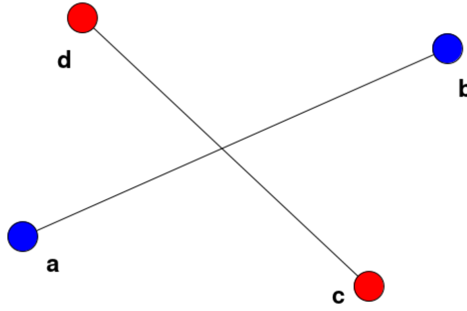


Figure 2.2: Segments intersection.

Given 4 points, called a, b, c and d, the test *InCircle* checks if d is inside the circle defined by a, b and c, assuming that they are in a CCW order (otherwise, the results will be reversed).

InCircle is defined as:

$$\textit{InCircle(a,b,c,d)} = \begin{cases} 1 & \text{if d lies inside the circle defined by } a, b \text{ and } c \\ 0 & \text{if d lies on the circle defined by } a, b \text{ and } c \\ -1 & \text{if d is outside the circle defined by } a, b \text{ and } c \end{cases} \tag{2.3}$$

This test can be easily calculated solving one of the following determinants:

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ \mathrm{d}_x & \mathrm{d}_y & \mathrm{d}_x^2 + \mathrm{d}_y^2 & 1 \end{vmatrix} = \begin{vmatrix} a_x - \mathrm{d}_x & a_y - \mathrm{d}_y & (a_x - \mathrm{d}_x)^2 + (a_y - \mathrm{d}_y)^2 \\ b_x - \mathrm{d}_x & b_y - \mathrm{d}_y & (b_x - \mathrm{d}_x)^2 + (b_y - \mathrm{d}_y)^2 \\ c_x - \mathrm{d}_x & c_y - \mathrm{d}_y & (c_x - \mathrm{d}_x)^2 + (c_y - \mathrm{d}_y)^2 \end{vmatrix} \tag{2.4}$$

If the determinant of the matrix is positive then the point d is inside the circumference. If it is 0 then the point is on the circumference, and if it is negative the point is outside the circumference. That way, evaluating the matrix and checking those conditions, we define the test *InCircle*.

Figure 2.3 summarizes the geometric tests *orient2d* and *InCircle*.

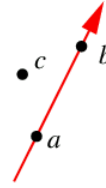
2.1.2 Precision and Robustness

In general, a floating point number is represented by a triplet ($\textit{mantissa}$, β, e) which represents the number $\textit{mantissa} \times \beta^{\mathrm{e}}$. For example, the number 13, or 1101_2 in binary, is represented as 1.101×2^3 in floating point format.

The most common representation for real numbers is the IEEE Standard 754, which has been reviewed several times [31, 7]. This standard not only dictates the floating point num-

Orientation

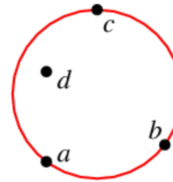
Does c lie on, to the left of, or to the right of $\overrightarrow{ab}$?



$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}$$

Incircle

Does d lie on, inside, or or outside of abc?



$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} = \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix}$$

Figure 2.3: Summary of the geometric tests. Image retrieved from `https://www.cs.cmu.edu/~quake/robust.html`

ber representation used, but also defines algorithms for every common operation (addition, subtraction, etc.), ensuring that these operations are equivalent in most computers. It uses a fixed number of bytes for every operation, rounding numbers when necessary due to differences in the exponents of the operands, which inevitably leads to numeric errors.

Unfortunately, it is impossible to represent all real numbers in a finite number of bits using floating point arithmetic. Also, operations between floating point numbers leads to errors too, which makes it inevitable to get precision errors when dealing with real coordinates in computer geometry, as shown by Goldberg in [12].

2.2 Triangle meshes

We will define a triangulation of a point set the following way: Given a set of 2-dimentional points S, a *triangulation* of S is a collection $\mathcal{T}$ of triangles, whose vertices are all the points in S, such that it satisfies the following conditions [30]:

1. All faces of the triangles of $\mathcal{T}$ are in $\mathcal{T}$. (Closure Property)
2. The intersection of any two triangles of $\mathcal{T}$ is a common edge between those, or is empty. (Intersection Property)
3. The union of these triangles is equal to the convex hull of S. (Union Property)

We call a triangle mesh the computer representation of a triangulation and the relation

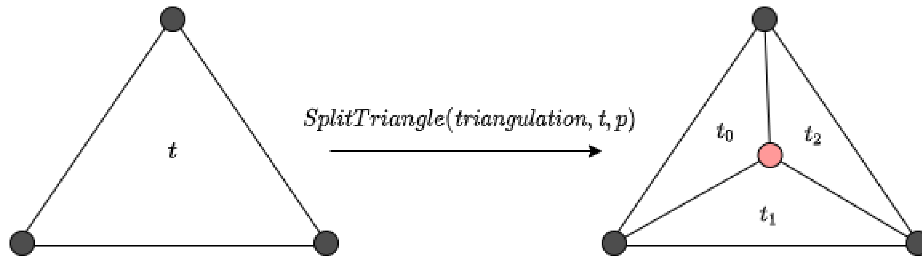


Figure 2.4: Graphic representation of the SplitTriangle function.

between its triangles or its edges.

2.2.1 Triangulation of a set of points

Let's say that we have a set of 2-dimentional points S, we will show that it is always possible to triangulate it, assuming that there are no 3 collinear points, using an incremental algorithm.

Algorithm 1: Incremental triangulation of a set of points.

Input: S
Output: Triangulation of S

1 *triangulation* ← Triangle arround S `// We start the triangulation with a triangle arround` S
2 **for** *Every point p in S* **do**
3   t ← triangle in *triangulation* that contains p
4   SplitTriangle(*triangulation*,t,p)
5 Remove the initial triangle from *triangulation*
6 Add the convex hull of S to the triangulation

1 **Function** `SplitTriangle(`*triangulation*, t, p`)`:
2   a, b, c ← points in t
3   t_0 ← triangle formed by a, b, p
4   t_1 ← triangle formed by a, c, p
5   t_2 ← triangle formed by b, c, p
6   Remove t from *triangulation*
7   Include t_0, t_1 and t_2 in *triangulation*
8   **return**

2.2.2 Flipping edges

Theorem 2.1 (Edge Flip) *Let $\mathcal{T}$ be a triangulation of a point set S. Given an interior edge e_{ab} (not part of the convex hull), we can flip this edge e_{ab} to another edge e_{cd}. This, in other words, means removing the edge e_{ab} and inserting the edge e_{cd}.*

2.2.3 Data structures

The most important things to take into consideration when choosing a data structure to represent a triangle mesh are memory consumption, operations supported by the data structure,

and data locality (this is especially important in parallel representations, as we will see in section 2.3). As we intend to design an algorithm based on flipping edges of a triangulation to calculate the MPFRNN, we will need the mesh to be able to flip edges easily, and to calculate the neighboring triangles of any vertex.

The simplest triangle mesh data structure consists only on a buffer of triangles, which has 6 floating point numbers per triangle, representing the position of every vertex on that triangle. This data structure does not support any operation, and it is very memory inefficient using 3 floating point numbers per triangle, representing at least 2 times every vertex (as in a triangle strip mesh).

One simple improvement to that structure is the addition of a buffer of vertices. That way, the buffer of triangles has 3 integers per triangle, which represent indices to the vertices buffer and the vertices buffer stores the position of every vertex. This data structure still does not support any operation, but it is very memory efficient, using only $3|t|$ integers and $2|v|$ floating point numbers.

A third buffer, called neighbors, is included in order to actually support operations on the previous data structure. This buffer consists on 3 integers per triangle, and each one represents indices to its neighboring triangles. That way, we can locate the neighbors of every triangle, only adding $3|t|$ integers. It is important to note that this representation allows us to flip two triangles, knowing their indices.

This data structure computes the triangles that share a vertex rather slowly, but it can be made faster adding one last buffer. This buffer will record for every vertex 1 integer, representing an index to the triangles buffer. That is the index representing one of the triangles that contains that vertex.

It is worth mentioning that some data structures used to represent meshes do not build up from buffers representing their faces, but from buffers representing their edges as the quadedge [15] mesh and the halfedge [21] mesh.

For a more detailed discussion on different common data structures, we refer the reader to the second chapter of [2].

2.2.4 Delaunay triangulations

Among all the possible triangulations of a point set, one of special interest is the Delaunay triangulation. A triangulation is a Delaunay triangulation if no vertex is inside any of the circumcircle defined by its triangles. We also say that an edge $\mathrm{e}_{a,b}$ of the triangulation is locally Delaunay if it either is on the convex hull, or its opposing angles sum less than 180°.

In this work we are going to use some Delaunay triangulations properties that are well known in the literature:

Property 2.2 (Closest neighbor property) *For every point in a point set S, on its Delaunay triangulation there is an edge that connects it with its nearest neighbor.*

Lemma 2.3 (Delaunay lemma) *If every edge of the triangulation $\mathcal{T}$ is locally Delaunay, then the triangulation is Delaunay.*

Proof of properties can be found in the book Computational geometry by Mark De Berg et al. [8].

A theorem shown by Lawson [18] says that:

Theorem 2.4 *Given any two triangulations of a 2D point set S, say T' and T'', there exists a finite sequence of flips by which T' can be transformed T''.*

There are lots of algorithms to build the Delaunay triangulation, but the Lawson algorithm [19] has a special importance to this work. This algorithm works building any triangulation from a 2D point set, and then flipping every edge that is not locally Delaunay and can be flipped (the triangles that share it form a convex quadrilateral), until all the edges are Delaunay, which means by the Delaunay Lemma 2.3, that the triangulation is Delaunay. It has been shown that this algorithm is correct, and it always terminates with any starting triangulation [30, 18].

2.3 GPU Computing concepts

In order to explore a possible parallel algorithm to solve the MPFRNN running on a Graphical Processing Unit (GPU), we need some background on GPU architecture and GPU programming models. We will use NVIDIA CUDA [23, 24], which is a general purpose parallel platform and model implemented in NVIDIA GPUs.

2.3.1 GPU architecture

Computing Processing Units (CPUs) are designed to compute complex and general tasks with a lot of branching, prioritizing bigger cache memory and smaller ALUs, enabling lower latencies and faster branching execution. Meanwhile, GPUs are designed to perform highly parallelizable arithmetically intensive tasks that require little to no synchronization between each other, favoring bigger instruction throughput and memory bandwidth. This difference in objective inevitably leads to great differences between those two in terms of transistor usage.

In a GPU, a bigger part of the transistors are dedicated to data processing than in CPUs, which is beneficial for highly parallel computations, because it allows the parallel execution of a program for different data elements many more times than a CPU would allow. While CPUs can execute few tens of threads in parallel, GPUs can execute thousands of threads in parallel.

The CUDA parallel programming model divides each GPU on several Streaming Multi Processors (SMP). Each one of those SMPs execute a block of threads in parallel. The order of execution of a block is not fixed and can be at the same time of other block, or in parallel with other blocks, depending on the GPU. As shown in figure 2.6, this means that

the execution order of a set of blocks of threads varies given different GPUs.

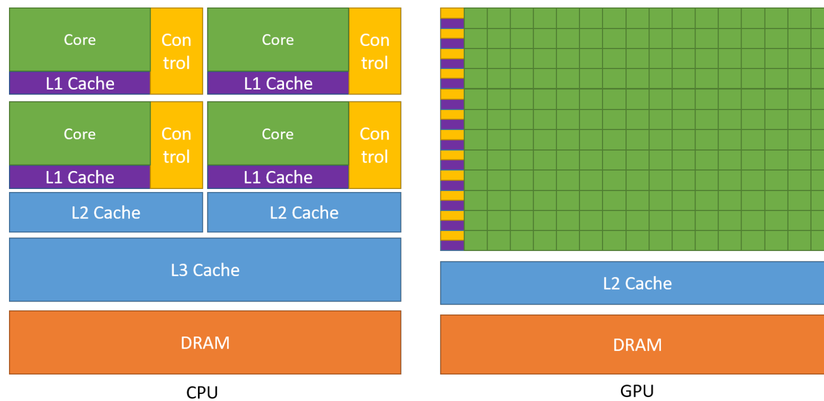


Figure 2.5: Comparison of physical space utilization between CPU and GPU architectures. Image retrieved from the CUDA Toolkit Documentation, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

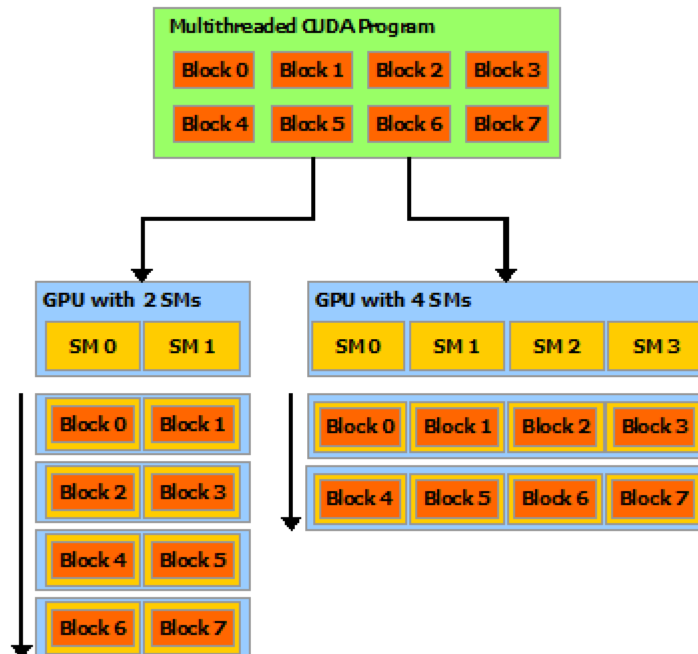


Figure 2.6: Automatic scalability in CUDA model. Image retrieved from the CUDA Toolkit Documentation, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

2.3.2 GPU programming

CUDA allows the definition of a special class of functions called kernels, which can be executed in the host or the device processor. The CUDA programming model assumes that CUDA threads are executed on a physically separate device that operates as a coprocessor to the host running the main program. This is the case when the kernels execute on a GPU and the rest of the main program executes on a CPU. CUDA also assumes that both the host and the device maintain their own separate memory spaces in RAM and VRAM, referred to as host

memory and device memory, respectively. Therefore, a program manages the global memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory. This programming model is also called heterogeneous programming, and is pictured in figure 2.7.

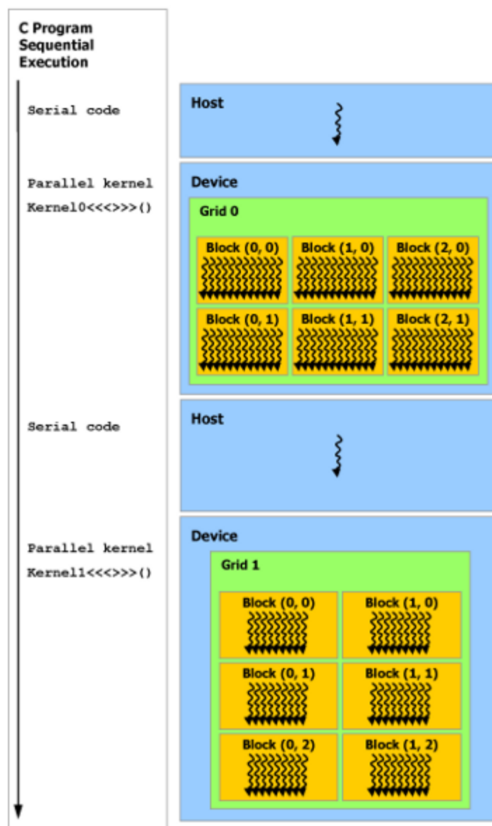


Figure 2.7: CUDA Heterogeneous programming. Image retrieved from the CUDA Toolkit Documentation, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

CUDA exposes three important abstractions: a hierarchy of thread groups, shared memory, and barrier synchronizations. These abstractions provide fine-grained data parallelism and thread parallelism (allowing control over what each thread does), nested within coarse-grained data parallelism and task parallelism (allowing control over how those threads cooperate within a thread block).

The threads in a CUDA application are grouped first in blocks, which can be one, two or three-dimensional, containing up to 1,024 threads on every direction, depending on the GPU. Those blocks are grouped in grids, which can also be three-dimensional, and can have up to 65,535 blocks in each direction.

There are 3 different levels of memory that a CUDA kernel can access, as shown in figure 2.8. First, every thread has its own fast registers. Second, every block of threads has a shared cache memory of around half a MB, with less latency that the global memory. Third, there is a global memory buffer that can be accessed by every thread in a kernel execution. This memory is way slower in comparison to the other 2, but also bigger.

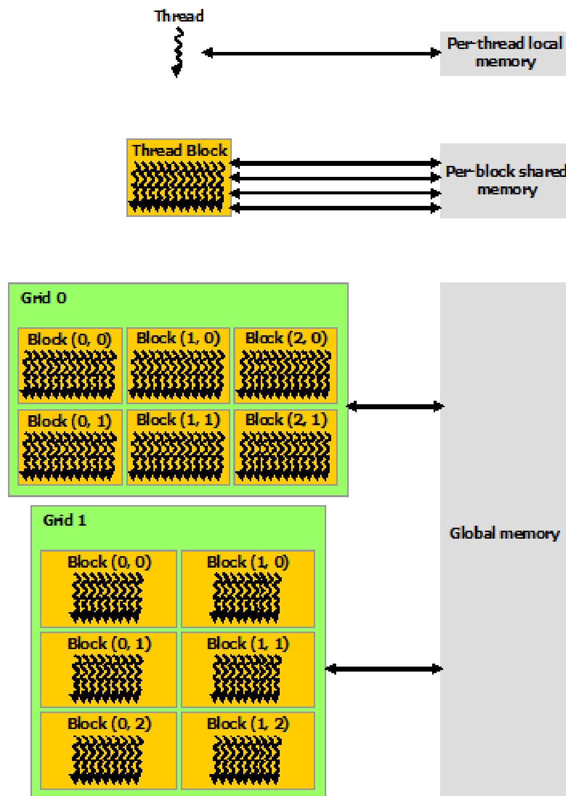


Figure 2.8: CUDA memory hierarchy. Image retrieved from the CUDA Toolkit Documentation, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Chapter 3

Related Work

In this section of the thesis, we describe the most common algorithms and methods used to solve the MPFRNN problem. We start with the most used data structure, the Cell Linked Lists. We then give a walkthrough in the related work about Delaunay triangulations, which we build in our algorithm to solve this problem in a novel way. Finally, we describe the different approaches that have been taken to deal with the finite floating point numbers precision problem in related works.

3.1 Cell Linked Lists

Cell Linked Lists (CLL), are data structures used to maintain an approximate Fixed Radius Nearest Neighbors (FRNN) of moving points, also called Neighboring List (NL). It first divides the simulation box into cells of size equal to the radius of the FRNN, and computes a linked list of particles that correspond to every cell. That way, the particles that can be in the FRNN of a given particle also have to be in the neighboring cells -or the same cell- than that particle. This structure is improved, maintaining not a list of the particles, but a certain order in the particle position buffer. That way, at a cost of sorting the buffer of particles according to their containing cell position index, we can store only the indices of the first particle in the cell and the last one using only $2L$ additional space, where L is the amount of cells.

If the amount of cells in a simulation is rather small compared to the amount of particles ($L < n$), this structure can be built using the cell number of a particle as keys with sorting algorithms that do not compare keys, such as counting ($O(n + L)$ time and space) or radix sort ($O(\log(L)n)$ time and $O(\log(L) + n)$ space). The reason we can do this is that in this context we do not mind the exact order of the particles given their position, but their cells' position. This structure can be queried in $O(nk)$ time for each particle, where k is the amount of particles that fit into a cell (that is because these particles have some radius that cannot be trespassed by another particle), with a required space of $2L$.

This data structure is easy to build in parallel, in time equal to the parallel sorting

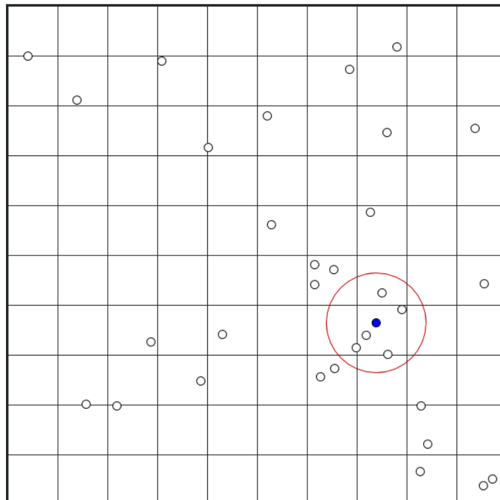


Figure 3.1: Graphic representation of Cell linked lists of a 2D point set.

algorithm used, as described by Hoetzlein [16], so is quite common that it is built in every time step in the simulation, even though there are some methods to update it, as mentioned by Gonnet [13]. It can also be implemented in more than 3 dimensions quite easily.

3.2 Related work on Delaunay triangulations

Guibas et al. [14] proposed a simple incremental algorithm -the randomized incremental algorithm- to build the Delaunay triangulation of a set of points in $O(n\log(n))$ time and additional $O(n)$ space, using an auxiliary data structure to accelerate the point location. Mucke et al described [20] a modification of this algorithm -the jump and walk algorithm- without the auxiliary data structure, which takes $O(n^{3/2})$ time. To locate where is the triangle that has to be modified in order to insert a point, this algorithm starts from a guess triangle and walks through the triangulation towards the triangle that it is looking for, following a straight line. Buchin in [3] showed that an incremental construction with biased randomized insertion order using a space filling curve guarantees an $O(n\log(n))$ time when no degenerate cases happen without the use of any auxiliary data structures.

Turau [32] showed that Delaunay triangulations, once built, can be used to solve the FRNN problem in $O(k)$ time (k is the number of neighbors of the vertex) in a static sequential setting. Then Devillers et. al [10] designed an algorithm to keep a dynamic version of the Delaunay triangulation, which supports point insertion in $O(\log(n))$ expected time and deletion in $O(\log(\log(n)))$ expected time.

Machado et. al. [9] designed an algorithm to keep a Delaunay triangulation for moving points. This algorithm is based upon the deletion and reinsertion of every moving point, with some optimization. They define safe regions where points can move without invalidating the triangulation Delaunay property, intersecting the safe regions of every bicell that contains each point. After that, they compute if a point movement inverts the orientation of

a triangle. If it does not, then the mesh can be fixed flipping edges near the vertex. These optimizations are intended to avoid the deletion and reinsertion of vertices from the mesh, which they claim is the most costly operation of this algorithm in practice.

Delaunay triangulations have been used by Carter et al. [5] to correct intersections in slow moving particles in GPU, taking advantage of the fact that every vertex in a Delaunay triangulation shares an edge with its closest neighbor. The mesh data structure used in this work to represent the triangulation, introduced by Navarro et al. [22], is especially designed to flip edges in parallel fast on the GPU. At every time step of the simulation and for every edge, they check if the edge in the new position complies the Delaunay condition, and if it does not, it gets flipped. They also check if a particle gets outside its cavity and flips one triangle outside it. In said case, they fix the triangulation performing flips. The said data structure does not allow querying of the star neighborhood of a vertex, so it cannot be used to solve the FRNN problem without modifications.

There have been attempts to design parallel algorithms designed to run on the GPU to build Delaunay triangulations from a set of points. Qi et al. [27] computed the Delaunay triangulation of a set of 2D points in the GPU using an approximation of the Voronoi diagram introduced by Rong et al. [28]. Then Cao et al. [4] designed a Divide and Conquer algorithm on the GPU to compute the Delaunay triangulation of 3D point clouds, which could be also applied in 2D point sets.

3.3 Precision problems in computer geometry

We focus our attention in the predicates described in section 2.1.1. These predicates are the fundamental building blocks upon which the Delaunay triangulation is built. As mentioned, these predicates can be determined by computing the determinant of a matrix, and we are interested in whether it results in a positive number, negative number or zero. Computing these determinants using finite floating point arithmetic of any fixed length inevitably falls into precision errors, as mentioned in section 2.1.2.

The simplest technique to work around this issue is to check whether said determinants are smaller than $-\varepsilon$, for the negative case, bigger than ε for the positive case or between those values otherwise, using a small real value for ε. This way we can say that the line that defines the result for *Orient2d* and *InCircle* gets 'thicker'. Therefore, there are non-collinear points that this technique states as collinear, and non-cocircular point that this technique states as cocircular (for example, when points are very close). This technique works well when there are no 3 points close to be collinear, or 4 points close to be cocircular, but this happens oftentimes when we are simulating physical mediums. Kettner et al. [17] showed that there are a lot of cases where the described technique does not work. He also showed that for every ε there is a procedure to build a set of points such that the predicates, calculated with only floating-point arithmetic, get the wrong result.

The obvious solution to this problem is to use exact floating point arithmetic. Unfortunately, this approach leads to a lot of time overhead. Shewchuk in [29] designed, implemented and shared a C implementation of adaptive algorithms to calculate those geometric tests. The said algorithms are adaptive in the sense that they do just as much work as necessary to calculate the sign of the determinant of the matrices, avoiding the calculation of the whole exact result. This technique bounds the possible error of every successive calculation of the value of the determinant, and if the difference between the calculated value and the error can change the sign of the calculated number, computes a more precise approximation, repeating this process until it is certain that the resulting sign is correct. Cao et al. [4] adapted Schewchuk's predicates to partially run in the GPU.

Chapter 4

Proposed Algorithm

In order to support FRNN queries on a set of moving points, we propose to keep a Delaunay triangulation of said points.

The algorithm receives as input a set of different points S in the plane. We assume that the convex hull of this point set does not change. This property can be enforced using a periodic triangulation, or adding 4 points which encloses the triangulation from very far away, such that the points never go outside. In the context of a physically based simulation, these points move with an unpredictable motion, and we have to calculate the FRNN for each one at every time step of the simulation. Put differently, every time we move a point we have to calculate its FRNN at some point. We also make into account that these points are initially uniformly distributed inside a square, in order to simplify the computational cost analysis.

First, we describe the data structure used to implement the algorithm, which has to support the following operations in order to implement the algorithm:

- Edge flipping.
- Given a point in the triangulation, locate a triangle that contains it.
- Given a point p in the triangulation, compute its surrounding cavity. We will call the cavity around p to the set of triangles around the p, plus the edges and vertices that form those triangles.
- Given a triangle, locate all its 3 neighbors.

Then we describe how we maintain the Delaunay triangulation while the points in S are moving. There are 2 key ideas we took advantage of:

- Deleting a point from the triangulation and then reinserting it in a close location is easy and inexpensive.
- If after the movement the point stays within its surrounding cavity, the triangulation can be fixed with flips only. This is still faster than deleting and reinserting the point, as showed in De Castro et al. [9] work.

To query the data structure, we do a breadth first search in the triangulation to determine the FRNN of every point, similar to Turau's work [32].

Then, we explore a parallel version of this algorithm, studying which are the difficulties of this parallelization.

4.1 Sequential Kinetic Delaunay triangulation for small displacements

In this section we will describe the data structure used, then the algorithm to build the Delaunay triangulation and how each of the operations is implemented in a sequential setting.

4.1.1 Data structure

We use an integer to identify every vertex, and a buffer with the information of all the vertices. The information of the vertex identified with an integer i is located at the position i in the buffer. We store the triangles' information in the same way. We use an integer to identify each one, and that integer corresponds to a position in a buffer, which stores the information of all the triangles.

In the vertices buffer, for every vertex, we store its position and the representing integer of a triangle that contains the vertex. In the triangles buffer, for every triangle, we store 3 integers that correspond to the vertices in the triangle, and 3 integers that correspond to the 3 neighboring triangles.

We also make sure that, for every triangle, its neighbors offset in the neighbors buffer (0, 1 or 2), is the same as the vertex in front of that neighbor.

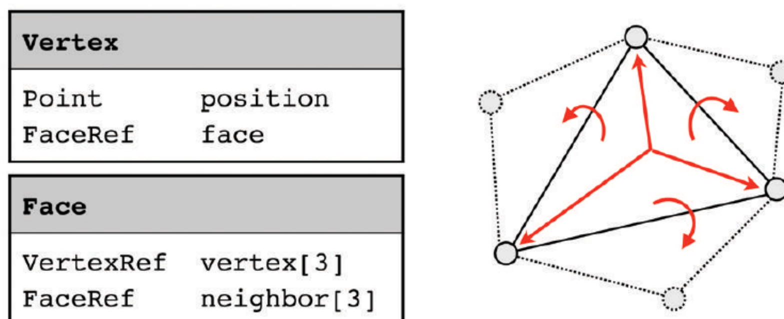


Figure 4.1: Information stored in the used data structure. Image retrieved from [2]

4.1.2 Construction

We use a variation of the algorithm described by Mücke et al. in [20] to build the Delaunay triangulation:

Algorithm 2: Delaunay construction based on Jump & Walk

```
   Input: Set of points S
   Output: A mesh that represents the Delaunay triangulation of S
1  mesh ← initialize the mesh with a triangle around S
2  for Every point p in the set S do
       // locates the triangle, or triangles that share an edge, where p has to be
          inserted
3      t ← jumpAndWalk(mesh,p,lastTriangleInserted)
4      if p is inside t then
5          insertPointInside(mesh,p,t)          // this creates the triangles t1 and t2
6          legalize(t)
7          legalize(t1)
8          legalize(t2)
9      else if p on the edge between t1 and t2 then
10         insertPointOnEdge(mesh,p,t1,t2)      // this creates the triangles t3 and t4
11         legalize(t1)
12         legalize(t2)
13         legalize(t3)
14         legalize(t4)
15 delete the initial triangle around S from the mesh
16 return mesh
```

The algorithm works as follows: We initialize the mesh with just one triangle around the point set, to make sure the *JumpAndWalk* procedure works correctly. Then, for every point p in S, we call the *JumpAndWalk* procedure, which finds either a triangle t in the mesh containing p or a pair of triangles t_1 and t_2 sharing an edge that contains the point. If p is inside the triangle t, formed by the vertices a, b, and c, we remove t and insert the triangles abp, acp and bcp, and then legalize each of those triangles. If p is on an edge $E(a, b)$ shared the triangles t_1 and t_2, which form the quadrilateral a, b, c and d, we remove the triangles t_1 and t_2 from the triangulation and add and legalize the triangles acp, adp, bcp and bdp (note that this triangle does not have to be convex in order to split it). Finally, we remove the initial triangle around the whole point set, and end up with the Delaunay triangulation of S.

The *legalize* procedure makes sure that an edge complies the Delaunay condition, doing the following:

Algorithm 3: Legalize

Input: Mesh m, triangle t

1 **for** *Every neighbor f of t* **do**
2   let a, b, c be the points of f
3   let $a, b,$ d be the points of t
4   **if** *f and t define a convex quadrilateral and (InCircle($a, b, c,$ d)<0 or InCircle($a, b,$ d$, c$)<0)* **then**
5   flip(t,f)
6   legalize(t)
7   legalize(f)

Guibas et al. [14] analyzed the complexity of this procedure. They showed that if we insert the points of the set S in a random order, is expected to take $O(n)$ flips due to the *legalize* procedure. In other words, this step of the algorithm takes an expected constant time every time is called.

The *JumpAndWalk* procedure is the following:

Algorithm 4: Jump & Walk procedure

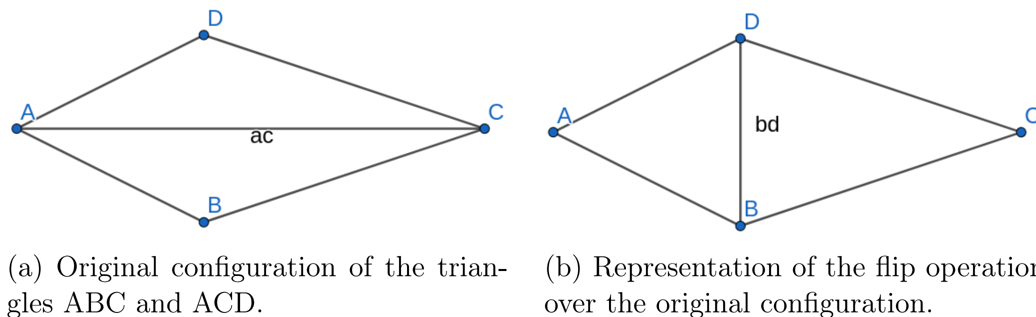
Input: Mesh m, point p, triangle guess t

Output: Triangle or triangles in the mesh m that have to be modified in order to insert p

1 **if** *p is inside t* **then**
2   **return** t
3 **if** *p is on an edge of t shared with its neighbor f* **then**
4   **return** *t and f*
5 $s \leftarrow$ segment between p and the centroid of t
6 **for** *Every neighbor f of t* **do**
7   $q \leftarrow$ segment defined by the edge shared by t and f
8   **if** *s and q intersects proper or improperly* **then**
9   **return** *JumpAndWalk*(m, p, f)

The runtime bounds for this procedure is strongly dependent on the triangle distribution in the space, and the starting triangle guess. If we do not assume anything about these, and remember that there are $O(n)$ triangles in a mesh that contains n points, this procedure can take $O(n)$ time. If we assume that the n points and $O(n)$ triangles are randomly distributed in a square, [20] showed that this procedure walks by $O(\sqrt{n})$ triangles in the worst case. It is important to note that this procedure can take even less time if we start with a better guess, or we have a better triangle arrangement. We also could have sorted the points beforehand using a space filling curve, as in [3], to have always a good initial triangle guess.

Figure 4.2 visually shows the flip procedure. It takes constant time, and can be implemented by manipulating the indices in the data structure.



(a) Original configuration of the triangles ABC and ACD.

(b) Representation of the flip operation over the original configuration.

Figure 4.2: Flip between two triangles

Therefore, if we call n the number of points in S, this algorithm takes an expected $O(n\sqrt{n})$ time if S is located randomly in a square, and takes it a space proportional to the number of points plus the number of triangles, which is $O(n)$.

4.1.3 Moving Points Delaunay triangulation

In order to maintain a valid triangulation while its vertices move, we use a similar approach to that of De Castro et al. [9]. To move a point, we first check whether it stays inside the surrounding cavity before the movement or not. If it stays inside it, the Delaunay triangulation can be fixed by legalizing the triangles in the said neighborhood, removing the need of a call to the *jumpAndWalk* procedure. If that is not the case, the point is removed from the triangulation (maintaining a valid triangulation) following the steps described on algorithm 6, and then reinserted in the next position.

Algorithm 5: Sequential Moving points Delaunay Triangulation

Input: Mesh m, point p and a delta d
Output: The mesh m, where p has been moved by d

1. Move p from mesh.*vertices[p]* to mesh.*vertices[p]* + d
2. sn ← CavityArround(m,p)
3. **if** p *is inside sn* **then**
4.   **forall** *triangles t in sn* **do**
5.   legalize(t)
6.   **return**
7. Move p to its original position, from mesh.*vertices[p]* to mesh.*vertices[p]* - d
8. t,t1,t2 ← DelaunayRemovePoint(m,p)
9. Move p from mesh.*vertices[p]* to mesh.*vertices[p]* + d
10. insertPointDelaunay(p,t1,t2,t)
11. **return** m

It is important to note that in algorithm 5 step 10, where we reinsert the point p, we have to call the *JumpAndWalk* procedure in order to find the triangle where p has to be reinserted. In the worst case this procedure can take as much as $O(n)$ time if all the triangles are in one line and p moves from one side of the triangulation to the other. We improve this using the triangle that is formed in step 8 as a guess in the *jumpAndWalk* procedure, and noting

that this makes this procedure faster in cases where the point does not move too far from its initial position. That allows us to say that in simulation scenarios where the points move slowly, the reinsertion of the point in the triangulation takes $O(1)$ time.

Algorithm 6: DelaunayRemovePoint

Input: Mesh m and point p

Output: The index of the created triangle, and the indices of the freed triangles

```
1  while There are more than 3 triangles that contain p do
2      for All the triangle pairs (t, f) around p do
3          if t and f are neighbors and define a convex cell then
4              flip(t,f)
5  t, t1, t2 ← Indices of the triangles around p
6  a, b, c ← CavityArround(m,p)
7  Edit the indices in m such that t points to a, b and c
8  Edit the indices in m such that t1 and t2 are invalid triangles
9  legalize(t)
10 return m
```

As shown by Guibas et al. [14], the amount of triangles in a point's cavity is not dependent on the number of points in a Delaunay triangulation. That means that the amount of flips that the algorithm 6 has to make is a constant, and the runtime of this procedure is constant too.

To summarize, the runtime of this algorithm, used to maintain the Delaunay triangulation of a set of moving points, is $O(n)$ if we assume that said movement is slow enough.

4.1.4 Moving Points Fixed radius neighbors

Once we maintain the Delaunay triangulation of a set of moving points, we can compute the fixed radius neighbors of every point with a DFS over the graph defined by the Delaunay Triangulation. This approach is similar to the one used by Turau [32].

We do a depth first search (DFS) starting from every point p, which halts every time we find a point that is outside the FRNN. This DFS takes $O(I + E)$, where I is the number of points q, such that the distance between p and q is less than a given r, and E is the number of edges between all the vertices in the FRNN around p. We know, from the properties of the Delaunay triangulation, that $E \in O(I)$, which means that it takes $O(I)$ to compute the FRNN for every point p given the Delaunay triangulation of the point set. See algorithm 7.

Note that it is not possible to leave a point q outside the calculation of the FRNN of another point p performing algorithm 7. It is easy to see that this only could happen if there was no path in the graph to q starting in p with all the nodes in the FRNN of p. Such a path always exists in the graph defined by a Delaunay triangulation.

An important part of this algorithm is to use a flat array to implement the visited abstract data structure in algorithm 7. This variable is used to check whether a point has been already

checked in the DFS, making sure that the distance calculation between point p and any other is computed at most once, using only $O(n)$ additional space.

Algorithm 7: Fixed radius near neighbors in Delaunay Triangulation

```
Input: Delaunay Mesh m and point p
Output: A list of the particles within a radius r around p
   // neighbors will represent the points in p's FRNN
1  neighbors ← empty set
   // visited will represent the points in that have been already checked
2  visited ← empty set
3  visited.add(p)
   // Empty stack for the dfs
4  stk ← empty stack
5  Add to the stack all the points next to p in the mesh
6  while stk is not empty do
7      q ← stk.top()
8      stk.pop()
9      if q is in visited then
10         continue
11     visited.insert(q)
12     if The distance between p and q is ≤ r then
13         neighbors.add(q)
14     Insert all the points around q to stk
15 return neighbors
```

4.2 Exploring Parallel Moving Points Delaunay triangulation

In this section we explore how to parallelize for the GPU the algorithms described the previous section. We start highlighting which parts of those algorithms are parallel in nature and which are sequential. We then give some directions on how we plan to do this, and which things are left to be solved.

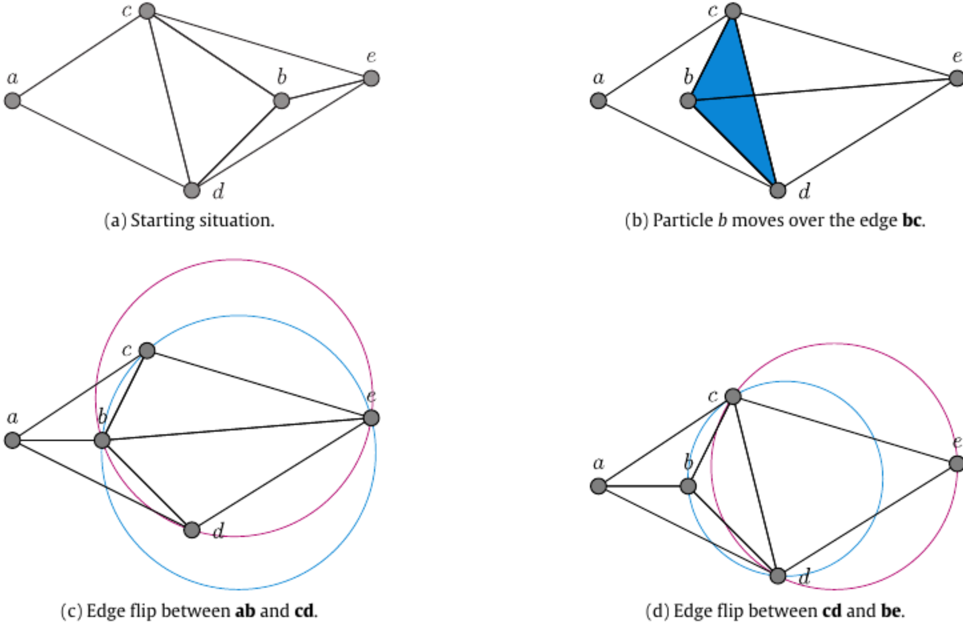
Previous works proved that building the Delaunay triangulation of a set of points taking advantage of the GPU is possible [27, 4]. It is also possible to build the initial Delaunay triangulation in the CPU, and then transform it into a Delaunay triangulation in the GPU as Navarro et al. work [22]. Afterwards, the said Delaunay triangulation can be maintained in the GPU as the particles move without going through each other, as Carter et al. [5] did.

In the figures in the following sections we use blue triangles to denote inverted triangles.

4.2.1 Keeping the moving points Delaunay triangulation

The first approximation to this problem is to implement the algorithm designed by Carter et al. [5]. At every time step of the simulation, the particles get moved in parallel according

to the simulation calculated displacement. It is then calculated which edges are opposing an inverted triangle that has to be flipped, checking if both of the opposite vertices (to the edge) are in the same semi plane defined by the edge (if both are in the same semi plane, then the triangle has been inverted) and one of the opposite vertex of the edge is inside one of those triangles (in order to not flip the wrong edges in this inverted triangle). Then, the triangles that have been inverted due to this movement get fixed using flips, and then every edge in the triangulation that is not Delaunay and that can be flipped is flipped, as shown in figure 4.3.



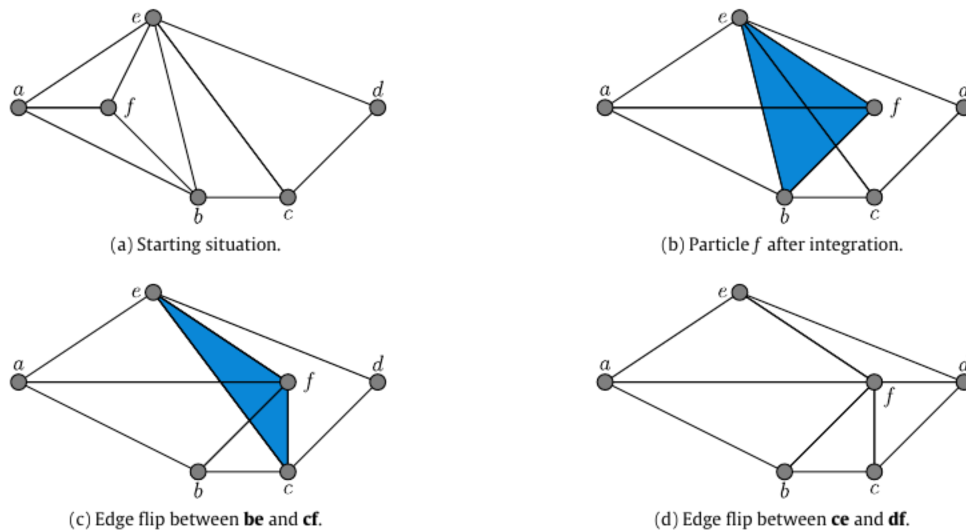
(a) Starting situation. (b) Particle b moves over the edge **bc**.
(c) Edge flip between **ab** and **cd**. (d) Edge flip between **cd** and **be**.

Figure 4.3: Triangle fix and Delaunay condition checking. Image retrieved from [5].

It can happen that a particle moves further away than inside a neighboring triangle. Carter et al. [5] solved this issue going back in the simulation and using a smaller timestep, such that the vertex does not go through more than one edge. They also describe a possible flipping sequence to solve this issue, showed in 4.4. A special case that this flipping sequence does not handle is what happens when a particle gets through another in one time step, ending up over an edge. This case is showed in figure 4.6. They did not handle this case in their work because their simulation imposed the restriction that it must not happen -this restriction in the simulation they performed is called excluded volume, and is widely accepted to be true in that area. We have to deal with it, if we want to design a more robust parallel algorithm.

One possible solution to this problem in figure 4.6 is to modify all the edges that connect vertices with H, to now connect them with G and vice versa, as soon as the particles moves through the other. This is exemplified in figure 4.5. Let n represent the number of vertices and p the number of parallel threads we can run, we can check in $O(\frac{n}{p})$ if this special case happens, storing 2 position buffers. One of those buffers with the previous position of each particle, and the other with the current position of each particle. If the vectors defined by the displacement of two particles, which are neighbors on the Delaunay triangulation, cross each other, it is supposed that the particles crossed each other. Therefore, we can define a

kernel that runs in parallel for every pair of neighbor particles that checks if this special case happened in $O(\frac{n}{p})$ time with an additional buffer of size n.



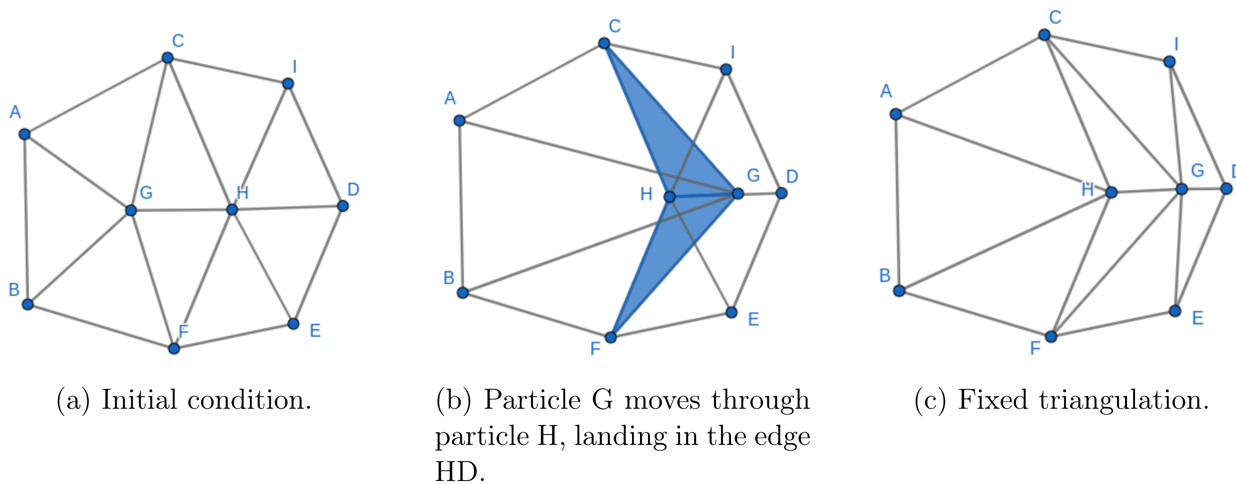
(a) Starting situation.

(b) Particle f after integration.

(c) Edge flip between **be** and **cf**.

(d) Edge flip between **ce** and **df**.

Figure 4.4: Fixing a triangulation when a particle mover further away, performing 2 flips. Image retrieved from [5].



(a) Initial condition.

(b) Particle G moves through particle H, landing in the edge HD.

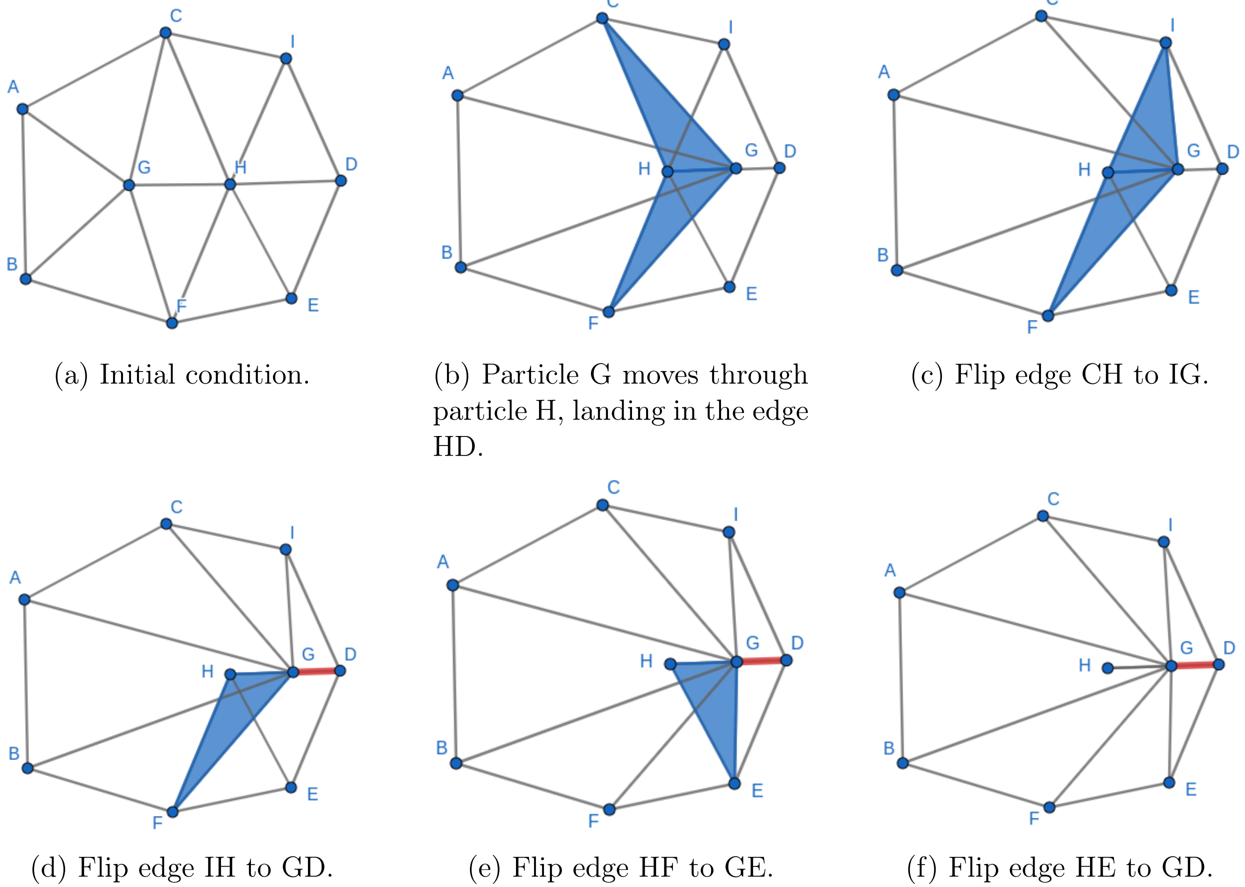
(c) Fixed triangulation.

Figure 4.5: Special case fix.

The algorithm described by Carter et al. [5] works better in situations where particles move slowly. There are particles that do not go far from their initial position and the triangulation does not get too degenerated with the movement, which also means that there are not many inverted triangles to fix, nor too many non-Delaunay edges to flip. But, in simulations where particles can move far away in one time step, this approach flips all the triangles between the particle's initial position and its final position in sequential order to fix the triangulation.

4.2.2 FRNN in GPU using the Delaunay triangulation

Compute the FRNN of every particle on a Delaunay triangulation in GPU is as simple as computing a DFS for every particle, just as in 4.1, but in parallel in the GPU.



(a) Initial condition.

(b) Particle G moves through particle H, landing in the edge HD.

(c) Flip edge CH to IG.

(d) Flip edge IH to GD.

(e) Flip edge HF to GE.

(f) Flip edge HE to GD.

Figure 4.6: Special case of the algorithm described by Carter et al. in [5].

Dickerson & Drysdale [11] showed that a DFS in the graph built from the Delaunay triangulation minus the edges that are longer than the FRNN radius, can still compute the correct neighborhood for every vertex. This means that we can precompute in parallel each of the edge lengths (launching one thread per edge), in order to store if they are longer than the FRNN radius. That information can be used to check at every branching step of the algorithm whether an edge is a valid path for the DFS. The described optimization saves up to at least one distance computation per every edge longer than the FRNN radius, while adding at least one comparison per edge in the triangulation.

Chapter 5

Results

5.1 Experimental setup

In order to be able to tell in which conditions the proposed algorithm performs better, a set of scenarios were gathered and tested against an implementation of the proposed algorithm and a baseline algorithm. At every scenario, we calculated the time that it takes to update and query the data structures. Also, the sum of those factors plus the simulation computation is reported in a graph.

The baseline algorithm is a simple implementation of the cell linked list data structure, shown in code A.1. This implementation could be improved by using techniques discussed in section 3.1, but works as a baseline to tell whether our implementation of the proposed algorithm is competitive with other algorithms in practice. An important aspect of this implementation is that it does not reorder the particles in the position buffer, which is faster in practice. We chose to do that in order to ease the integration of this implementation into the simulation scenarios we already gathered for the Delaunay based algorithm.

We ran all the tests on a computer with an *AMD Ryzen 5 3550H* processor, *8GiB SODIMM DDR4* memory and a *GeForce GTX 1650 Mobile* GPU.

5.1.1 Shaking points

In the shaking points scenario a set of random generated points move with a random generated velocity, bouncing in the walls of the simulation. We tested a set of fixed amount of particles, whose velocities are uniformly distributed between *-maxVelocity* and *maxVelocity*. We fixed the parameters $xsize$ and $ysize$ to 50000. This algorithm is explained in more detail in algorithm 8.

The Said simulation aims to test the relation between the velocity of the particles and the update time of each of the data structures. This scenario was also good for testing the robustness of the implementation of the proposed algorithm, because particles oftentimes go through each other.

Algorithm 8: Shaking points scenario

Input: Set of points S, buffer v with the velocities of each point, simulation dimensions xsize and ysize.

Output: The same point set S with displaced positions

```
1 for Every point p in S do
2     if p.pos.x ≥ xsize or p.pos.x ≤ 0 then
3         v[p].x = −v[p].x
4     if p.pos.y ≥ ysize or p.pos.y ≤ 0 then
5         v[p].y = −v[p].y
6     p.pos = p.pos + v[p]
```

5.1.2 Fluid simulation

We implemented a simple 2D fluid simulation based in the double density relaxation step of the algorithm described in [6]. We have to calculate the FRNN of every particle in order to approximate its density in every time step of the simulation. In this scenario, as in the shaking points one, we did not check for collisions between particles, but let them go through each other.

For every time step of the simulation, in order to calculate the displacement of every particle, we performed the algorithm described in figure 5.1. For more details on how the simulation is performed, see code A.2.

Algorithm 2: Double density relaxation.

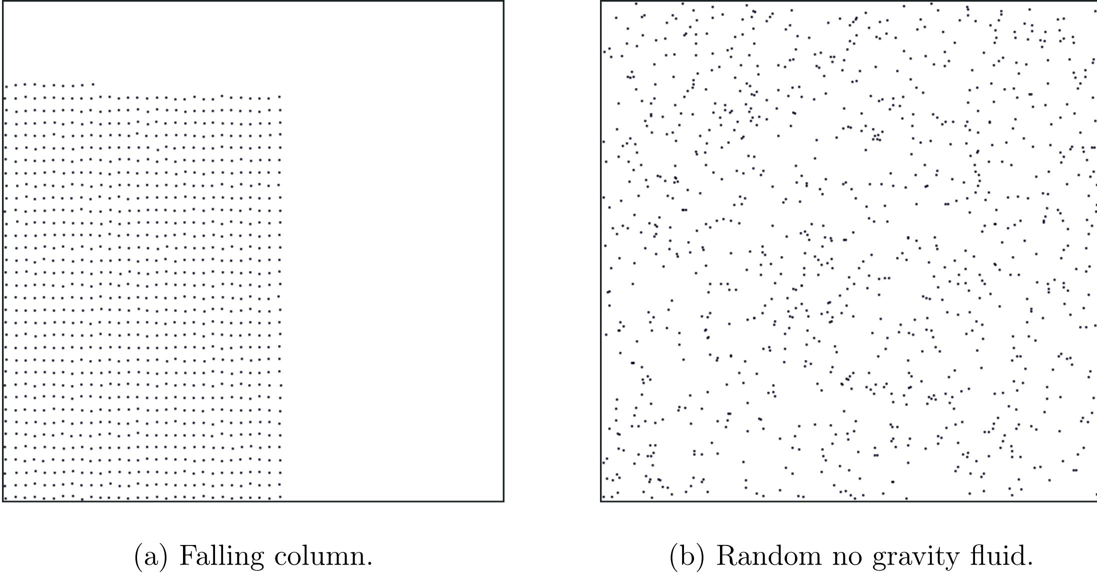
```
1.  foreach particle i
2.      ρ ← 0
3.      ρ^near ← 0
4.      // compute density and near-density
5.      foreach particle j ∈ neighbors( i )
6.          q ← r_ij/h
7.          if q < 1
8.              ρ ← ρ + (1 − q)^2
9.              ρ^near ← ρ^near + (1 − q)^3
10.     // compute pressure and near-pressure
11.     P ← k(ρ − ρ_0)
12.     P^near ← k^near ρ^near
13.     dx ← 0
14.     foreach particle j ∈ neighbors( i )
15.         q ← r_ij/h
16.         if q < 1
17.             // apply displacements
18.             D ← Δt^2 (P(1 − q) + P^near (1 − q)^2) r̂_ij
19.             x_j ← x_j + D/2
20.             dx ← dx − D/2
21.     x_i ← x_i + dx
```

Figure 5.1: Double density relaxation. Image retrieved from the paper 'Particle-based viscoelastic fluid simulation' by Clavet et al. [6].

Important variables in this simulation are:

- x_i is the position of the particle i.
- h is the SPH radius, and also the FRNN radius.
- ρ, ρ^{near}, P and P^{near} are simulation variables that each particle carries, and have to be calculated at each time step of the simulation. ρ and ρ^{near} are the density and near density carried by a particle, while P and P^{near} are pseudo-pressures.
- k, k^{near} and ρ_0 are simulation parameters. k and k^{near} are stiffness parameters in the calculation of P and P^{near}, while ρ_0 is the rest density of the fluid.

This simulation is run with different parameters in order to check how the proposed algorithm performs in simulations with different particle density and velocity. The value of the fixed variables is dt = 1/60, $\rho_0 = 6$, $k = 0.005$ and $k_{near} = 0.4$, while the SPH radius, simulation box size and number of particles, and in the case of the falling wall case, the number of particles in the walls are varied. Also, the particles initially are positioned on two configurations shown in figure 5.2. The simulations are run for 1000 time steps (approximately 17 seconds), which is enough to reach a stable state in the simulation, where particles do not move very much. This time was approximated empirically.



(a) Falling column. (b) Random no gravity fluid.

Figure 5.2: Initial configurations of fluid simulations.

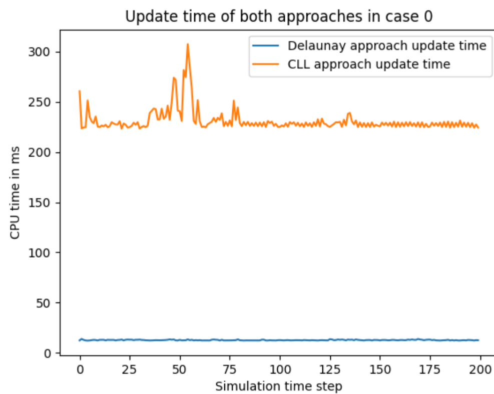
The configuration in figure 5.2a has a gravity of 0.8 while the one in figure 5.2b has no gravity (a gravity of 0.0).

5.2 Shaking points simulation results

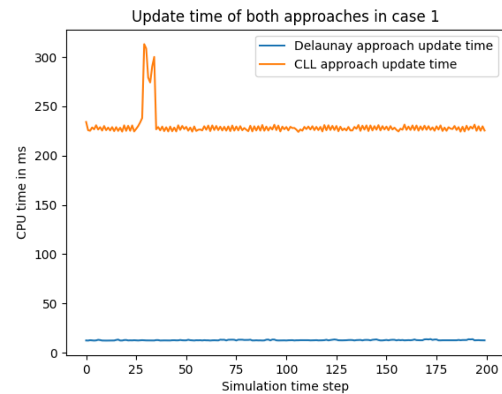
We call each of these shaking particles settings 'cases'. These 12 cases are numbered from 0 to 11, and we measured the update time of these different settings.

Table 5.1: Shaker simulation variables used.

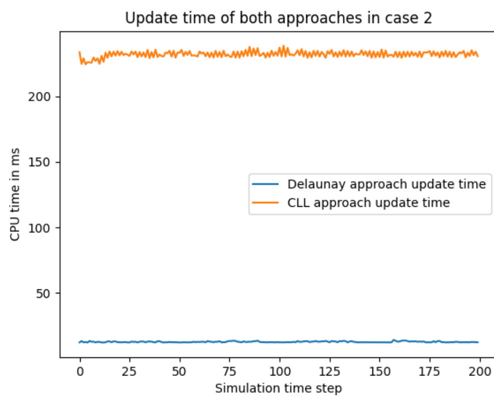
Number of particles	Max Velocity
5,000	200
5,000	500
5,000	1,000
5,000	2,000
10,000	200
10,000	500
10,000	1,000
10,000	2,000
100,000	200
100,000	500
100,000	1,000
100,000	2,000



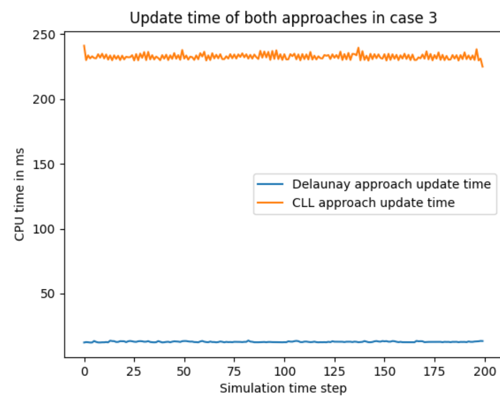
(a) Trace of scenario 0.



(b) Trace of scenario 1.

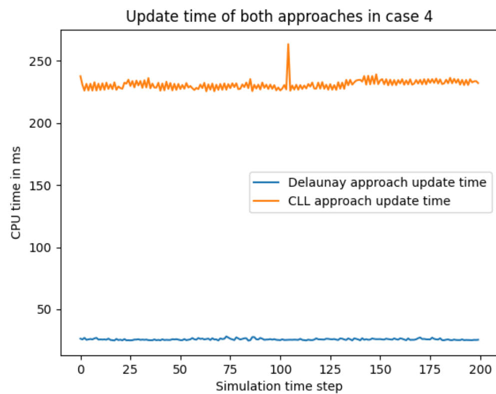


(c) Trace of scenario 2.

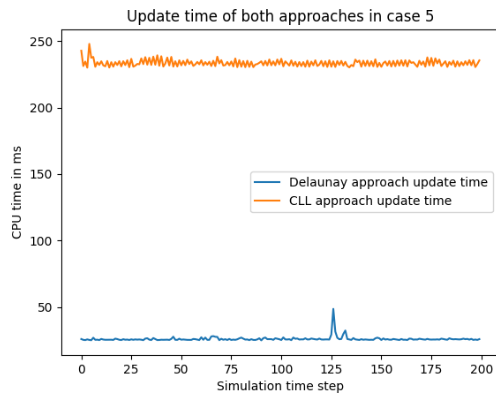


(d) Trace of scenario 3.

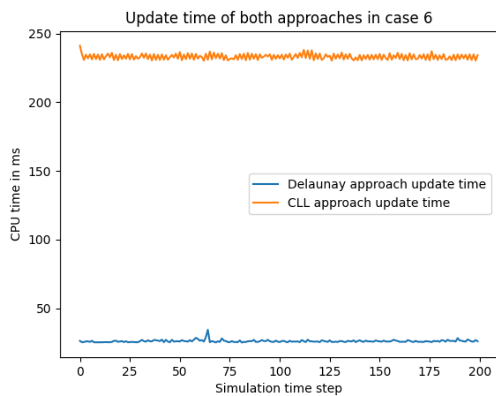
Figure 5.3: Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 5,000 particles and different velocities.



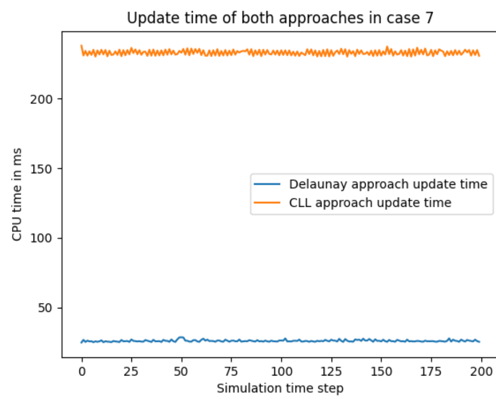
(a) Trace of scenario 4.



(b) Trace of scenario 5.

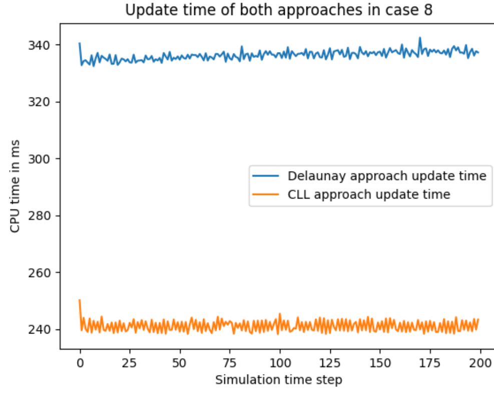


(c) Trace of scenario 6.

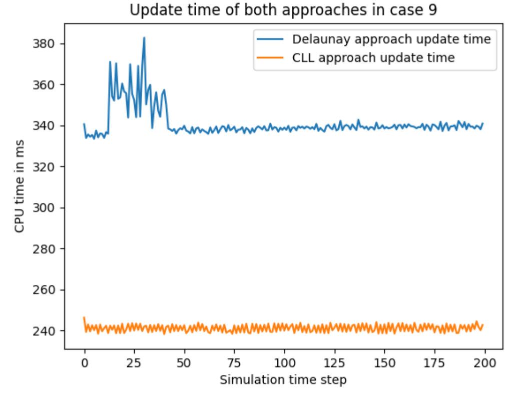


(d) Trace of scenario 7.

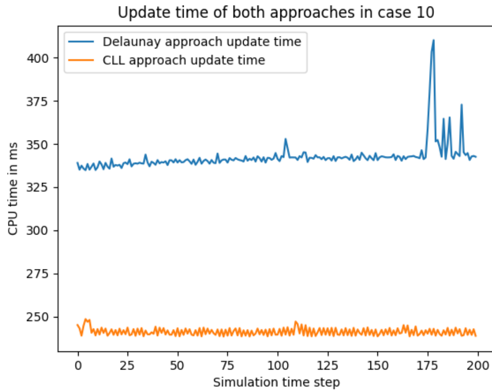
Figure 5.4: Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 10,000 particles and different velocities.



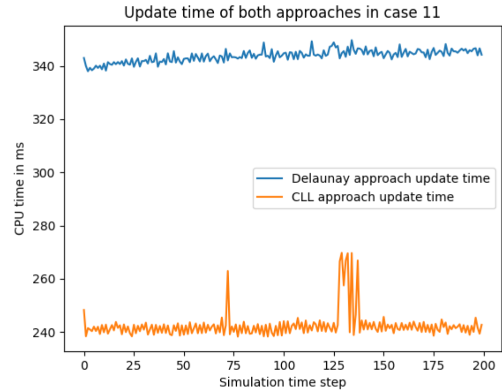
(a) Trace of scenario 8.



(b) Trace of scenario 9.



(c) Trace of scenario 10.



(d) Trace of scenario 11.

Figure 5.5: Shaking points update time comparison between Delaunay and Cell Linked Lists approach. Cases with 100,000 particles and different velocities.

5.3 Falling column fluid simulation results

Table 5.2: Variables set in different settings of the falling column fluid simulations.

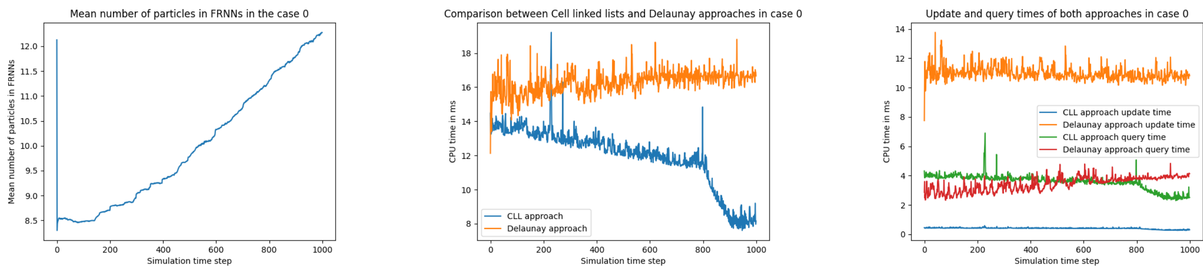
FRNN radius	Simulation box size	N° of particles	N° of particles in wall
200	4,800x4,800	3,000	200*6
300	4,800x4,800	3,000	200*6
400	4,800x4,800	3,000	200*6
400	6,000x6,000	10,000	400*6
500	6,000x6,000	10,000	400*6
600	6,000x6,000	10,000	400*6
600	12,000x12,000	40,000	800*6
700	12,000x12,000	40,000	800*6
700	24,000x24,000	100,000	1,600*6
700	90,000x90,000	100,000	1,600*6

We call each of these falling column settings 'cases'. These 11 cases are numbered from 0 to 10. It is important to note that cases 9 and 10 only differ in the size of the simulation

box.

We decided to show 3 different graphs from every simulation setting:

- The first graph shows the mean number of particles that are within the FRNN radius of every particle, or in other words, the mean size of the FRNN in the simulation. These graphs allow us to see the correlation between this measure and the running time of the algorithms.
- The second graph shows the total running time of every time step of the simulation.
- The third graph shows the update and query time that every data structure takes in every time step of the simulation. This allows us to see what step (update or query) takes more time in the simulation.

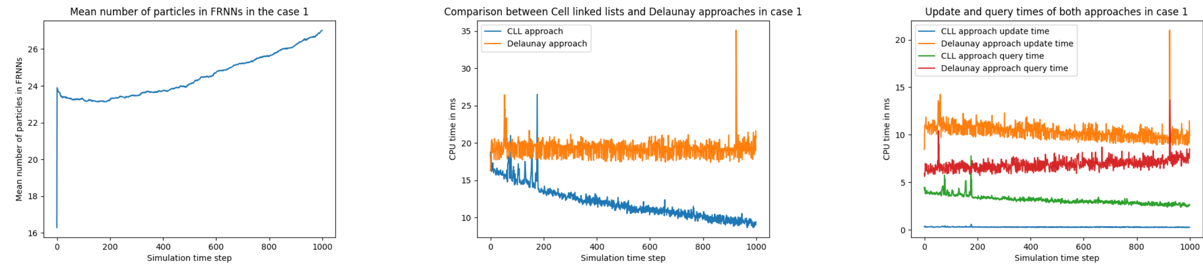


(a) Falling column mean number of particles on the FRNN execution trace scenario 0.

(b) Falling column total time execution trace scenario 0.

(c) Falling column update and query time execution trace in scenario 0.

Figure 5.6: Falling column case 0 graphs.

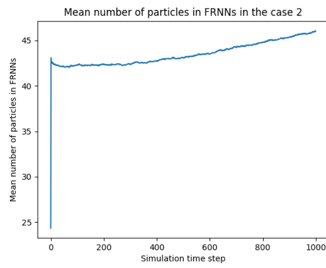


(a) Falling column mean number of particles on the FRNN execution trace scenario 1.

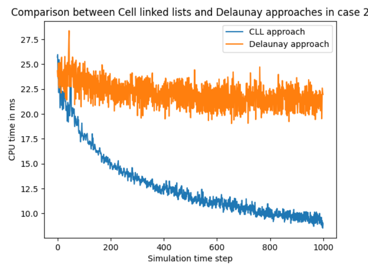
(b) Falling column total time execution trace scenario 1.

(c) Falling column update and query time execution trace in scenario 1.

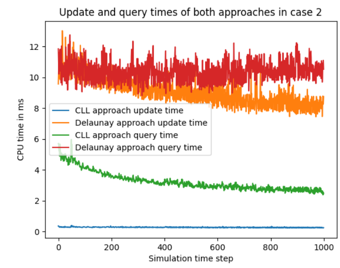
Figure 5.7: Falling column case 1 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 2.

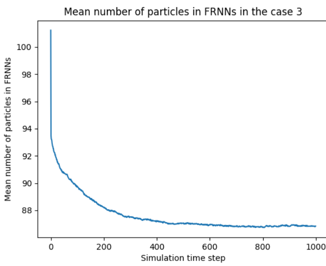


(b) Falling column total time execution trace scenario 2.

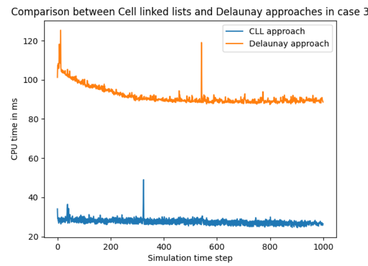


(c) Falling column update and query time execution trace in scenario 2.

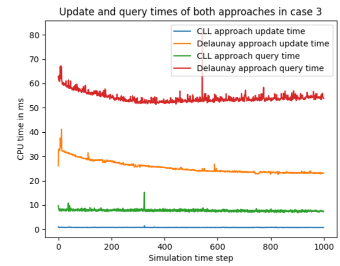
Figure 5.8: Falling column case 2 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 3.

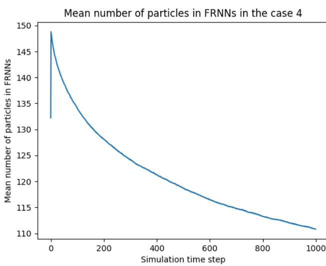


(b) Falling column total time execution trace scenario 3.

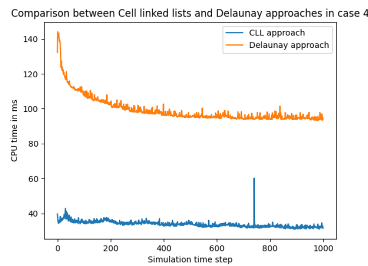


(c) Falling column update and query time execution trace in scenario 3.

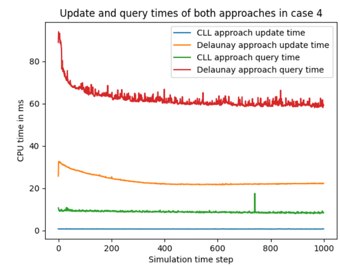
Figure 5.9: Falling column case 3 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 4.

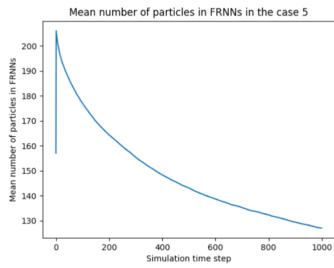


(b) Falling column total time execution trace scenario 4.

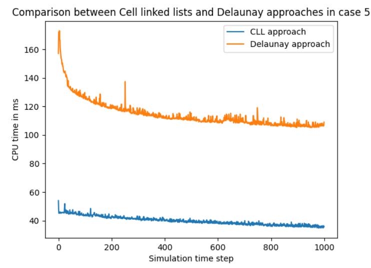


(c) Falling column update and query time execution trace in scenario 4.

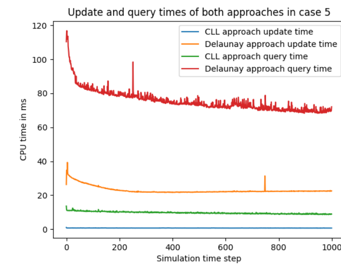
Figure 5.10: Falling column case 4 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 5.

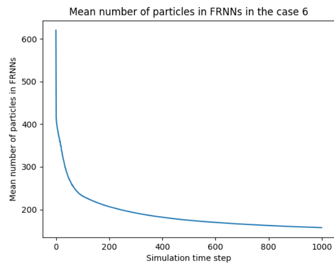


(b) Falling column total time execution trace scenario 5.

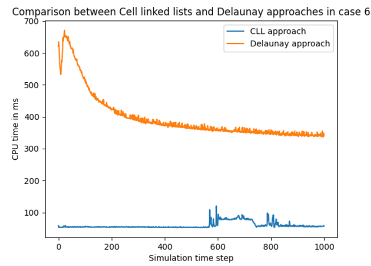


(c) Falling column update and query time execution trace in scenario 5.

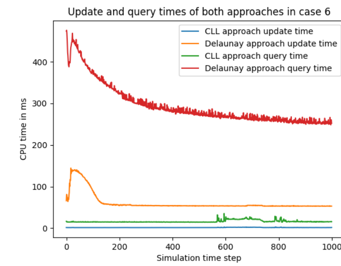
Figure 5.11: Falling column case 5 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 6.

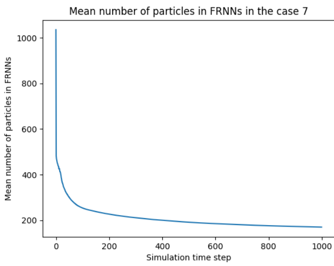


(b) Falling column total time execution trace scenario 6.

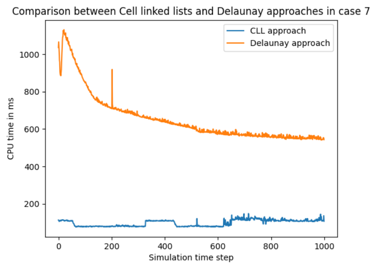


(c) Falling column update and query time execution trace in scenario 6.

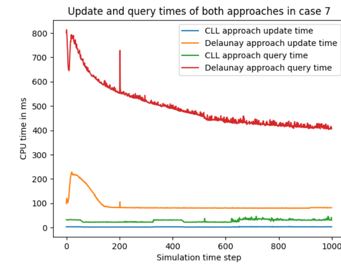
Figure 5.12: Falling column case 6 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 7.

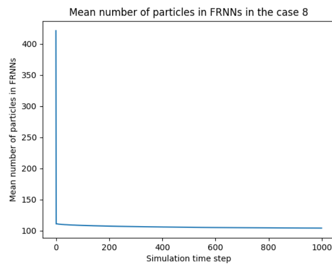


(b) Falling column total time execution trace scenario 7.

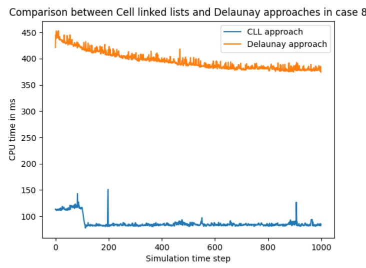


(c) Falling column update and query time execution trace in scenario 7.

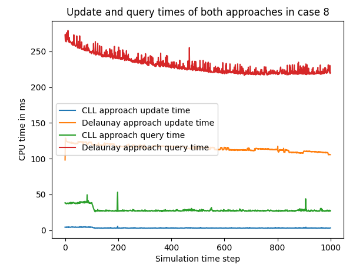
Figure 5.13: Falling column case 7 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 8.

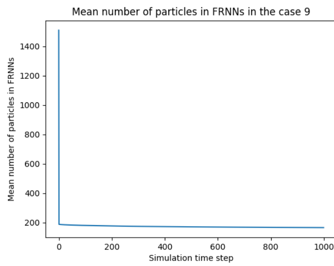


(b) Falling column total time execution trace scenario 8.

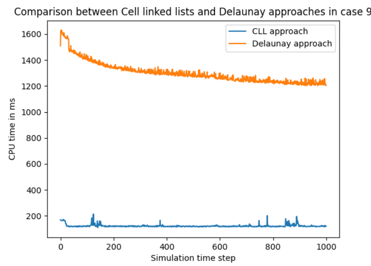


(c) Falling column update and query time execution trace in scenario 8.

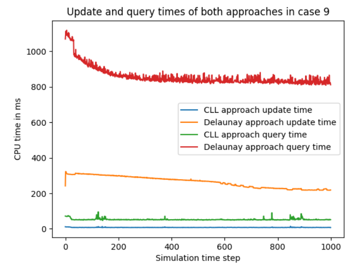
Figure 5.14: Falling column case 8 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 9.

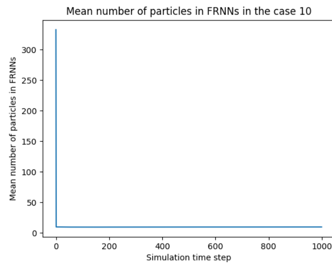


(b) Falling column total time execution trace scenario 9.

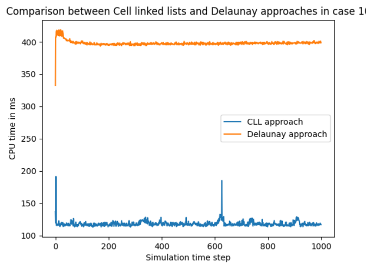


(c) Falling column update and query time execution trace in scenario 9.

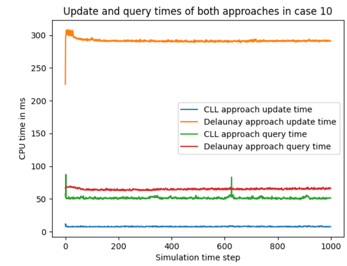
Figure 5.15: Falling column case 9 graphs.



(a) Falling column mean number of particles on the FRNN execution trace scenario 10.



(b) Falling column total time execution trace scenario 10.



(c) Falling column update and query time execution trace in scenario 10.

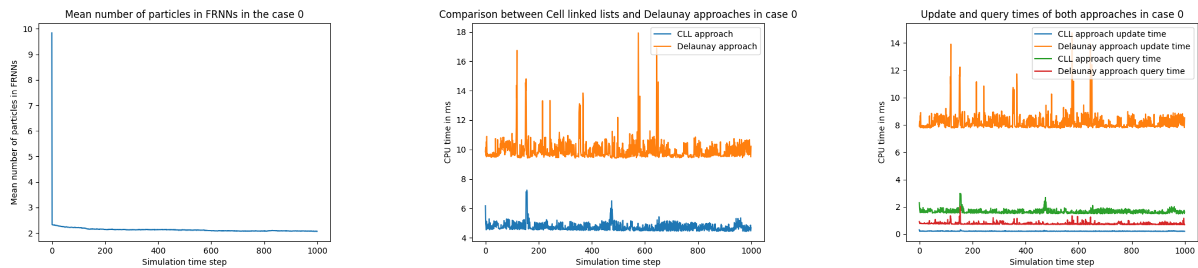
Figure 5.16: Falling column case 10 graphs.

5.4 Fluid simulation without gravity results

Table 5.3: Different settings of the initial random position fluid simulation without gravity.

FRNN radius	Simulation box size	Number of particles
200	4,800x4,800	3,000
300	4,800x4,800	3,000
400	4,800x4,800	3,000
400	6,000x6,000	10000
500	6,000x6,000	10,000
600	6,000x6,000	10,000
600	12,000x12,000	40,000
700	12,000x12,000	40,000
700	24,000x24,000	100,000
700	90,000x90,000	100,000

We call each of these fluid simulation settings without gravity 'cases'. These 11 cases, are very similar to the ones described before, and are also numbered from 0 to 10. We show the same graphs as in the previous scenario.

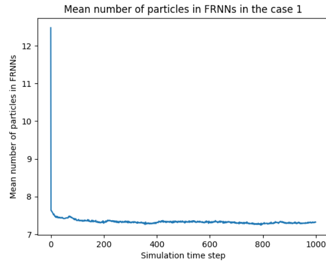


(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 0.

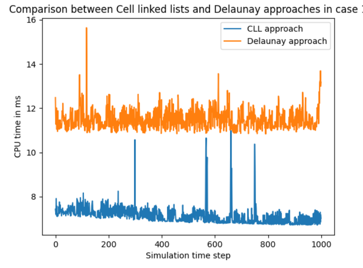
(b) Fluid simulation without gravity total time execution trace scenario 0.

(c) Fluid simulation without gravity update and query time execution trace in scenario 0.

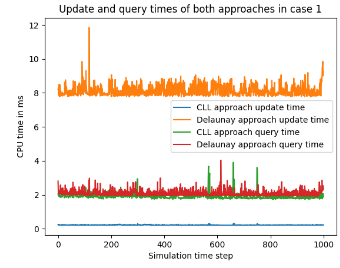
Figure 5.17: Fluid simulation without gravity case 0 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 1.

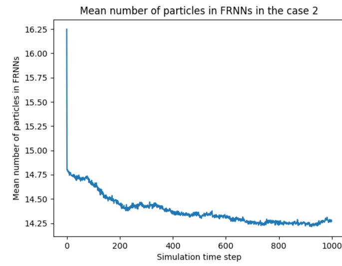


(b) Fluid simulation without gravity total time execution trace scenario 1.

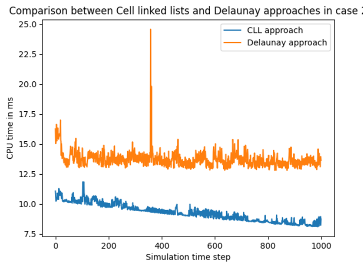


(c) Fluid simulation without gravity update and query time execution trace in scenario 1.

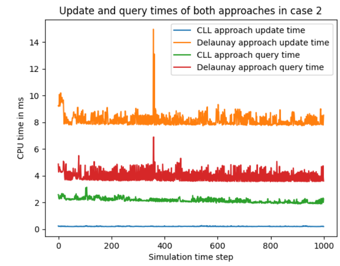
Figure 5.18: Fluid simulation without gravity case 1 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 2.

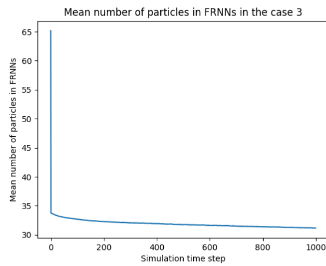


(b) Fluid simulation without gravity total time execution trace scenario 2.

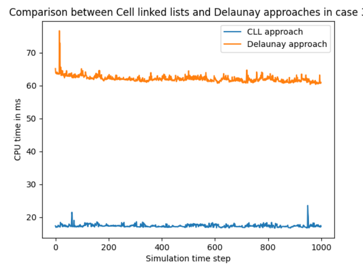


(c) Fluid simulation without gravity update and query time execution trace in scenario 2.

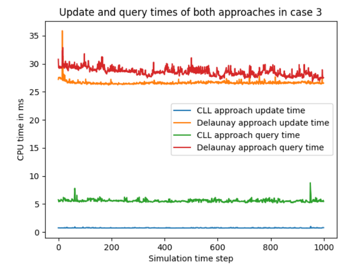
Figure 5.19: Fluid simulation without gravity case 2 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 3.

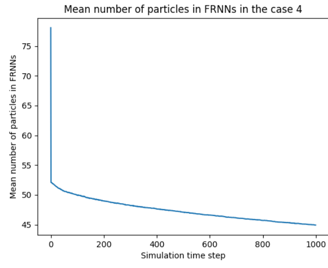


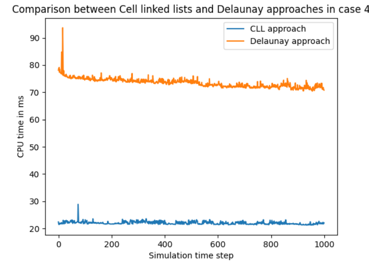
(b) Fluid simulation without gravity total time execution trace scenario 3.

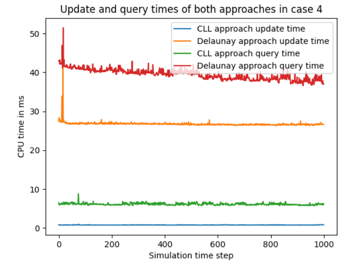


(c) Fluid simulation without gravity update and query time execution trace in scenario 3.

Figure 5.20: Fluid simulation without gravity case 3 graphs.





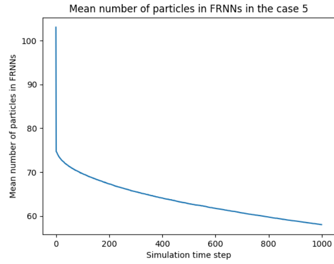


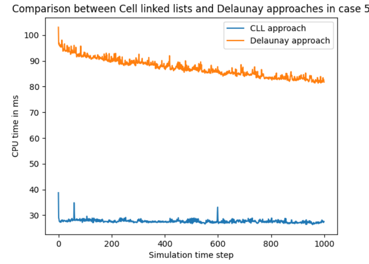
(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 4.

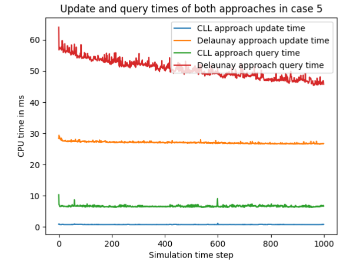
(b) Fluid simulation without gravity total time execution trace scenario 4.

(c) Fluid simulation without gravity update and query time execution trace in scenario 4.

Figure 5.21: Fluid simulation without gravity case 4 graphs.





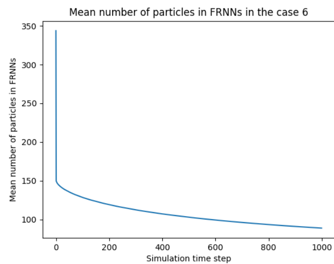


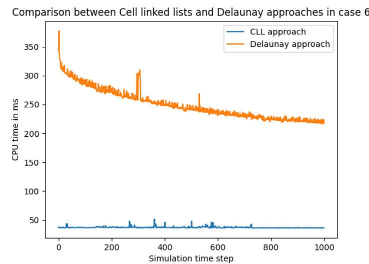
(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 5.

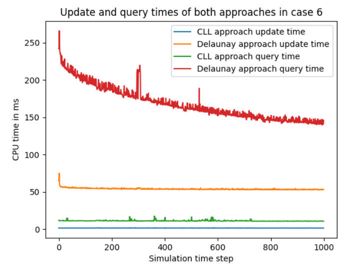
(b) Fluid simulation without gravity total time execution trace scenario 5.

(c) Fluid simulation without gravity update and query time execution trace in scenario 5.

Figure 5.22: Fluid simulation without gravity case 5 graphs.





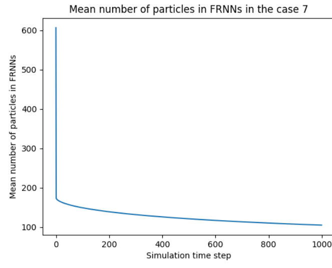


(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 6.

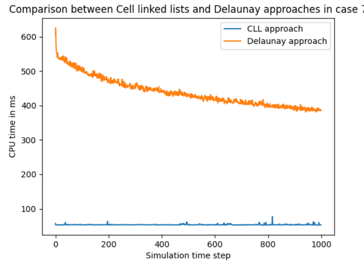
(b) Fluid simulation without gravity total time execution trace scenario 6.

(c) Fluid simulation without gravity update and query time execution trace in scenario 6.

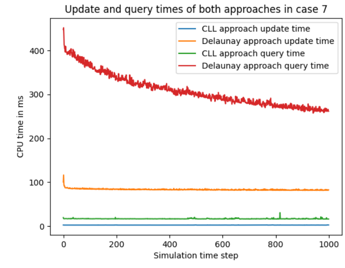
Figure 5.23: Fluid simulation without gravity case 6 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 7.

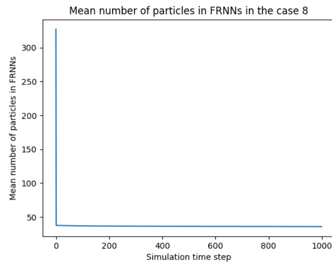


(b) Fluid simulation without gravity total time execution trace scenario 7.

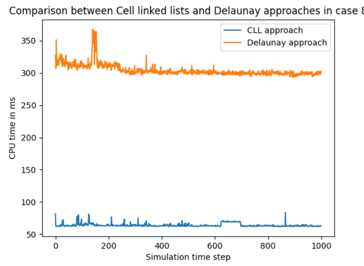


(c) Fluid simulation without gravity update and query time execution trace in scenario 7.

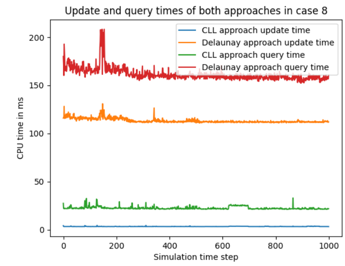
Figure 5.24: Fluid simulation without gravity case 7 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 8.

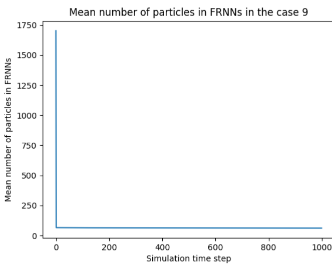


(b) Fluid simulation without gravity total time execution trace scenario 8.

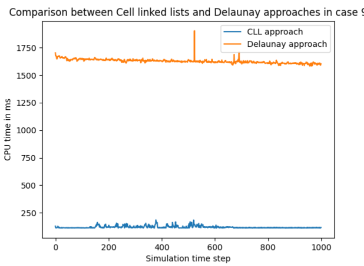


(c) Fluid simulation without gravity update and query time execution trace in scenario 8.

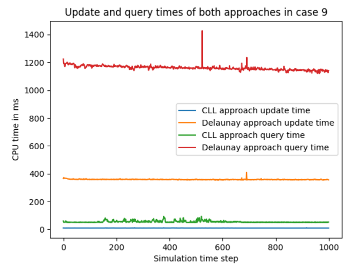
Figure 5.25: Fluid simulation without gravity case 8 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 9.

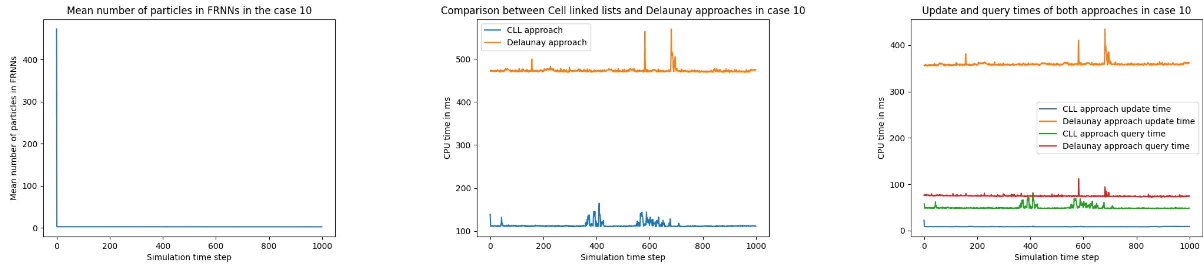


(b) Fluid simulation without gravity total time execution trace scenario 9.



(c) Fluid simulation without gravity update and query time execution trace in scenario 9.

Figure 5.26: Fluid simulation without gravity case 9 graphs.



(a) Fluid simulation without gravity mean number of particles on the FRNN execution trace scenario 10.

(b) Fluid simulation without gravity total time execution trace scenario 10.

(c) Fluid simulation without gravity update and query time execution trace in scenario 10.

Figure 5.27: Fluid simulation without gravity case 10 graphs.

5.5 Analysis

5.5.1 Shaking Points

We can see from all the figures in section 5.2, that it does not matter if some particles move very fast compared to others inside the simulation box, the time that it takes to update the proposed data structure is close to the time it takes to update a setting with particles moving slower. This follows from the fact that, as stated by De Castro et al. [9] and discussed in section 4.1, it takes approximately the same time to move a particle anywhere as long as the new position is outside the cavity around the particle.

While the Delaunay approach has a faster update time than the CLL one in the first 8 configurations (figures 5.3 and 5.4), in the case of the latest configurations (configurations 8 to 11) the opposite happens (figure 5.5). We claim that this is due the fact that there are more particles in cases 8 to 11 than in the other cases, while keeping the same simulation space. That means that in these cases the triangles are smaller than the ones in the first configurations, making a particle more likely to go outside its cavity in one time step, which is slower than moving inside its cavity.

We also see in the graphs that the difference between the time it takes to update both data structures is close to each other, or at least at the same execution time order, as expected from the analysis done in sections 3 and 4.

5.5.2 Fluid simulation

We will talk about situations that apply to the fluid simulation without gravity scenario and the falling column scenario.

If we see the graphs that compare the total running time of the algorithms, it is clear that the first 3 cases (figures 5.6, 5.7 and 5.8) behave very differently from the others (figures 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15 and 5.16). The first cases show that both approaches have very close running times, opposed to the latter ones, which show that the Delaunay based algorithm is around one order of magnitude slower. We claim this difference is because these

cases have very low number of particles, and we will no longer consider them to analyze the behavior of the algorithms.

Even though it has been shown in Chapters 3 and 4 that the Cell linked list and Delaunay approaches have the same theoretical running time bounds, we can see in the figures that the query times in the Delaunay approach are at least two magnitude factors over CLL query times in configurations 3 to 9. We claim that this happens because the running time of the Delaunay approach query has a big constant associated with it, which depends on the relation between the number of vertices and edges of a Delaunay triangulation (as mentioned in Chapter 4). Configurations with more particles might show close execution traces.

Opposed to the observations done in cases 3 to 9, we can see that in case 10 both algorithms running times are in the same order. This happens because the bigger simulation box, which makes the mean FRNN size smaller, meaning that the DFS performed in the proposed algorithm traverses fewer edges and vertices.

Chapter 6

Conclusions

The main objective of this thesis was to design, implement and test a novel algorithm to solve the Moving Points Fixed Radius Near neighbors MPFRNN problem based on Delaunay triangulations, and then explore a parallel version of this algorithm using GPUs.

We have presented how to construct a sequential algorithm to solve the MPFRNN problem. It is important to note that we solved the problem with particles going through each other. Additionally, we explored a possible parallelization of this algorithm in the GPU, and showed its difficulties. Finally, we compare the performance of this algorithm against a baseline algorithm based in Cell Linked Lists, in different simulation scenarios.

In this chapter, we discuss the main conclusions about the objectives mentioned at the beginning of this thesis, the contribution and relevance of it, and finally, what can be improved in the future.

6.1 Conclusions

The main contribution of this work is the presentation and evaluation of a novel sequential algorithm which solves the moving points fixed radius nearest neighbors problem.

The theoretical lower bound for the update of the underlying data structure of the proposed algorithm is better that the state of the art methods, and it keeps more topological information about the point set configuration in the plane, while also having a good query bound.

Through experimentation, we showed that this algorithm update time is comparable with the update time of the baseline algorithm, which solves the same problem in the CPU. This brought us to the conclusion that keeping the Delaunay triangulation of a set of moving points is about as expensive as building a cell linked list of them. On the other hand, query

times (the calculation of the FRNN given the data structure) are not as competitive as update times in most of the simulations we performed, even though the proposed algorithm has a better theoretical bound than the cell linked list approach. Experimentation also showed that the time taken to compute the FRNN in the proposed algorithm strongly depends on the mean size of the FRNN itself.

We also showed that, even though the proposed approach benefits in theory from particle sets that move slowly, because reinserting points might be faster, in practice it does not make a noticeable difference.

The variable that influences most on the calculation time of the FRNN in the proposed algorithm is the mean size of the particles' FRNN. On the contrary of what we thought, experimental data showed that the smaller the mean size of the particles' FRNN is, the smaller the query time in the proposed algorithm is, and closer it gets to the query time of the baseline algorithm. We think that this is because the constant accompanying the lower bound calculated is high, as it is dependent in the number of edges of the triangulation as well as the number of vertices in the triangulation (we calculate the FRNN doing a DFS starting from every point in the Delaunay triangulation).

We explored some of the problems that arise when parallelizing this algorithm in GPU. The construction of the Delaunay triangulation has been already parallelized in the GPU. Meanwhile, keeping it in the GPU as the particles move is a problem that has been addressed by Carter et al. [5]. There, they only used it to check collisions, and only did local transformations and flips. They did not solve a border case (figure 4.6), because physical conditions in their simulation did not allow it to happen in that case, and we explored a possible solution to it (figure 4.5). Finally, the parallel computation of the FRNN was also explored in section 4.2.2.

6.2 Future Work

Future work must consider parallelizing this algorithm in the GPU, as explored in section 4.2.

The proposed algorithm performed worse than the baseline algorithm. Still, both update times were similar, whilst the proposed algorithm stores more topological information about the particles. This fact suggests that it may be a good idea to combine the moving points Delaunay triangulation along other data structures used commonly in simulation scenarios, such as Verlet lists. Winkler et al. [33] showed that the combination of Verlet lists with cell linked lists works faster than cell linked lists alone. We think that keeping a Delaunay triangulation along Verlet lists might work even faster.

In this thesis we evaluated the performance of the algorithm in a simplified SPH fluid simulation and a set of moving particles bouncing in the walls, both allowing the particles

to go through each other. As the runtime of the FRNN calculation of the algorithm strongly depends on the way the set of particles behaves, it would be beneficial to test this algorithm within more simulation scenarios, such as granular material simulations [1], flocking simulations [12], etc.

The Delaunay triangulation of a set of points, as shown by Carter et al. [5], can be used to enforce the excluded volume property in simulation scenarios efficiently, because it connects all the vertices to its closest neighbor. Meanwhile, other data structures, such as cell linked lists, do not handle this issue, but take some time to calculate what are the colliding pairs of particles. This difference was not explored in this thesis.

Bibliography

[1] Nathan Bell, Yizhou Yu, and Peter J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 77–86, 2005.

[2] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. CRC press, 2010.

[3] Kevin Buchin. Constructing delaunay triangulations along space-filling curves. In *European Symposium on Algorithms*, pages 119–130. Springer, 2009.

[4] Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow-Seng Tan. A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 47–54, 2014.

[5] Francisco Carter, Nancy Hitschfeld, Cristóbal A. Navarro, and Rodrigo Soto. Gpu parallel simulation algorithm of brownian particles with excluded volume using delaunay triangulations. *Computer Physics Communications*, 229:148–161, 2018.

[6] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228, 2005.

[7] Microprocessor Standards Committee et al. 754-2019-ieee standard for floating-point arithmetic, 2019.

[8] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry*. Springer, 1997.

[9] Pedro Machado Manhães de Castro and Olivier Devillers. Delaunay triangulations for moving points. 2008.

[10] Olivier Devillers, Stefan Meiser, and Monique Teillaud. Fully dynamic delaunay triangulation in logarithmic expected time per operation. *Computational Geometry*, 2(2):55–80, 1992.

[11] Matthew T. Dickerson and Robert Scot Drysdale. Fixed-radius near neighbors search algorithms for points and segments. *Information Processing Letters*, 35(5):269–273, 1990.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[13] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007.

[14] Leonidas J. Guibas, Donald Ervin Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1-6):381–413, 1992.

[15] Leonidas J. Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM transactions on graphics (TOG)*, 4(2):74–123, 1985.

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

[17] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78, 2008.

[18] Charles L. Lawson. Transforming triangulations. *Discrete mathematics*, 3(4):365–372, 1972.

[19] Charles L. Lawson. Software for c1 surface interpolation. In *Mathematical software*, pages 161–194. Elsevier, 1977.

[20] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two-and three-dimensional delaunay triangulations. *Computational Geometry*, 12(1-2):63–83, 1999.

[21] David E. Muller and Franco P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[22] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. A gpu-based method for generating quasi-delaunay triangulations based on edge-flips. In *GRAPP/IVAPP*, 2013.

[23] NVIDIA Corporation. About cuda. `https://developer.nvidia.com/about-cuda`, 2021. [Online; Accessed on April 19 2021].

[24] NVIDIA Corporation. Cuda c programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2021. [Online; Accessed on April 19 2021].

[25] Joseph O'Rourke et al. *Computational geometry in C*. Cambridge university press, 1998.

[26] Franco P. Preparata and Michael Ian Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[27] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2d constrained delaunay triangulation using the gpu. *IEEE transactions on visualization and computer graphics*, 19(5):736–748, 2012.

[28] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, 2006.

[29] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[30] Hang Si. On monotone sequences of directed flips, triangulations of polyhedra, and structural properties of a directed flip graph. *arXiv preprint arXiv:1809.09701*, 2018.

[31] David Stevenson et al. Ieee 754-1985 ieee standard for binary floating-point arithmetic. *20pp, IEEE, July*, 1985.

[32] Volker Turau. Fixed-radius near neighbors search. *Information processing letters*, 39(4):201–203, 1991.

[33] Daniel Winkler, Massoud Rezavand, and Wolfgang Rauch. Neighbour lists for smoothed particle hydrodynamics on gpus. *Computer Physics Communications*, 225:140–148, 2018.

Appendix A

Code appendix

A.1 Cell Linked Lists code

Listing A.1: Cell Linked Lists implementation.

```
1  #include<algorithm>
2
3  template<class V>
4  struct CLL {
5      V* pos; // Buffer which has the values we have to sort according the keys
6      std::pair<int,int>* keys; // hash, index pair
7
8      size_t elem_num;
9      float e; // side of the cells
10     float height, width; // box dimentions
11     float x_offset, y_offset; // offset from the center of the simulation square
   ↪  (lower left point coordinates)
12     int xnum,ynum; // number of rows and columns of the cll
13
14     int *start_indices,*end_indices; // indices of the starting and ending
   ↪ position of every cell
15
16     CLL(size_t elem_num, V* pos, float height, float width, float x_offset,
   ↪ float y_offset, float e) : pos(pos), elem_num(elem_num), height(height), width
   ↪ (width), x_offset(x_offset), y_offset(y_offset), e(e) {
17         keys = new std::pair<int,int>[elem_num];
18         xnum = (int)(ceil(width/e))+1;
19         ynum = (int)(ceil(height/e))+1;
20         start_indices = new int[xnum*ynum];
21         end_indices = new int[xnum*ynum];
22     }
23     ~CLL(){
24         delete keys;
25         delete start_indices;
26         delete end_indices;
```

```
27         }
28         inline int calc_linear_hash(int i){
29             return xnum*floor((pos[2*i+1]-y_offset)/e) + floor((pos[2*i+0]-x_offset)
    ↪ /e);
30         }
31         void calc_keys(){
32             for(size_t i=0;i<elem_num;i++){
33                 if((pos[2*i+1]-y_offset) >= height){
34                     keys[i] = std::make_pair(-1,i);
35                     continue;
36                 }
37                 if((pos[2*i+1]-y_offset) < 0){
38                     keys[i] = std::make_pair(-1,i);
39                     continue;
40                 }
41                 if((pos[2*i+0]-x_offset) >= width){
42                     keys[i] = std::make_pair(-1,i);
43                     continue;
44                 }
45                 if((pos[2*i+0]-x_offset) < 0){
46                     keys[i] = std::make_pair(-1,i);
47                     continue;
48                 }
49                 keys[i] = std::make_pair(calc_linear_hash(i),i);
50             }
51         }
52         void update_stl(){
53             std::sort(keys,keys+elem_num);
54             for(size_t i=0;i<xnum*ynum;i++){
55                 start_indices[i] = -1;
56                 end_indices[i] = -1;
57             }
58             if(keys[0].first!=-1)
59             start_indices[keys[0].first] = 0;
60             for(size_t i=1;i<elem_num;i++){
61                 if(keys[i].first!=-1)
62                 if(keys[i-1].first!=keys[i].first)
63                     start_indices[keys[i].first]=i;
64             }
65             if(keys[elem_num-1].first!=-1)
66             end_indices[keys[elem_num-1].first] = elem_num-1;
67             for(size_t i=0;i<(elem_num-1);i++){
68                 if(keys[i].first!=-1)
69                 if(keys[i].first!=keys[i+1].first)
70                     end_indices[keys[i].first]=i;
71             }
72         }
73         std::vector<int> get_neighbors(int i){
74             std::vector<int> res = std::vector<int>();
75             int hs = calc_linear_hash(i);
```

```
76
77              if(hs<=-1)return res;
78              if(hs>=xnum*ynum) return res;
79
80              int x = hs%xnum;
81              int y = hs/xnum;
82
83              // Same cell
84              if(start_indices[hs]!=-1)
85              for(int j=start_indices[hs];j<=end_indices[hs];j++){
86                  if(keys[j].second!=i)res.push_back(keys[j].second);
87              }
88              // Right
89              if(x+1<xnum)
90              if(start_indices[hs+1]!=-1)
91              for(int j=start_indices[hs+1];j<=end_indices[hs+1];j++){
92                  res.push_back(keys[j].second);
93              }
94              // Right & Bottom
95              if((x+1<xnum) && (y-1>=0))
96              if(start_indices[hs+1-xnum]!=-1)
97              for(int j=start_indices[hs+1-xnum];j<=end_indices[hs+1-xnum];j++){
98                  res.push_back(keys[j].second);
99              }
100             // Bottom
101             if(y-1>=0)
102             if(start_indices[hs-xnum]!=-1)
103             for(int j=start_indices[hs-xnum];j<=end_indices[hs-xnum];j++){
104                 res.push_back(keys[j].second);
105             }
106             // Left && Bottom
107             if((x-1>=0) && (y-1>=0))
108             if(start_indices[hs-1-xnum]!=-1)
109             for(int j=start_indices[hs-1-xnum];j<=end_indices[hs-1-xnum];j++){
110                 res.push_back(keys[j].second);
111             }
112             // Left
113             if(x-1>=0)
114             if(start_indices[hs-1]!=-1)
115             for(int j=start_indices[hs-1];j<=end_indices[hs-1];j++){
116                 res.push_back(keys[j].second);
117             }
118             // Left & Top
119             if((x-1>=0) && (y+1<ynum))
120             if(start_indices[hs-1+xnum]!=-1)
121             for(int j=start_indices[hs-1+xnum];j<=end_indices[hs-1+xnum];j++){
122                 res.push_back(keys[j].second);
123             }
124             // Left & Top
125             if(y+1<ynum)
```

```
            if(start_indices[hs+xnum]!=-1)
            for(int j=start_indices[hs+xnum];j<=end_indices[hs+xnum];j++){
                res.push_back(keys[j].second);
            }
            // Right & Top
            if((x+1<xnum) && (y+1<ynum))
            if(start_indices[hs+1+xnum]!=-1)
            for(int j=start_indices[hs+1+xnum];j<=end_indices[hs+1+xnum];j++){
                res.push_back(keys[j].second);
            }
            return res;
        }
    };
```

A.2 Fluid simulation code

Listing A.2: Simulation step code.

```
    void step(REAL dt){
        for(int i=0;i<numP;i++){
            p[i] = 0;
            p_near[i] = 0;
        }

        for(int i=0;i<numP;i++) {
            auto neighbours = data_structure->getFRNN(i,h);
            for (int j: neighbours) {
                REAL q = hmod(t->vertices[i].pos - t->vertices[j].pos) / h;
                if (q < 1 && i!=j) {
                    p[i] += pow2(1 - q);
                    p_near[i] += pow3(1 - q);
                }
            }
            P[i] = k * (p[i] - p0);
            P_near[i] = k_near * p_near[i];
            dx[i] = Vec2{0, 0};
            for (int j: neighbours) {
                REAL q = hmod(t->vertices[i].pos - t->vertices[j].pos) / h;
                if (q < 1 && i!=j) {
                    Vec2 rij = (t->vertices[j].pos - t->vertices[i].pos);
                    normalize(rij);
                    Vec2 D = rij * (REAL)(P[i] * (1 - q) + P_near[i] * pow2(1 - q));
                    D /= 2.0;
                    dx[i] -= D;
                }
            }
        }
        for(int i=4;i<numP;i++)if(state_code[i]!=0){
            data_structure->movePoint(i,dx[i] + velocity[i]*dt);
```

```
            velocity[i] = dx[i];
            velocity[i] -= Vec2{0,g};
            velocity[i] /= dt;
        }
    }
```

In the previus code, the variable *data_ structure* is the data structure used to accelerate the computation of the FRNN. The buffer *state_ code* stores an integer that is 0 if the particle corresponds to a particle that does not move. The *REAL* type is a single floating point precision number. The function *hmod* is an inline function that calculates the modulus of a *Vec2*, which is a structure that represents a 2D vector.