A GPU-based Method for Generating quasi-Delaunay Triangulations based on Edge-flips

Cristobal A. Navarro[1], Nancy Hitschfeld-Kahler[1] and Eliana Scheihing[2]
[1]*Department of Computer Science, FCFM, Universidad de Chile, Santiago, Chile*
[2]*Instituto de Informática, Universidad Austral de Chile, Valdivia, Chile*

Keywords: Delaunay Triangulations, Edge-flip Technique, Parallel Realtime Applications, CUDA, OpenGL, GPGPU.

Abstract: The Delaunay edge-flip technique is a practical method for transforming any existing triangular mesh S into a mesh $T(S)$ that satisfies the Delaunay condition. In this paper we present an iterative GPU-based method capable of improving triangulations under the Delaunay criteria. This method is based on the edge-flip technique and its implementation is fully integrable with the OpenGL rendering pipeline. Since the algorithm uses an ε value to handle co-circular or close to co-circular point configurations, we can not guarantee that all triangles fulfill the Delaunay condition. However, we have compared the triangulations generated by our method with the ones generated by the Triangle software and by the CGAL library and we obtained less than 0.05% different triangles. Based on our experimental results, we report speedups from $14\times$ to $50\times$ against Lawson's sequential algorithm and of approximately $3\times$ against the $O(nlog\ n)$ CGAL's and Triangle's constructive algorithms while processing bad quality triangulations.

1 INTRODUCTION

The Delaunay triangulation T of a point set P is the triangulation that maximizes the smallest angle over all triangulations of P. Numeric computations on this kind of triangulation is known to be more precise than in the other ones (De Berg, 2000). Good quality meshes are needed in many applications such as scientific simulations, terrain rendering, video-games and medical 3D reconstruction, among others.

Delaunay triangulations can be achieved in two ways: (a) by creating them from a PSLG (Planar straight linear graph), or (b) by transforming an already existing triangulation into one that satisfies the Delaunay condition. In general, in the case (a), a Delaunay triangulation is generated for the set of points of the PSLG. The segments (boundary edges) are then inserted to generate either a constrained Delaunay triangulation or a conforming Delaunay triangulation (Shewchuk, 1996). Case (b) assumes that a triangulation is given as input and the mesh needs to be transformed into a Delaunay mesh. A known technique for making this transformation is to flip the edges that do not satisfy the Delaunay condition. The edge-flip technique was first introduced by Lawson (Lawson, 1972) and the proposed sequential algorithm has a worst-case complexity $O(n^2)$, where n is the number of points of the triangulation (Fortune, 1993; Edelsbrunner, 2001).

Real-time applications cannot make use of sequential algorithms when handling meshes close to a million triangles. To achieve faster computations, parallel solutions are needed. In recent years, GPU computing has become an important research area for parallel computing due to its high performance and low cost. Several applications that require geometric modeling and visualization benefits strongly from the use of a GPU. In particular, the generation of Delaunay triangulations with a fast GPU-based method is today a topic of research.

The main contribution of this paper is the design and implementation of an iterative GPU-based algorithm that generates quasi-Delaunay triangulations starting from any existing triangulation. The algorithm maps threads to edges. Each thread is responsible for checking one edge Delaunay condition, doing one edge-flip and updating one edge data inconsistency if necessary. The performance and the quality of the generated meshes is compared with two well known and efficient sequential constrained Delaunay algorithms: the algorithm inside the software Triangle (Shewchuk, 1996) and the algorithm available in the CGAL library (CGAL, 2012). As test examples we used bad quality triangulations (with

Navarro C., Hitschfeld-Kahler N. and Scheihing E..
A GPU-based Method for Generating quasi-Delaunay Triangulations based on Edge-flips.
DOI: 10.5220/0004281900270034
In *Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications* (GRAPP-2013), pages 27-34
ISBN: 978-989-8565-46-4
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

minimum angles close to 0), with mesh sizes ranging from 100 thousand up to 5 millions points. We are not using exact predicates nor floating point filters because these techniques can not be efficiently implemented on GPU architectures without sacrificing performance. These techniques would require adding if-else conditionals and handle irregular-data access patterns. That is why, in favor of speed, some results are quasi-Delaunay triangulations and not fully Delaunay triangulations.

In principle, the comparison of a transformation algorithm that generates quasi-Delaunay triangulations, as the proposed in this paper, with respect to constructive ones that generate exact Delaunay triangulations such as CGAL and Triangle can seem unfair because they solve different problems. However, since the complexity of a constructive algorithm is $O(nlog\ n)$ and the Lawson algorithm is $O(n^2)$, many times it is preferred to build the Delaunay mesh from scratch instead of improving an existing one. In some way, this research is intended to show that a parallel method based on edge-flips can become fast and useful in practice for applications that not require exact Delaunay meshes.

The paper is organized as follows: Section 2 presents our data structures and how they are compatible with OpenGL. Sections 3 and 4 cover the algorithm and implementation details. Section 5 shows experimental results with 2D random inputs and 3D surface triangulations. Section 6 describes some related work and the similarities and differences with other GPU-based approaches. Finally, section 7 concludes our work.

A preliminary and short version of this paper was presented at the EuroCG11 workshop (Navarro et al., 2011).

2 DATA STRUCTURES

Proper data structures have been defined to efficiently represent a triangulation on the GPU. This representation is inspired by the Dynamic Render Mesh (Tobler and Maierhofer, 2006). Figure 1 illustrates the three main components: Vertices, Triangles and Edges.

Vertices are represented with a one-dimensional array in the same way as the OpenGL VBO (Vertex buffer object). Each position is of the type (x,y) or (x,y,z) depending on the used spatial dimension. The *Triangles* array is a set of indices to the *Vertices* array. For each three consecutive indices, a triangle is defined. Each edge of the *Edges* array contains a pair of indices v_1, v_2 to the *Vertices* array and two pairs of indices $t_a = \{t_{a_1}, t_{a_2}\}$ and $t_b = \{t_{b_1}, t_{b_2}\}$ to the *Triangles* array (for boundary edges, t_b remains unused).

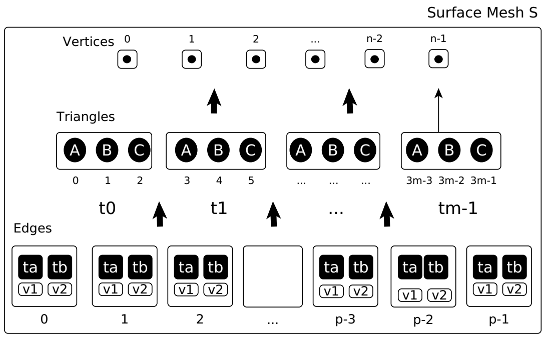


Figure 1: Data structures for mesh rendering/processing.

This way, an edge can know its endpoint indices directly through v_1, v_2 or indirectly via the pairs $\{Triangles[t_{a_1}], Triangles[t_{a_2}]\}$ and $\{Triangles[t_{b_1}], Triangles[t_{b_2}]\}$. This redundant information becomes useful for checking neighborhood consistency after each flip. In addition, indices to the opposite vertices per edge are stored in the *Opposites* array in order to speed up angle computations (boundary edges have only one opposite vertex). This data model can be naturally implemented and integrated with the OpenGL API and CUDA (Nvidia, 2011) or OpenCL. It is important to mention that for rendering, only the *Vertices* and *Triangles* arrays are accessed by the graphics API; the *Edges* and the *Opposites* arrays are for efficiently accessing neighbor information. All mentioned arrays use $\Theta(n)$ of memory space, where n is the number of points.

3 ALGORITHM OVERVIEW

The algorithm we propose is iterative. In each iteration, two consecutive phases of parallel computing are executed:

- Phase 1: Detection, exclusion & processing.
- Phase 2: Repair.

On each iteration the algorithm transforms the mesh a step closer to the Delaunay mesh. The algorithm finishes when the Delaunay triangulation is reached. The following sub-sections explain the phases in more detail.

3.1 Detection, Exclusion and Processing

This phase is in charge of three steps: (1) detection of edges that do not fulfill the Delaunay condition, (2) exclusion of edges that can not be flipped in parallel and (3) processing the edges that can be flipped in parallel. Our algorithm maps threads to edges by using

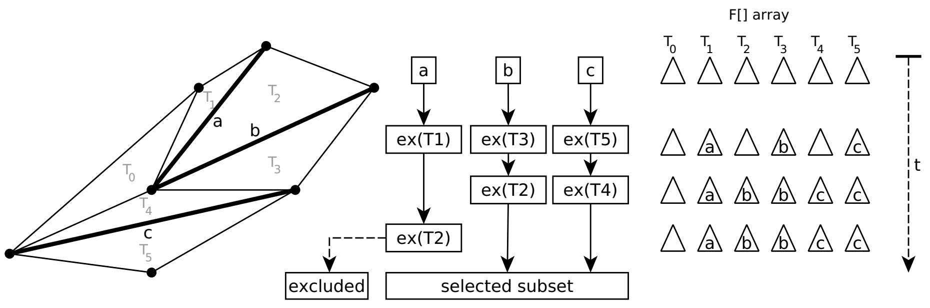


Figure 2: Exclusion mechanism. Each thread performs an atomic operation ex(T_i) to select its triangles. The F array has the information of which thread has taken a given triangle.

the PRAM model in such a way that thread t_i handles edge e_i with $i \in [0, ne-1]$ (*ne* is the number of edges). The execution threads detect first the edges that need to be flipped (bad edges), then they go through a filter where only the independent threads survive and finally, the survivors flip their edge.

For the detection step, threads test their corresponding edge e against the Delaunay condition by computing the opposite angles λ and γ of e using the information from t_a, t_b. The test must satisfy the following condition:

$$\lambda + \gamma \leq \pi \tag{1}$$

If the test of equation (1) fails, the edge is a bad edge and needs to be flipped. On the other hand, if the test passes, the execution thread ends. Most of the time it is not possible to flip the complete set of bad edges in one iteration because the flip of a given edge e compromises the consistency of the neighbor edges that belong to t_a and t_b. However, it is possible to process a subset A of the edges that satisfy the following condition:

$$\forall e_1, e_2 \in A \quad T_{e_1} \cap T_{e_2} = \emptyset; \;\; T_e = \{t \in T : e \in t\} \tag{2}$$

For implementing the exclusion step, the algorithm internally uses a *Flags* array where $Flags[i] ==$ *Taken* if the *i*-th triangle was flagged by a thread, and $Flags[i] == Free$ if it was not. Each thread that needs to flip an edge requires two flags to be set, the ones associated with the triangles that share its edge. This operation is done atomically (atomic operations are sequential only when two or more threads access the same memory location). When a thread flags the first triangle, some neighbor threads will be excluded (i.e the ones that failed to catch this flag). When a thread flags the second one, the rest of the neighbors will get excluded. Figure 2 shows an example, where edges a,b and c need to be flipped but only $\{a,c\}$ or $\{b,c\}$ can be processed at the same time. By using condition (2), the thread associated with edge a is excluded.

For the processing step, the per thread edge-flip method is designed as a swap of indices between the associated triangles t_a, t_b making a rotating effect of the triangles (see Figure 3). For a given edge e_i, our parallel edge-flip proceeds in the following way:

1. Variables: $O[] =$ Opposites, $T[] =$ *Triangles*, $E[] =$ Edges;
2. Get the opposite vertex indices o_1, o_2:
   o_1 = O[i][0]; o_2 = O[i][1];
3. Get $c_1 \in t_a, c_2 \in t_b$ such that $v_1 = T[c_1], v_2 = T[c_2]$:
   c_1 = E[i].t_{a_1}; c_2 = E[i].t_{b_2};
4. do the edge-flip:
   $T[c_2] = T[o_1]$; $T[c_1] = T[o_2]$;
5. Update t_a, t_b and v_1, v_2:
   $E[i].t_a = [o_1, c_1]$; $E[i].t_b = [c_2, o_2]$;
   $E[i].v_1 = E[i].t_{a_1}$; $E[i].v_2 = E[i].t_{b_2}$;

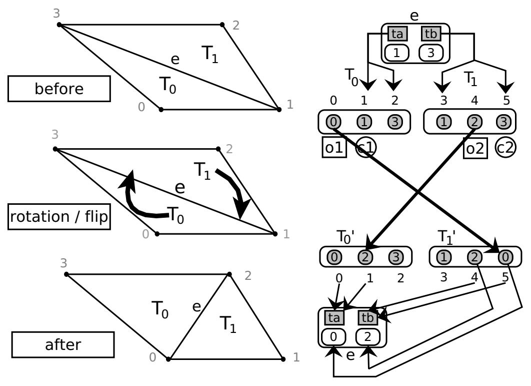


Figure 3: Visual example of edge-flip procedure for e.

The steps of this phase are summarized in Figure 4 showing how threads make their way down.

3.2 Repair

After the parallel edge-flips were executed, inconsistent information can be stored on neighbor edges. Some edges can store references to triangles whom they no longer belong (obsolete t_a and t_b pairs). Figure 5 shows a simple mesh where inconsistent information appears at edges d and b after e was flipped.

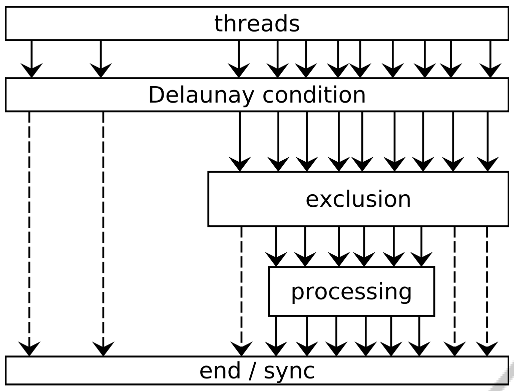


Figure 4: General view of the detection, exclusion and processing mechanism. Each block acts as a thread filter.

The information of the new triangles to whom *d* and *b* belong are in the triangles that were rotated while flipping *e*.

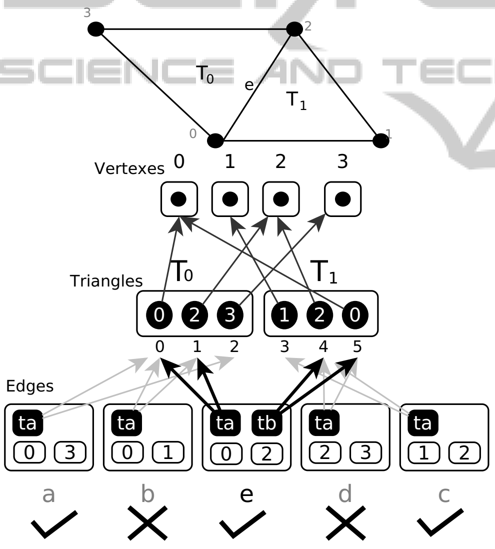


Figure 5: Edges marked with a cross are inconsistent.

Fortunately these inconsistencies can be easily identified with the following two expressions:

$$q = |v_1 - t_{a_1}| + |v_2 - t_{a_2}| \quad (3)$$

$$w = |v_1 - t_{b_1}| + |v_2 - t_{b_2}| \quad (4)$$

If $q > 0$ ($w > 0$) then t_a (t_b) needs to be repaired.

The rotation relations are stored at the moment of flipping an edge *e* using an array of rotations R[] of size *m* (number of triangles). The triangle that rotated with t_a and t_b is t_{ra} = R[$t_{a_1}/3$] and t_{rb} = R[$t_{b_1}/3$], respectively. Note that the indices stored in t_a and t_b point to the *Triangles* array and each triangle is defined by three consecutive vertex indices.

3.3 Handling Problematic and Worst Cases

During the first phase, there are two scenarios that require a more detailed explanation: **case (1);** the existence of co-circular configurations and **case (2);** the possible existence of dead-locks.

Case (1): if there are co-circular or almost co-circular configurations, our algorithm could fall into an infinite loop of edge-flips due to floating point errors. We solve this issue by using a small tolerance value in the evaluation of condition (1):

$$\lambda + \gamma \leq \pi + \varepsilon \quad (5)$$

This leads to ignoring some flips that in theory, should have been performed. That is why the generated triangulations may be quasi-Delaunay triangulations and not fully Delaunay triangulations. The ε value was experimentally estimated.

Case (2): a dead-lock could occur if there exists a circular chain of triangles, where all edges must be flipped and each thread can flag only one of its triangles. This kind of chain can not exist because it must have at least one edge that fulfills the Delaunay condition: the smallest edge of the chain. Note that the triangles that share the smallest edge are free to be flagged by a neighbor thread. Then, in chains like these there will always be at least one edge that can be flipped, therefore a dead-lock will never occur.

The known worst case configuration for Lawson's sequential algorithm is the one shown in Figure 6 (Edelsbrunner, 2001).

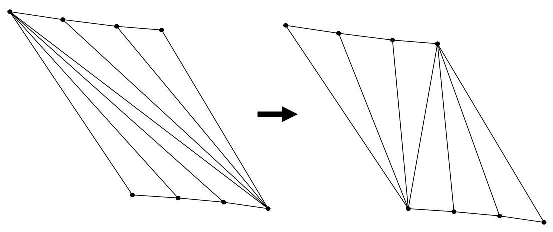


Figure 6: One of the worst cases for any edge-flip based method.

This worst-case triangulation has eight vertices, thirteen edges and six triangles. The algorithm executed five iterations and the number of flips per iteration was $\{1,2,3,2,1\}$. As this triangulation gets larger, the number of triangles increases and the number of iterations also grows. However, the performance of the algorithm is better than the sequential algorithm, because under the PRAM model the cost per iteration is $\Theta(1)$ as the algorithm can do several edge flips in parallel. Experimentally, we observed that for these configurations the number of required iterations is $m-1$, with m the number of triangles. The amount of edge-flips per iteration increases by one until the $m/2$-th iteration. Then, the number of parallel flips decreases by one until the last iteration is reached. The computational complexity is $O(n)$ (note that $m = O(n)$). This is an improvement over the sequential method, which in this case is $O(n^2)$.

4 IMPLEMENTATION DETAILS

Nvidia's CUDA architecture and API were chosen to implement the kernels, while OpenGL was chosen to render the triangulations. Using C type data structures for the mesh model, it is possible to represent vertex and triangle data via the OpenGL buffer objects: the VBO (Vertex Buffer Object) and EBO (Element array Buffer Object). In addition, CUDA supports OpenGL interoperability, meaning that threads can read and write directly into the VBO and EBO arrays. As with the vertices, the edges are also sent to the GPU at mesh loading time, and they can optionally be sent back at the end if needed (for example, to save the mesh into a file). The exclusion step is handled with atomic operations available from the CUDA C API. The performance is increased by using loop unrolling, coalesced memory on per edge data, minimal branching, constant types and shared memory to reduce registry usage. The implementation is available as a functionality of *cleap*, an open source C/C++ library (http://sourceforge.net/projects/cleap/).

5 EXPERIMENTAL RESULTS

In the following sections, we will refer to our implementation as MDT (Massive Delaunay Transformer). In order to analyze its performance and its behavior, we will evaluate the following aspects of the algorithm:

- Quality of the generated triangulations: how close they are to being Delaunay triangulations
- Computational time against (a) the Triangle software and (b) the CGAL library and (c) our own implementation of Lawson's algorithm
- Number of edges that can be flipped, number of edges that were flipped and number of edges that could not be flipped at each iteration.
- Influence of the mesh size in the number of iterations

Table 1 shows the hardware used for the evaluation. We have selected the algorithm available inside Triangle and the one available in the CGAL library because they are known to generate full Delaunay meshes and they have $O(nlog\ n)$ sequential implementations. Note that these algorithms start from a PSLG geometry and not from a given triangulation.

Table 1: Hardware used for testing.

Hardware	Detail
CPU	AMD Phenom I X4 9850 2.5 Ghz
GPU	Nvidia Geforce GTX 580
Mem	4GB RAM DDR2 800Mhz

5.1 2D Triangulations

The set of tests consists of fully random bad-quality triangulations in the sense that they need a high number of edge flips to be transformed into Delaunay triangulations. These inputs are generated by placing random points inside two adjacent triangles starting from a square domain. For each new inserted point, the triangle that includes the point is divided into three smaller triangles as shown in Figure 7. The size of the test triangulations ranges from 100 thousand to 5 million points and the smallest angle of all the meshes is practically zero (less than 10^{-6} radians).

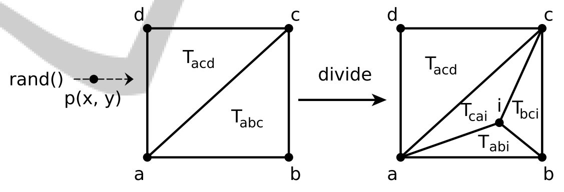
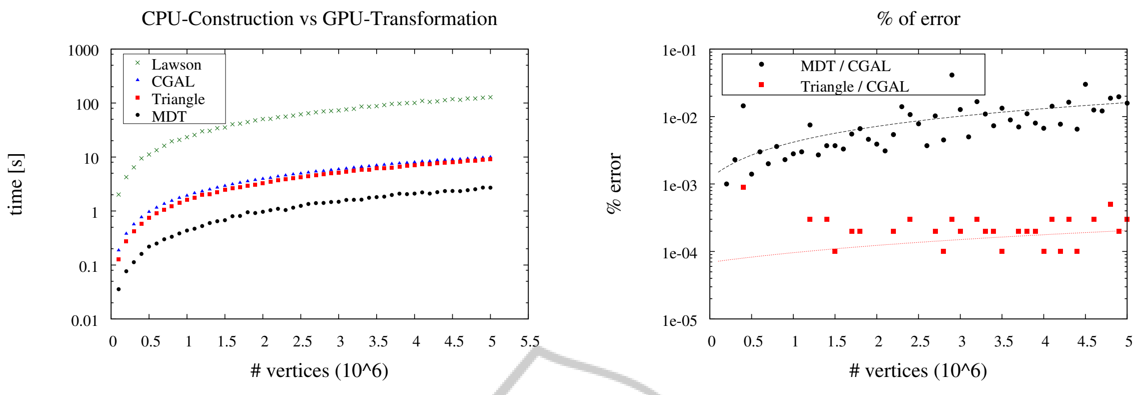


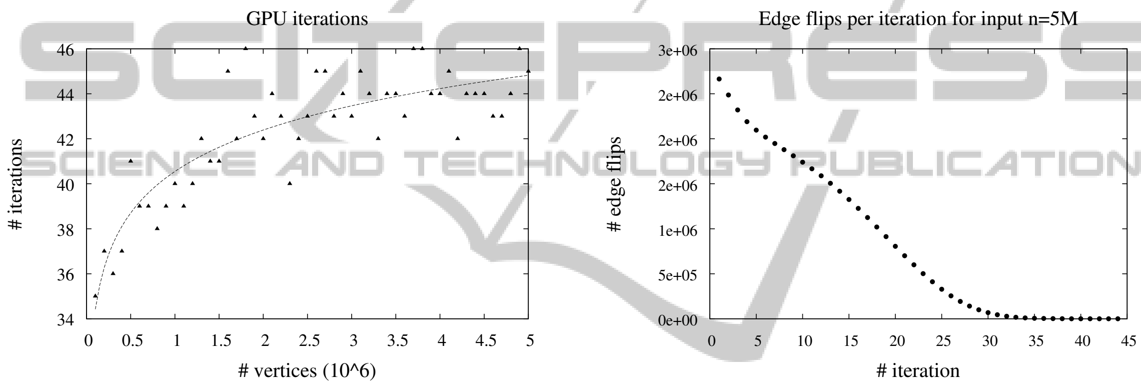
Figure 7: Construction of a full random mesh.

In Figure 8(a) we present the computational time for each mesh size. It can be observed that *MDT* is approximately three times faster in these bad quality triangulations than the algorithms inside the CGAL library and the Triangle software. There is also a speedup of 50× with respect to our sequential implementation of Lawson's original edge-flip method. In Figure 8(b) we show the quality of the generated meshes in the sense of how close they are to being full Delaunay meshes. We took the meshes generated by the CGAL library as reference triangulations. Both MDT and Triangle generate different triangles with respect to the reference triangulations. However, the missed triangles, i.e, the triangles that are in the triangulations generated by MDT and Triangle, and are not in the reference triangulations, are less than 0.05%. The error rate is computed as the number of missed triangles with respect to the total number of triangles of the reference triangulation. Note that the triangulations generated by CGAL and Triangle are different because Triangle modifies the vertex list if two points



(a) (b)

Figure 8: (a) Computational time for all methods and (b) Differences of Triangle and MDT triangulations with respect to CGAL triangulation.



(a) (b)

Figure 9: (a) Number of iterations vs. problem size, (b) number of edge-flips vs. iterations for the biggest input case (a mesh of 5 million points).

are too close to each other. Some important aspects of the behavior of the MDT are shown in Figure 9. Figure 9(a) shows an approximated curve that represents the number of iterations versus the mesh size. Empirically, this curve shows a complexity of $O(log\, n)$. Figure 9(b) shows how the number of edge flips changes among the iterations while transforming the triangulation of 5 million vertices and approximately 15 million edges. We can observe that during the first half of the iterations, most of the edge-flips are done, while in the last iterations few edge flips are executed. For this input, both MDT and Lawson's edge-flip methods performed approximately 37 million edge-flips. It is worth mentioning that in all the tested triangulations, the percentage of edge flips done in parallel was more than 80% (i.e., the excluded threads were less than 20%).

5.2 3D Surface Triangulations

MDT was originally intended for improving the minimum angle of smooth 3D surface triangulations for the modeling of tree stem deformations. An edge *e* is considered for flipping only if the normal vectors of the two neighbor triangles that share *e* are almost parallel according to some threshold value. Figure 10 shows the different test inputs with their corresponding number of vertices, edges and triangles.

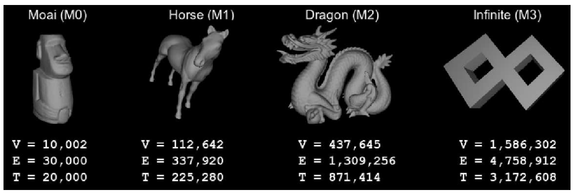

Moai (M0)
Horse (M1)
Dragon (M2)
Infinite (M3)
V = 10,002 E = 30,000 T = 20,000
V = 112,642 E = 337,920 T = 225,280
V = 437,645 E = 1,309,256 T = 871,414
V = 1,586,302 E = 4,758,912 T = 3,172,608

Figure 10: 3D Surface test-case meshes.

The dragon was taken from the Stanford Computer Graphics Laboratory, the horse from Cyberware Inc, the Moai from the GeomView examples and the Infinite was built with our custom tools. Figure 11 shows the performance of the MDT and our implementation of the Lawson sequential algorithm. (The traced lines were added to connect the measurements using the same implementation.) As expected, the MDT method achieves a speedup of 80×. Table 2 shows the number of flipped-edges and the percentage of excluded threads for each iteration in the four input meshes. As in the 2D tests, the first iterations do most of the required edge-flips. The number of iterations is lower than in the 2D tests because these surface triangulations have an overall better quality.

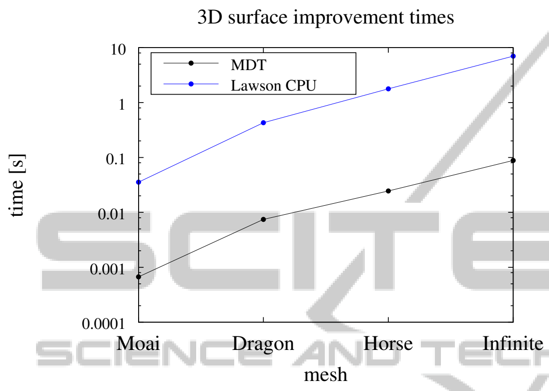


Figure 11: Performance results on 3D meshes.

6 RELATED WORK

There is a considerable amount of work on the subject of computing Delaunay triangulations, from sequential implementations (Paul Chew, 1989; Shewchuk, 1996; De Berg, 2000) to parallel ones (Antonopoulos et al., 2005; Healey et al., 1997; Kohout and Kolingerová, 2003; Rong et al., 2008). Most of these works belong to the case when a Delaunay triangulation needs to be computed from a set of points or from a PSLG, and not from an existing triangulation. To the best of our knowledge, only recently has the edge-flip technique been used in the design of parallel GPU-based algorithms for the generation of triangulations. Cao Thang (Cao, 2010) developed an algorithm for the generation of a Delaunay triangulation and its Voronoi diagram from a set of points. One step of his method performs GPU-based edge-flips; the algorithm maps threads to triangles. Cervenanský et al. (Cervenanský et al., 2010) propose a GPU-based triangulation algorithm for image processing. Edges are flipped in parallel, as we also do, but by using a different approach for deciding which subset of them can be flipped in parallel (i.e., they do not use Delaunay conditions). Harada (Harada, 2011) proposed a constraint solver for rigid body simulation. In his work, threads are assigned to pairs of adjacent triangles by using atomic operations in the same way as we do for deciding which non-Delaunay edges can be flipped in parallel. Unfortunately, we could not compare our implementation directly with the parallel edge-flip methods of the authors because different hardware was used in their results and they only compare their edge-flip routine against prior work of themselves. We think it is a better practice to use the standard method of comparison in parallel computing; to measure speedups against a reference sequential algorithm.

7 DISCUSSION AND CONCLUSIONS

We have presented a GPU-based implementation for computing quasi-Delaunay triangulations. The solution is compatible with OpenGL, handles special cases such as co-circular point configurations and is free of dead-locks. The behavior of the MDT shows several interesting aspects. The amount of edge-flips per iteration quickly decreases, making the first half of the iterations much more important than the rest. We report an exclusion rate of threads under 20% serving as a guarantee that parallelism can indeed be useful. The curve of the number of iterations as a function of the mesh size empirically shows a complexity of $O(log\ n)$. This is a good behavior since GPU methods are aimed at addressing large problems and less iterations means more parallelism. The worst behavior of the algorithm is when edge-flips can not be done in parallel. In this case the computational complexity for the sequential and parallel algorithms is the same.

We analyzed the performance of MDT under different inputs; bad-quality random 2D triangulations and popular 3D surface meshes. Our experimental evaluation shows that the percentage of missed triangles of the triangulations generated by MDT with respect to the triangulations generated by CGAL was less than 0.05% in all experiments. On these bad quality meshes, MDT obtains a speedup of up to 50× with respect to Lawson's $O(n^2)$ edge-flip method on CPU and a speedup of 3× with respect to the 2D $O(nlog\ n)$ algorithms available inside CGAL and Triangle. This speedup seems to be not so impressive as we are comparing GPU with CPU implementations and quasi-Delaunay triangulations with exact ones. However, it is important to mention that the MDT implementation is sensitive to the topology of the input triangulation and the CGAL and Triangle implementations are not because they are constructive methods. This means that if the input mesh needs little work to become Delaunay, the speedup of MDT with respect to CGAL and Triangle should be higher than 3×.

Table 2: Detail of effective edge-flips and parallelism ratio at each iteration for the 3d examples.

	Moai		Horse		Dragon		Infinite	
#iteration	flipped	excluded	flipped	excluded	flipped	excluded	flipped	excluded
1	1,786	3.88%	31,453	0.08%	106,101	6.63%	508,502	8.62%
2	215	1.83%	4,376	0.21%	22,608	4.08%	237,340	8.48%
3	41	4.66%	598	1.65%	4,455	5.86%	95,975	3.81%
4	3	0%	131	4.38%	995	1.49%	29,568	5.95%
5	1	0%	33	8.34%	207	1.43%	7,882	4.52%
6			10	0%	34	0%	2,844	1.6%
7			4	0%	1	0%	762	5.81%
8			1	0%			195	6.25%
9							40	0%
10							11	0%
Total flips	2,046		36,596		134,401		883,119	

Our proposed implementation is useful for applications that need to quickly improve the minimum angle of triangulations and visualize a mesh at the same time; dynamic terrain manipulation and tree stem deformations to name some examples. In the near future, we will compare the in-circle test with the opposite angle test used in this implementation. We also want to test the algorithm with bad quality 3D surface meshes.

ACKNOWLEDGEMENTS

This work was partially supported by Fondecyt Project N^o 1120495.

REFERENCES

Antonopoulos, C. D., Ding, X., Chernikov, A., Blagojevic, F., Nikolopoulos, D. S., and Chrisochoides, N. (2005). Multigrain parallel Delaunay mesh generation: challenges and opportunities for multithreaded architectures. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 367–376, New York, NY, USA. ACM.

Cao, T. T. (2010). Computing 2d Delaunay triangulation using GPU. *Manuscript in preparation*.

Cervenanský, M., Tóth, Z., Starinský, J., Ferko, A., and Srámek, M. (2010). Parallel GPU-based data-dependent triangulations. *Computers & Graphics*, 34(2):125–135.

De Berg, M. (2000). *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA.

Edelsbrunner, H. (2001). Geometry and topology for mesh generation (Cambridge monographs on applied and computational mathematics).

Fortune, S. (1993). A note on Delaunay diagonal flips". *Pattern Recognition Letters*, 14(9):723 – 726.

Harada, T. (2011). A parallel constraint solver for a rigid body simulation. In *SIGGRAPH Asia 2011 Sketches*, SA '11, pages 22:1–22:2, New York, NY, USA. ACM.

Healey, R. G., Minetar, M. J., and Dowers, S., editors (1997). *Parallel Processing Algorithms for GIS*. Taylor & Francis, Inc., Bristol, PA, USA.

Kohout, J. and Kolingerová, I. (2003). Parallel Delaunay triangulation based on circum-circle criterion. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 73–81, New York, NY, USA. ACM.

Lawson, C. L. (1972). Transforming triangulations. *Discrete Mathematics*, 3(4):365 – 372.

Navarro, C., Hitschfeld-Kahler, N., and Scheihing, E. (2011). A parallel GPU-based algorithm for Delaunay edge-flips. In *Abstracts from 27th European Workshop on Computational Geometry (EUROCG2011)*, pages 75–78. Morschach, Switzerland.

Nvidia (2011). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.

Paul Chew, L. (1989). Constrained Delaunay triangulations. *Algorithmica*, 4:97–108.

Rong, G., Tan, T.-S., Cao, T.-T., and Stephanus (2008). Computing two-dimensional Delaunay triangulation using graphics hardware. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97, New York, NY, USA. ACM.

Shewchuk, J. R. (1996). Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In ACM, editor, *First Workshop on Applied Computational Geometry*, pages 124–133. (Philadelphia, Pennsylvania).

CGAL (2012). CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

Tobler, R. F. and Maierhofer, S. (2006). A mesh data structure for rendering and subdivision. In *WSCG '2006: Proceedings of WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision)*, pages 157–162.