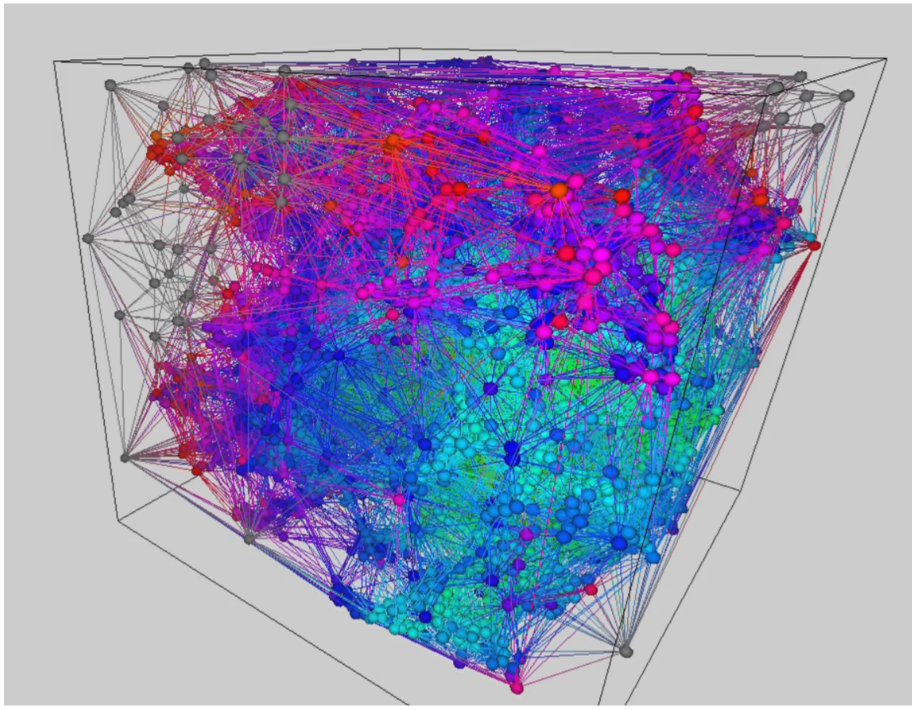
Internship report: Neighborhood research using Delaunay tetrahedralization

Hugo Bec, Claire Morin

July 2022



Introduction

Purpose of the project

This project consisted in experimenting methods of finding the nearest points using Delaunay tetrahedralization. This tetrahedralization consists in linking particles by tetrahedrons so that when we draw the circumscribed sphere of a tetrahedron, no other particle is included in it. This method allows to link directly each particle with their closest neighbors.

To test our algorithms, we created two simulations:

- A simulation of 3D pseudo-Brownian particle motion with diffusion limited aggregation.
- A 3D simulation of boids.

Objects are initialized in a finite space according to a uniform distribution and cannot leave this space. If a particle hits an edge of the space, it bounces off the wall. If a Boid exits the space, its velocity slowly redirects inward to keep a smooth motion.

The idea will be to search for each of the points, the set of other points having a distance smaller than a given radius using a Delaunay tetrahedralization.

These experiments allow to test their concrete efficiency sequentially and to make them parallelizable for a future adaptation on GPU.

Parameters

General

- DSTRUCTURE_VERBOSE: Enable/Disable execution details in the terminal
- TYPE_SIMULATION: Choose between the Diffusion-limited aggregation mode (0) or the Boids mode (1)
- NB_POINT: Choose the number of Particles / Boids
- SPEED_POINTS: Selects the speed of the Particles / Boids
- ATTRACT_RADIUS: Choose their radius of attraction, for the DLA it is preferable to initialize it to $2 \times SIZE\_OBJECTS$, for the Boids determines its radius of influence
- SIZE_OBJECTS: Selects the size of the 3D meshs
- TETRA_REFRESH_RATE: Determines the number of frames before recalculating the Delaunay tetrahedralization
- CAGE_DIM: Determines the size of the cage
- NB_INIT_FIXED_POINTS: Choose the number of fixed points at the start (for Boids must be set to 0)

Boids

- BOIDS_SEPARATION_DISTANCE: Determines the average distance between each Boid
- BOIDS_BOX_MARGIN: Determines at what distance from the edge the Boids initiate a turn to return to the cage
- BOIDS_COHESION_FACTOR : Determines the cohesion factor added to each frame
- BOIDS_ALIGNMENT_FACTOR: Determines the alignment factor of the Boids at each frame
- BOIDS_SEPARATION_FACTOR: Determines the Boid separation factor at each frame
- BOIDS_TURN_FACTOR: Determines the turn factor at the edge of the cage that a Boid makes each frame.

The libraries

For this project, we used the libraries :

- `tetgen` to calculate the Delaunay tetrahedralization;
- `OpenGL` to represent finite space, tetrahedralization and 3D particles in real time;
- `Assimp` to read .mesh files;
- `ImGUI` for the graphical interface.

The whole thing is executable from Visual Studio 2022.

1 Proceedings

1.1 Tetrahedralization

To realize the tetrahedralization in three dimensions, we used the library `tetgen` allowing, from a cloud of points to generate a Delaunay tetrahedralization under the format `tetgenio`. The idea is to read this data structure in order to create a visualization (See part 1.2 Visualization) but also to perform optimizations on the search for the closest points. To do this, we have created a class `DelaunayStructure`. In this class we generate the Delaunay tetrahedralization by calling the library `tetgen`, we display it, then we look for the closest points.

1.2 Visualization

Before developing optimization solutions, we wanted to be able to visualize the solutions in an intuitive way and in real time. To do so, we used a code skeleton `OpenGL` from M.MARIA that we adapted to display the particles, their tetrahedralization and the finite space in three dimensions. Several modes of visualization are available with the interface `ImGUI` or with the keys of the keyboard.

- ZQSDRF + mouse: camera
- +/- : visualize the next/previous point and its attractive points (depends on the radius of attraction)
- E : display the associated edges of the Delaunay tetrahedralization

- T : display all the points of the simulation
- P : pause/play the simulation
- O : displays only the following frame
- M : change between the visualization of the attractive points and the visualization of the fixed points
- B: enables/disables 3D visualization of points (depends on the size of a point)

We added a spherical visualization to the particles to better visualize the aggregation by limited diffusion. To do this we used an iso-sphere mesh (of precision 1, 2 and 3 generated on Blender) that we load in the program to display it with `OpenGL`. The idea was then to make a copy of the mesh structure for each particle to avoid reading the file several times. We initially wanted to use the instantiation technology of OpenGL to gain in display performance but we could not implement it due to lack of time.

2 Comparison of versions

2.1 Function compute_attract_by_double_radius

Knowing the maximum speed of the particles, it is possible to deduce the maximum distance that a particle can travel on a number N of frames with the following formula: $speed \times N \times 2$, times two because we consider that the two particles move away in opposite direction, this represents a finite area that we call the radius of Verlet. The idea is then to recompute a tetrahedralization only for all N frames instead of all frames, and to compute the Verlet radius as well as all particles included in this radius for each particle. The idea is then to compute the attractive particles of the next N frames using only the set of particles included in the Verlet radius.

This process avoids recalculating the tetrahedralization every frame and allows, for a reasonable particle speed and number N of frames, to gain in performance. A drawback however is that it creates "freezes" (slowdowns on a single frame) at regular intervals (at the time of recalculating the tetrahedralization as well as for browsing the graph and recording all the particles included in the current Verlet radius).

In reality, traversing a list of N particles takes less time than traversing N particles in a graph. Breadth-First Search has a higher complexity than simple array traversal.

2.2 Function compute_attract_by_flooding

Another method has been experimented, it consists, at each frame, to go through the tetrahedralization graph without recalculating it. The idea is to traverse the width of the graph starting from a point to a fixed depth. The same problem arises in relation to the cost of traversing the BFS and this method is slower than the previous one.

The older the tetrahedralization, the more it is necessary to go deeper and deeper into the tree to be sure to recover all the attractive points. It can happen that if we do not go far enough in the graph, some points are not found. Thus, to be sure to recover all the attractive points, it is necessary to choose a rather large depth and thus expensive in computation time (due to the BFS), especially when the tetrahedralization has not been updated for a long time.

We can see that with a small radius of attraction and a small refresh frame the double ray method performs better than the deep path method.

Chart comparison

For this graph we have 10 000 points with a speed of 0,1. The time calculation is done on 20 frames.

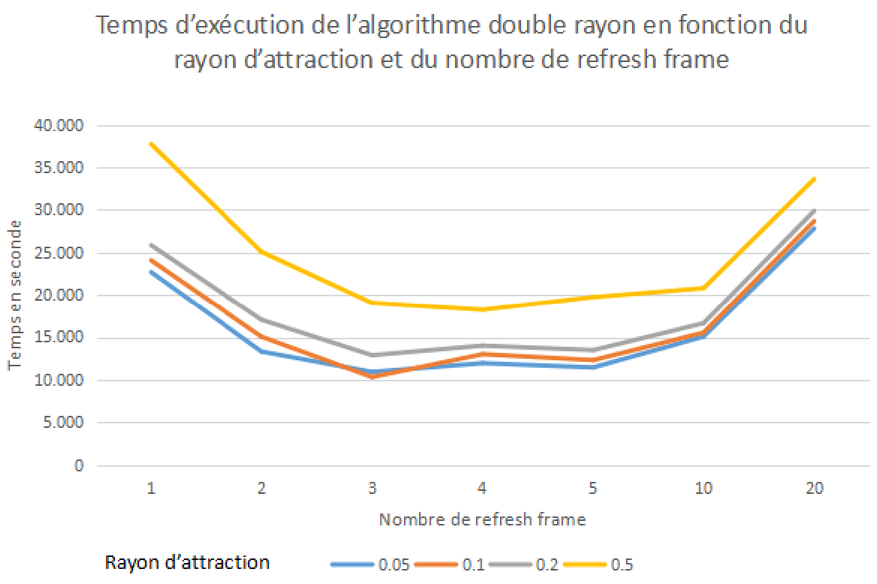


Figure 1: Chart of the function `compute_attract_by_double_radius` for 10,000 particles

With the help of this chart, we can see that the method `compute_attract_by_double_radius` is more efficient when the refresh frame is 3 or 4.

For this graph we have 1000 points with a speed of 0,01 and radii of 2,3,4 and 5. The time calculation is done on 100 frames.

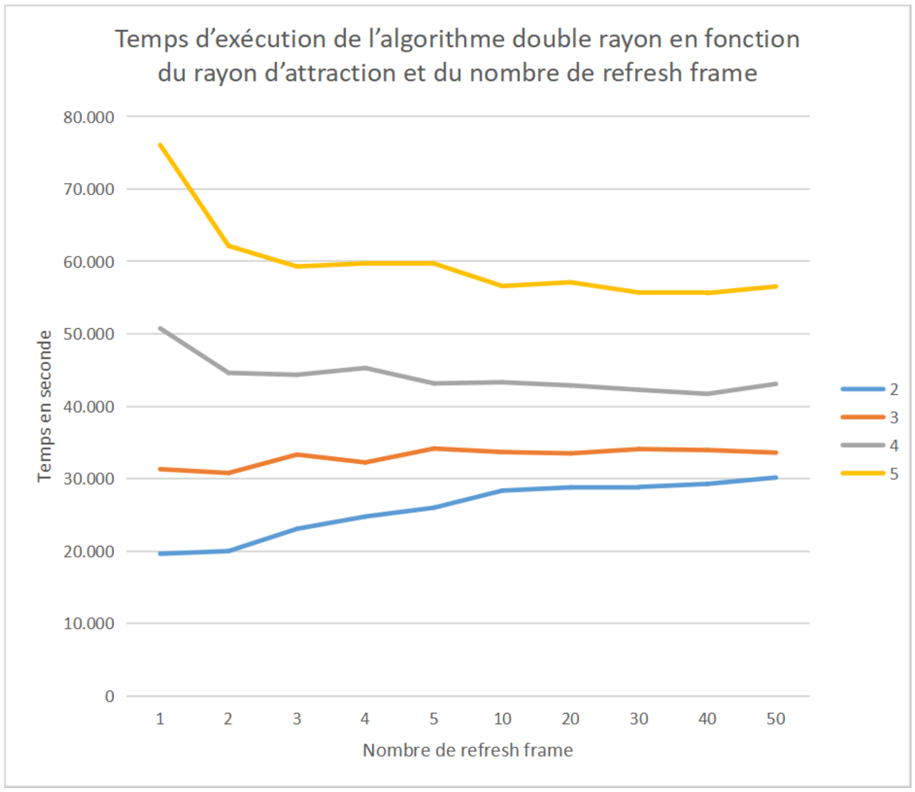


Figure 2: Chart of the function `compute_attract_by_double_radius` for 1,000 particles

We can see that the larger the radius the larger the refresh frame must be to be effective.

3 Algorithm "Diffusion Limited Aggregation" : `compute_diffusion_limited_aggregation`

To use this algorithm it is necessary that the number of fixed particles, `NB_INIT_FIXED_POINTS`, is different from 0.

The objective is that all particles are fixed. They are fixed when they enter the radius of attraction of a fixed particle.

To make the performance faster, we look for the attractive particles of the fixed particles as long as the number of fixed particles is less than half of the total number of particles. Otherwise we search for the attractive particles of the non-fixed particles. Thus we only search for at most half of the particles. To find the attractive particles we use the list
`_possible_futur_attract` with the method `compute_attract_by_double_radius`.

The coloring is done according to the number of frames during which the particle remained in movement. It is colored when it becomes fixed.

In order for the particles to bind when they touch, we put a radius of attraction that is twice the size of a particle, which is the radius of the particle.

We made a timelapse visible on Youtube (Timelapse of an aggregation)

4 Visual result of the algorithm `compute_diffusion_limited_aggregation`

The red particles are the oldest to have been fixed, then they are the colors: orange, yellow, green, blue, purple and pink. Then we start again this cycle of colors. We can observe a tree structure on the Figure 4.

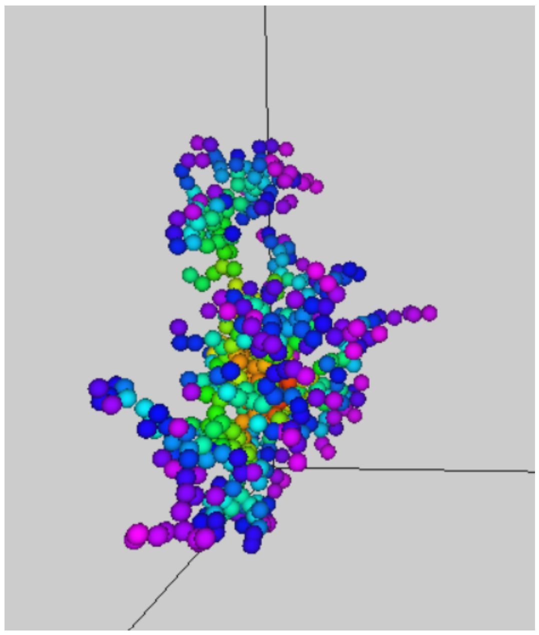


Figure 3: Colored aggregation 1 with 3000 points

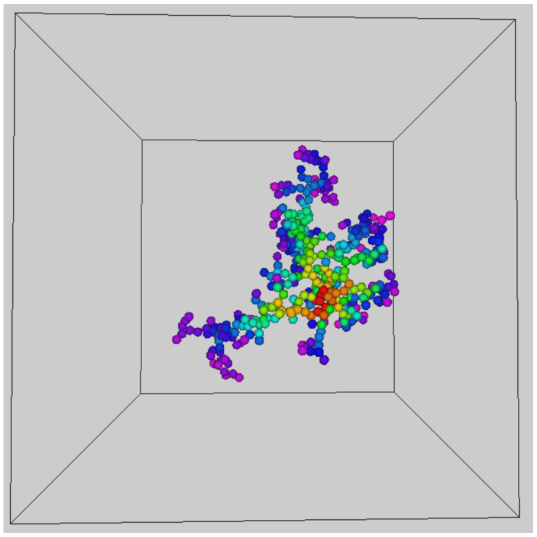


Figure 4: Colored aggregation 2 with 3000 points

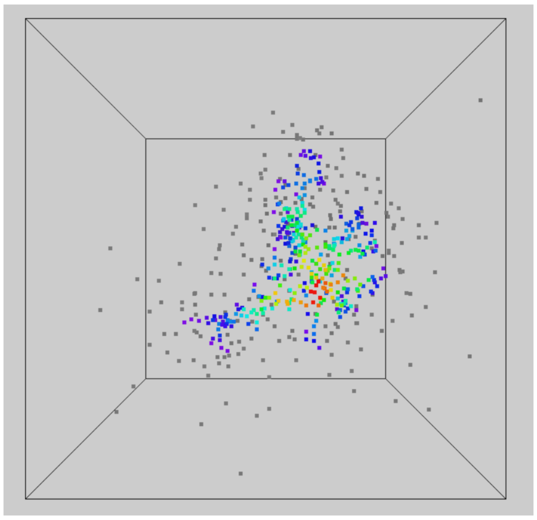


Figure 5: Colored aggregation 2 with 3000 points without meshes

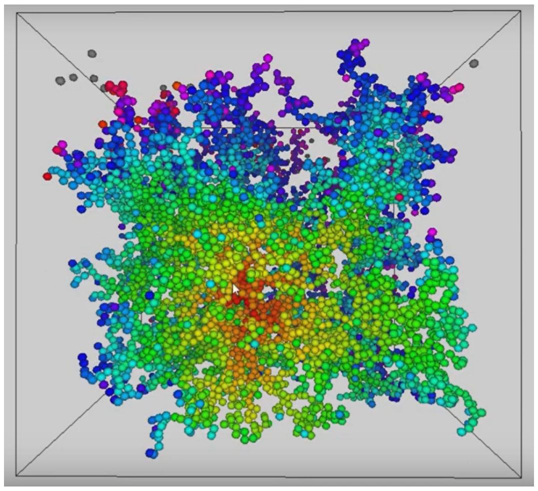


Figure 6: Final result colored aggregation with 10 000 points du (Timelapse of an aggregation)

5 3D Boids

To test our methods we have reproduced the behavior of Craig Reynolds' Boids (Reference website) configurable. A boid is the equivalent of a particle with just a different motion described by Craig Reynold's algorithm. Thus the advantage is to use `compute_attract_by_double_radius` to recover the neighbors of each Boid.

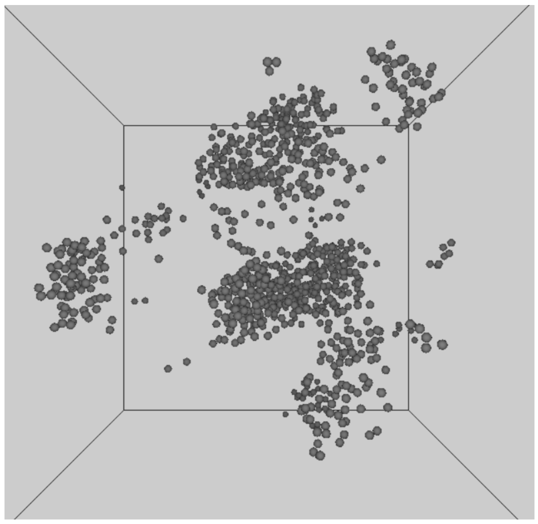


Figure 7: 3D Boids