MAINTAINING 2D DELAUNAY TRIANGULATIONS ON THE GPU FOR PROXIMITY QUERIES OF MOVING POINTS

Heinich Porro[1] Benoît Crespin[1] Nancy Hitschfeld[2] Cristóbal Navarro[3] Francisco Carter[2]

[1]*XLIM UMR CNRS 7252, University of Limoges, Limoges, France.*
heinich.porro@unilim.fr, benoit.crespin@unilim.fr
[2]*Department of Computer Science (DCC), Universidad de Chile, Santiago, Chile.*
nancy@dcc.cl, francisco.carter@ug.uchile.cl
[3]*Instituto de Informática, Universidad Austral de Chile, Chile.*
cristobal.navarro@uach.cl

ABSTRACT

In this paper we explore the problem of maintaining the Delaunay triangulation of moving 2D points on the GPU with discrete time steps, using only local transformations. We show that our Delaunay triangulation structure is efficient at answering proximity queries, such as closest neighbor or fixed-radius nearest neighbors problems. We also characterize the difficulties of updating the triangulation, and also the cases where it is possible to do it only through local operations.

Keywords: Delaunay triangulation, Parallel mesh data structure, Dynamic mesh modifications, Mesh untangling

1. INTRODUCTION

Delaunay Triangulations (or DT) are useful in numerous applications related to meshing as they are composed of good quality triangles that maximize their minimum angle, preventing the existence of very thin ones and favoring precision when doing numerical computation. For example, terrain generation techniques are frequently based on DT to obtain surface meshes from 2D elevation points or from noise functions. They are also used to construct meshes for finite element methods because of the guarantee on the angles and the speed of the algorithms. DT can also be found in finite volume methods to generate meshes on complex geometries, especially in fluid mechanics or deformable objects simulations. Algorithms for constructing DT rely on efficient data structures to store triangles and vertices. The neighbourhood relations stored in such structures can then be used to implement FEM simulations or proximity queries in particle based simulations.

In this work we study how efficient is a DT as a data structure for answering parallel proximity queries efficiently. More in detail, we are interested in the parallel GPU computation of the DT for sets of points whose position evolves over time. In this case it is generally more efficient to try to fix the inconsistencies of the mesh dynamically during the simulation rather than reconstruct the triangulation entirely at each time step, especially for very large sets of points. The problem to be solved in this case is that of mesh untangling, since the displacement of the points over time generally induces situations in which the mesh presents self-intersections and no longer satisfies the Delaunay condition.

Below we present a quick summary of the various approaches for dynamically managing DT on the GPU is presented, along with more details on specific GPU data structures. The structure we have created and the fundamental procedures required to maintain the DT are covered next, followed by a discussion on proximity queries.

2. PREVIOUS WORKS

Computing Delaunay triangulations in parallel
There is a vast amount of work in different algorithms to build DT in parallel. Divide-and-conquer approaches [1, 2] separate the work in different threads and then merge these partial solutions in a final step. This has been done in a variety of parallel models: many-cores, GPU, multi-GPU, etc. In flip-based approaches, an initial mesh is built without explicitly satisfying Delaunay conditions, before the topology is modified through local transformations implemented on the GPU to get the final DT [3, 4].

The method designed by Navarro et al. [4] stores the mesh in neither counter clock-wise (CCW) order nor clock-wise (CW) order. It checks in parallel, for every edge, if it fulfils the Delaunay condition. If the edge doesn't fulfil the Delaunay condition, it may be flipped if the thread gets the ownership of both triangles that share it. Finally, they perform a final step to fix inconsistencies in the data structure. These three steps are repeated in sequence until the triangulation is legalized. Also, the data structure used in this work doesn't allow building the graph related to the triangulation efficiently, as it stores only the necessary information to check the Delaunay condition, and flip the edges. Another approach is the GPU computation of a discrete Voronoi diagram [5], which can in turn be converted into a valid DT.

Maintaining triangulations without rebuilding
Filtering methods to maintain triangulations have been studied and experimentally tested [6, 7]. These methods rely on two phases : filtering, then displacing the vertices. Filtering consists in defining a safe region around each vertex, computed through geometric tests on its neighborhood. If a point ends up inside the region after the perturbation, then no further action is required. Otherwise, the point is removed from the triangulation and reinserted afterwards. Even though these methods are efficient, their parallelization imposes additional difficulties related to the order of displacement of the points and the computation of the safe regions.

Untangling triangulations instead of deleting and reinserting
Carter et al. [8] built an algorithm that maintains a triangulation under small movements in the GPU in order to speedup NNS in a colloids simulation. After each time-step of the simulation, they fix the triangulation whenever the points go through one edge (so the triangle gets invalid). Afterwards they utilize Navarro et al. [4] approach to legalize the triangulation. This algorithm doesn't handle movements further than one edge crossing, and uses the same data structure as Navarro et al. [4]. Shewchuk [9] describes an algorithm that updates a triangulation by taking decisions through only local geometric tests, even when triangles get inverted after vertex displacements. In order to do that, it performs flips, vertex insertion, vertex deletion and vertex displacement. Another approach to fix the triangulation after displacement is the one described by Agarwal et al. [10], which defines and identifies regions in the triangulation composed of inverted triangles. A re-triangulation is applied after fixing those cases.

Mesh representations on the GPU
Matrix based representations of static meshes [11, 12] have been tailored to efficiently retrieve information about the neighborhood of the elements in the mesh on the GPU. These representations do not allow efficient modifications of the elements, and Mahmoud et al. [11] also show that an implicit half edge representations on the GPU is more efficient for the same purposes.

Proximity queries
Many GPU implementations of proximity queries are already available in existing software. The most used techniques rely on cell sorting [13, 14]. This techniques sort the particles according to where they are on a regular grid, and then use adjacent cells to explore neighborhood. Such a method is used for example in [14] for molecular surface generation, as well as in particle-based fluid simulations in computer graphics [14]. Other important method to solve proximity queries is based on hardware ray-tracing acceleration [15], which reports a significant performance over grid-based methods.

3. OUR METHOD

3.1 Data structure and basic operations

We use an indexed half edge data structure to conduct flip operations while also verifying the local geometrical requirements of each edge. Our data structure is very similar to the "parallel directed edges" described in [11]. The most important difference is that we store the index of the vertex opposed to each half edge to efficiently check the geometric conditions of the edge.

Listing 1: Data structure memory layout

```
struct half_edge {
/* Indices to the vertex position buffer */
```

```
 int v;    // Vertex at the beginning of the half-edge
 int op;   // Vertex oposite to the half-edge
/*************************************/
 int t;    // Triangle index that contains this half-edge
 int next; // Index to the next half-edge in this triangle
};

struct triangulation {
 double v[numVertices*2];    // Vertex positions
 int v_to_e[numVertices];    // Vertex to edges
 uint t_to_he[numTriangles]; // Triangles to half_edges
     indices
 half_edge he[numEdges*2];   // Half-edges
};
```

Along with the vertices coordinates, the triangulation is stored as three data arrays as seen in Listing 1. This representation is practical primarily because it enables retrieval of the two triangles for each fully indexed edge with only one random access to global memory per edge, and of the geometrical information of the points in these triangles in only four more random reads to global memory (in order to check the geometry of the triangles that share that edge). It also makes the 1-to-1 mapping from threads to edges easy, as we have to assign only 1 thread per every 2 half-edges.

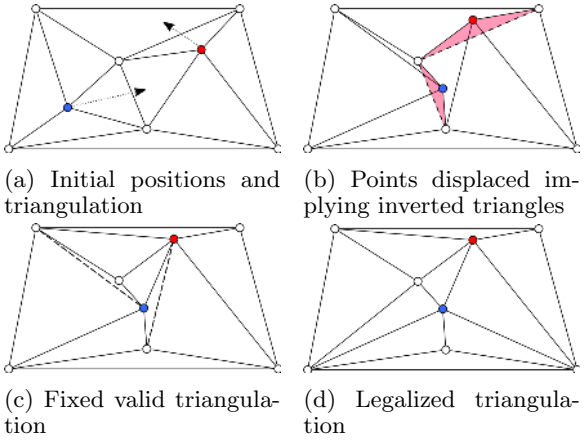
The **flipping operation** using this data structure takes only 5 reads and 5 writes to global memory (all the edges of the two triangles we are flipping), and 2 global atomic operations. Opposed to the data structure used by Navarro et al. [4] and Carter et al. [8], with this data layout we don't need to repair any inconsistent data after performing the flips.

3.2 Updating the triangulation

Starting with a fixed set of particles S, we assume this set undergoes a series of displacements d_i with $i \in \{0, n\}$ and n the number of timesteps in the simulation; particles positions at timestep i are noted S_i. We start the simulation with a triangulation T_0 built offline from the initial set S_0, and modify it to meet the Delaunay condition on the GPU as in [4]. The goal now is to keep the triangulation Delaunay for each timestep $i \in \{1, n\}$ from the previous triangulation T_{i-1} and displacements d_{i-1}.

In our approach the problem is divided in two steps: first, we check if T_{i+1} is still a valid triangulation for the positions S_{i+1}. If it's not, we fix it only with local geometric tests and flips, just like [8]. In the second step, we flip as many edges as necessary in order to ensure the Delaunay condition of the edges, just as in [4]. We call the first part of the process **fixing** (described in the next section), and the second **legalizing**.

In the case where it is not possible to fix the triangulation after displacements d_i, we simply divide the displacements in smaller (and fixable) steps. An example of the whole process is depicted in Fig. 1.



(a) Initial positions and triangulation

(b) Points displaced implying inverted triangles

(c) Fixed valid triangulation

(d) Legalized triangulation

Figure 1: Fixing and legalizing a triangulation

3.3 Fixing the triangulation

In the following we take some definitions from Shewchuk [9], slightly adapted to simplify the exposition of the ideas in this research note.

Assuming that T_i is a CCW oriented mesh, once we move the points to S_{i+1} and keep the connectivity of T_i, some triangles become CW oriented and can be called **inverted triangles**. Let's note first that an inverted triangle could be part of a larger group of inverted triangles that all share common edges, also called *inverted triangle zone* in [10]. This zones are delimited by edges shared between an inverted triangle and a non-inverted triangle. We will call those edges **creased edges**, and the ones shared by two inverted triangles will be called **inverted edges**. Finally, if an edge is shared by two non-inverted triangles we call it an **upright edge**.

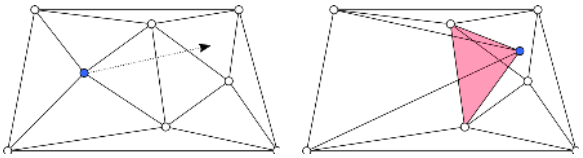
It is easy to see that if there are no inverted triangles nor creased edges after the displacement, no other action is required in this step because the triangulation is still valid and will be legalized afterwards.

As stated in [8], if there are points going through only one edge, we can fix the triangulation by simply flipping the crossed edge. We can also fix an inverted triangle whenever it has a creased edge and the vertex "coming" from that edge is inside the triangle. We implement this check by testing if another edge has both of its opposite vertices in opposite sides of the plane defined by the current edge. An example of this process is shown on Figs 1b and 1c.

The real problems arise if a point goes through two or more edges, i.e. if a creased edge is shared by an inverted triangle which is not contained by an non-inverted one or the opposite. This is illustrated in Fig. 2: In this case flipping one of the creased edges -the three edges of the inverted triangle- does not guarantee

Table 1: Performance results for RTX 3090 and RTX A3000 GPUs for a diffusion limited aggregation simulation, comparing our Delaunay approach with a grid-based structure [14]. Timings are averaged over 1K timesteps.

Particles	Scale	GPU	Delaunay			Grid		
			Mesh update	NN	Total (ms)	Construction	NN	Total (ms)
100K	10K	RTX 3090	0.65	0.06	0.71	0.56	0.01	0.57
		RTX A3000	0.67	0.06	0.73	0.64	0.01	0.65
1M	10K	RTX 3090	1.71	0.13	1.84	0.69	0.04	0.73
		RTX A3000	3.59	0.26	3.85	1.03	0.02	1.05



(a) Initial position, initial triangulation

(b) Point displaced, a triangle got inverted

Figure 2: A point going through two edges. None of the creased edges is shared by a triangle that contains the other.

that the triangulation gets fixed.

We characterize this as the difficult case of fixing the tangled triangulation, as it is still an open question whether or not we can solve this in a local and efficient way using only flips. It is important to note that Shewchuk [9] handles this case by adding, removing and/or displacing vertices to untangle the triangulation and fix the topological information before going back to the original geometrical positions stored in S_i.

4. RESULTS AND DISCUSSION

From our data structure storing a Delaunay triangulation, it is easy to compute the associated graph traversing the neighborhood of each vertex, thus allowing to compute the nearest neighbor (NN) of any vertex since they are both connected by an edge of the graph.

Table 1 shows how well our algorithm implemented with CUDA performs in a simple Diffusion-Limited Aggregation simulation [16], by comparing against a grid-based approach [14]. We made sure that the displacements in this simulation are small enough so we can always fix the triangulation, by keeping a displacement radius 100M times smaller than the whole simulation box.

Even though our method is efficient at answering proximity queries, it is not as fast as a grid-based method. However, we store a lot of information on the connectivity of points and a complete triangulation of the points during the simulation, which we believe can be beneficial in other applications. One of them is to retrieve the vertices within a certain distance to a given vertex, also called fixed-radius near neighbors (FRNN). For this we can apply a Breadth First Search (BFS) algorithm in the graph, using a naive approach where each thread on the GPU has its own queue of non-visited vertices. In order to check whether or not a vertex has been visited before, we search linearly in the set of already discovered neighbors. This linear pass is currently the bottleneck of our algorithm and needs to be improved before we can make comparisons with state-of-the-art methods.

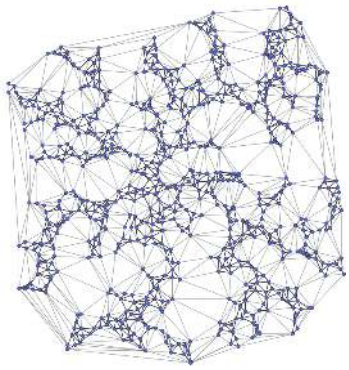


Figure 3: Depiction of the Delaunay triangulation of the final state of diffusion limited aggregation with 1K points.

5. CONCLUSION AND FUTURE WORK

The approach presented in this paper is slightly behind state-of-the-art techniques, still we believe it is a valuable method to update triangulations in slow moving scenarios, and a starting point towards a more robust method. It would be interesting to evaluate the performances in other simulation scenarios, because updating the mesh depends heavily on how fast the points move. This would require to define metrics, to estimate which simulations could benefit the most from our approach.

One possible way of improvement is the fact that a sequence of flips that can untangle a triangulation should always exist, as stated by Agarwal et al. [10]. It would be interesting to find this sequence using only local geometric tests, or to prove that it is indeed impossible in

some situations. Another future work concerns the Delaunay graph, which could be enhanced to make proximity queries faster. The Voronoi diagram could be used alternatively because it has a fixed vertex degree of 3, thus decreasing thread divergence. Finally, we could also try to improve our naive BFS algorithm by using a concurrent approach to solve FRNN queries. This is an interesting ongoing research topic in the GPU community, e.g. Liu et al. [17] address a very similar problem, and we could use or even improve their approach to solve the problem. In addition, we believe our method can be naturally extended to use constrained Delaunay triangulations to simulate particle movement a non rectangular domain.

References

[1] Chen M.B. "A Divide-and-Conquer Algorithm of Delaunay Triangulation with GPGPU." *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*, pp. 175–177. 2012

[2] Marot C., Pellerin J., Remacle J.F. "One machine, one minute, three billion tetrahedra." *International Journal for Numerical Methods in Engineering*, vol. 117, no. 9, 967–990, 2019

[3] Cao T.T., Nanjappa A., Gao M., Tan T.S. "A GPU accelerated algorithm for 3D Delaunay triangulation." *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 47–54. 2014

[4] Navarro C., Hitschfeld-Kahler N., Scheihing E. "A parallel gpu-based algorithm for delaunay edge-flips." *The 27th European Workshop on Computational Geometry, EuroCG*, vol. 11. 2011

[5] Qi M., Cao T.T., Tan T.S. "Computing 2D constrained Delaunay triangulation using the GPU." *IEEE transactions on visualization and computer graphics*, vol. 19, no. 5, 736–748, 2012

[6] Manhães de Castro P.M., Tournois J., Alliez P., Devillers O. "Filtering relocations on a Delaunay triangulation." *Computer Graphics Forum*, vol. 28, pp. 1465–1474. Wiley Online Library, 2009

[7] Zhou Y., Sun F., Wang W., Wang J., Zhang C. "Fast Updating of Delaunay Triangulation of Moving Points by Bi-cell Filtering." *Computer Graphics Forum*, vol. 29, pp. 2233–2242. Wiley Online Library, 2010

[8] Carter F., Hitschfeld N., Navarro C.A., Soto R. "GPU parallel simulation algorithm of Brownian particles with excluded volume using Delaunay triangulations." *Computer Physics Communications*, vol. 229, 148–161, 2018

[9] Shewchuk J.R. "Untangling Triangulations." *Unpublished paper*, p. 10, 2006

[10] Agarwal P.K., Sadri B., Yu H. "Untangling triangulations through local explorations." *Proceedings of the twenty-fourth annual symposium on Computational geometry*, pp. 288–297. 2008

[11] Mahmoud A.H., Porumbescu S.D., Owens J.D. "RXMesh: a GPU mesh data structure." *ACM Transactions on Graphics (TOG)*, vol. 40, no. 4, 1–16, 2021

[12] Yu C., Xu Y., Kuang Y., Hu Y., Liu T. "MeshTaichi: A Compiler for Efficient Mesh-based Operations." vol. 41, no. 6, 18, 2022

[13] Green S. "Particle simulation using cuda." *NVIDIA whitepaper*, vol. 6, 121–128, 2010

[14] Hoetzlein R.C. "Fast fixed-radius nearest neighbors: interactive million-particle fluids." *GPU Technology Conference*, vol. 18, p. 2. 2014

[15] Zhu Y. "RTNN: accelerating neighbor search using hardware ray tracing." *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 76–89. 2022

[16] Spicher A., Fatès N.A., Simonin O. "From Reactive Multi-Agent models to Cellular Automata - Illustration on a Diffusion-Limited Aggregation model." Springer, editor, *1st International Conference on Agents and Artificial Intelligence*. Portugal, Jan. 2009

[17] Liu H., Huang H.H., Hu Y. "ibfs: Concurrent breadth-first search on gpus." *Proceedings of the 2016 International Conference on Management of Data*, pp. 403–416. 2016