A Quasi-Parallel GPU-Based Algorithm for Delaunay Edge-Flips*

Cristobal A. Navarro
Informatics Institute, Austral University of Chile, Valdivia, Chile
e-mail:cnavarro@cecs.cl

Nancy Hitschfeld-Kahler
Department of Computer Science, FCFM, University of Chile, Santiago, Chile
e-mail:nancy@dcc.uchile.cl

Eliana Scheihing
Informatics Institute, Austral University of Chile, Valdivia, Chile
e-mail:eliana.scheihing@gmail.com

February 28, 2011

Abstract

The Delaunay edge-flip algorithm is a practical method for transforming any existing triangular mesh S into a mesh $T(S)$ that satisfies the Delaunay condition. Although several implementations of this algorithm are known, to the best of our knowledge no parallel GPU-based implementation has been reported yet. In the present work, we propose a quadriphasic and iterative GPU-based algorithm that transforms 2D triangulations and 3D triangular surface meshes into Delaunay triangulations and improves strongly the performance with respect to a sequential CPU-implementation in large meshes. For 3D surface triangulations, we use a threshold value to prevent edge-flips in triangles that could deform the original geometry. On each phase, we address the main challenges that arise when adapting the problem to a parallel architecture and we present a GPU-based solution for each high CPU-consuming time step, reducing drastically the number sequential operations needed.

*Sent to the ACMTog journal, August 2010

1 Introduction

Delaunay triangulations are widely used in several applications because they have several good properties, that make them very useful in 2D and 3D simulations. In particular, a 2D Delaunay triangulation of a point set P maximizes the size of its smallest angle over all the possible triangulations for this point set P. Thus, this type of triangulation is composed by triangles that are more close to the equilateral ones than the triangles of the other triangulations.

Triangulations are present in different applications such as video-games, physic simulations, terrain rendering and medical 3D reconstruction among others. Delaunay triangulations can be achieved in two ways: (a) create it from an input geometry or a set of points, or (b) transform an already generated triangulation into one that satisfies the Delaunay condition. For the first case, there is already a considerable amount of work done, both for CPU [1, 2] and GPU architectures [3, 4, 5]. The second case appears when a triangulation was generated with some triangulation method or a triangulation (Delaunay or not) was deformed by applying some transformation to the mesh points, and a user wants to transform it in a Delaunay

mesh. The well known edge-flip algorithm becomes a practical candidate to transform any triangulation to a Delaunay one because of its simplicity.
With the high processing power of todays GPUs, it becomes natural to design a GPU implementation of the edge-flip algorithm and test if the performance of the known recursive CPU-based algorithm [2] can be improved. To our knowledge no such algorithm and its implementation has been reported yet. By solving the different challenges for a GPU parallel implementation of the edge-flip algorithm, we found out that the solution was not as straightforward as it looked initially. This is why flipped edges compromise the state of their neighbors, therefore the algorithm cannot flip all edges at the same time. Then, it is necessary to select a subset of them, where each one of these edges can be processed independently.
The contribution of this work consists on proposing a new quadriphasic-iterative GPU-based algorithm for transforming a given 2D triangulation or surface triangulation S into a Delaunay triangulation $T(S)$. The four phases are: (1) "Labeling", where the edges that can be flipped are marked, (2) "Selection", where a subset of edges that can be processed in parallel is selected, (3) "Processing", where the selected edges are flipped and, (4) "Update", where some inconsistency links are repaired. Phases (1), (3) and (4) are completely performed by the GPU, while the CPU is in charge of the "Selection" phase, because we can not decide in parallel which edges can be selected and which ones cannot.

The rest of the paper is organized as follows: Section (2) describes prior work done on the subject, as well as what can still be done as a contribution on this topic. Section 3 describes the designed data structures and how they are related among them. Sections 4 and 5 cover the algorithm and implementation respectively. In section 6 we present quantitative results from different tests, to finally discuss and conclude our work in section 7. Section 8 includes the code of the proposed algorithm.

2 Related Work

In the last two decades there has been a considerable amount of work done on the subject of computing Delaunay triangulations, from different sequential implementations [1, 2] to recently parallel [6], and in particular, GPU-based methods [5]. However, most of these works belong to the case when a Delaunay triangulation needs to be created from a given set of points and not from an existing triangulation. In the other hand, transformation methods have relied on the edge-flip technique applied recursively over a mesh, transforming an existing mesh S intro $T(S)$. The edge flip method was first introduced by Lawson [7] and originally oriented for 2D triangulations, but by using the proper conditions, it can be applied to 3D surface triangulations as well. Though the concept of the edge-flip is simple and the implementation is straightforward, the order of the algorithm is $O(n^2)$ in the worst case [8, 9] where n is the number of points of the triangulation. For a large number of edge flips, the algorithm can be quite slow at processing it sequentially. The idea of adapting the problem to a parallel architecture, where each execution thread can process independent edges, is not free of challenges. Neighbor dependency, data consistency, and adequate edge selection are some of the main difficulties when creating a GPU-based parallel solution.

3 Involved data structures

Proper data structures have been defined to represent a triangulation and neighborhood relationships in order to use as efficiently as possible the GPU's architecture. Figure 1 illustrates how we represent the most important mesh components: Vertices, triangles and edges.
Vertices are handled via the Vertices array where each element contains a position (x, y) for a 2D triangulation or (x, y, z) for a 3D surface triangulation, plus any other additional information needed (normal vector, color, etc). The Triangles array is a set of indices to the Vertices array where each three consecutive indices corresponds to a triangle. For each edge, the

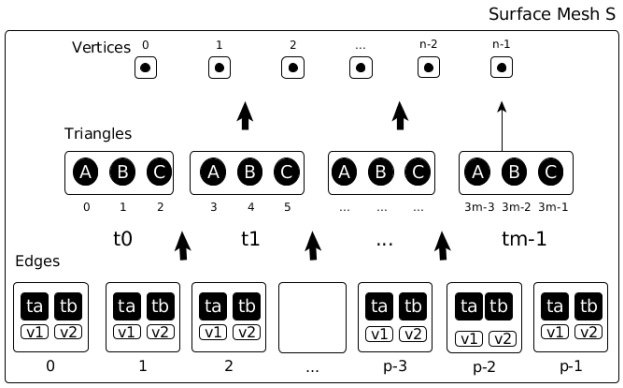

Figure 1: Data structures used for mesh representation.

Edges array contains a pair of vertex indices $v1$, $v2$ to the Vertices array and a pair of references ta, tb to the triangles that share this edge (for boundary edges, tb remains unused). Both, ta and tb are defined by a pair of indices that point to the exact positions in the Triangles array where this edge can be found. In other words, each edge can access its vertex data by using the indices provided by $v1$, $v2$ or by going through the references ta, tb. This redundant information becomes very useful to efficiently check if the edge information is still consistent after an edge flip was performed. This data model was designed so that it could be naturally implemented and integrated with the OpenGL API and, at the same time, could be fully and efficiently implementable on the CUDA [10] architecture.

4 Algorithm Overview

The flip-edge algorithm is divided into four phases that work together in the following order:

- Phase 1: Labeling.
- Phase 2: Selection.
- Phase 3: Processing.
- Phase 4: Update.

4.1 Labeling

On the labeling phase, all edges are tested with the Delaunay angle condition. For any given edge e, the test routine computes its opposite angles λ and γ from the triangles t_a, t_b that share e (by using the Edges array previously defined in section 3) and checks if the sum satisfies the following condition:

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

If the opposite angles fulfills the condition, then e is labeled as $d = 1$, otherwise it is labeled as $d = 0$. This label can be computed independently for each edge of the mesh. Figure 2 shows a small mesh S_e composed by only two triangles, where edge e does not satisfy Delaunay condition (1). In the example,

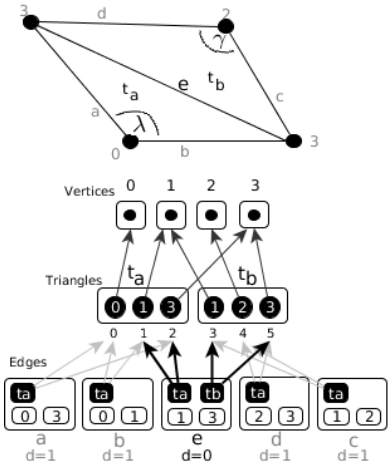

Figure 2: Labeling phase for a given triangulation.

edge e is the unique internal edge of the mesh, thus the only one to be analyzed. The return value when testing e is $d = 0$, while boundary edges are always labeled as $d = 1$ by default. The "Labeling" routine can be modeled for the GPU with a parallel paradigm, since each edge is an isolated problem that can be assigned to an individual execution thread. In addition, all r/w instructions are performed on the thread's analyzed edge while the Triangles array remains read-only. This last observation assures that

all threads access unmodified and consistent data at any execution instant.

In 2D triangulations the "Labeling" phase is just done as mentioned before. For tridimensional surface triangulations, we do not flip edges that would deform the domain geometry. We use a threshold value for the angle formed by the normal vector defined by the triangles that share an edge. Let β be the threshold value, then an edge can be labeled if the angle μ formed by the normal vectors $\overrightarrow{N_0}$ and $\overrightarrow{N_1}$ is less than or equal to β (see Figure3).

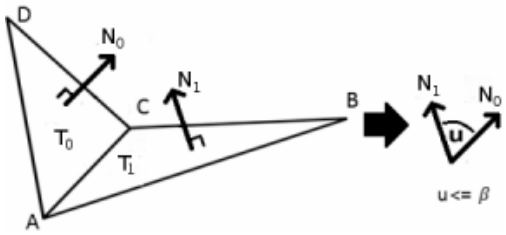

Figure 3: Condition: $\mu \leq \beta$

4.2 Selection

Because of neighbor dependency problems, it is not possible to flip all the edges labeled as $d = 0$ at the same time. Maybe in some particular cases this will be possible, but it cannot be taken as a general rule. By definition, the flip of a given edge e will always produce a transformation on the triangles (t_i, t_j) where this edge belongs to. This transformation affects directly the state of the other edges of the triangles that share e, making impossible to flip them while e is being flipped. However, it is possible to achieve the same desired result by processing in parallel a subset of edges that are independent among them. In order to get a proper subset, a scan is done through all the edges that have got labeled as $d = 0$. As a general rule, a given subset A will be completely independent only and only if:

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

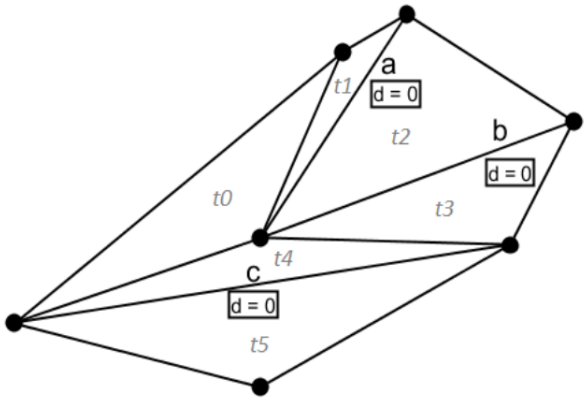

Figure 4: The first Subset for this mesh can either a,c or b,c.

In other words, an edge with $d = 0$ is added to the subset A if it does not share any of their associated triangles with the other edges already included in A. To illustrate how condition 2 is applied, Figure 4 shows the selection process for an arbitrary mesh, where edges a, b, and c need to be flipped but they cannot be flipped at the same time. After applying condition 2, the resulting subset can be either $\{a, c\}$ or $\{b, c\}$ but a and b cannot be together in any case, because they share t_2.
The implementation of this phase fits better on a CPU architecture because these dependencies need to be solved sequentially in order to build a consistent subset of parallel processing edges.

4.3 Processing

Once the subset A is correctly generated, the algorithm proceeds to the third phase which can also be processed in parallel via the CUDA kernel. We define the per thread edge-flip method as an index exchange between the two triangles t_a, t_b related to the processed edge e. Rather than using dynamic arrays where e could be deleted and created again as a flipped edge, we propose a different approach, where the operation can be seen as a geometric transformation of the pair of triangles that share the edge e. More precisely, this transformation can be seen as a

rotation of the triangles t_a, t_b fully independent from the rest of the mesh. All execution threads can perform this transformation in parallel because they do not share any triangle. The transformation is done on the Triangles array using the information related to t_a and t_b stored in the Edges array by following these steps:

- Find the opposite vertex indices u_a and u_b of e in the Triangles array going through ta and tb (Edges array).
- Locate the position of the first common vertex index c_1 in the Triangles array going through ta.
- Locate the position of the second common vertex index c_2 in the Triangles array going through tb.
- Exchange data, by copying u_1 into Triangles[c_2], and u_2 into Triangles[c_1].
- Update the values of ta, tb and $v1$, $v2$ related to e in the Edges array.

Figure 5 illustrates the case of flipping the edge e from the example mesh S_e used in "Labeling" phase (Fig.2). For this example, just one execution thread is needed. For larger meshes, the subset A can contain thousands of edges and then the same amount of threads will be needed to process them. However, the logic inside each thread will always remain simple as in this small case.

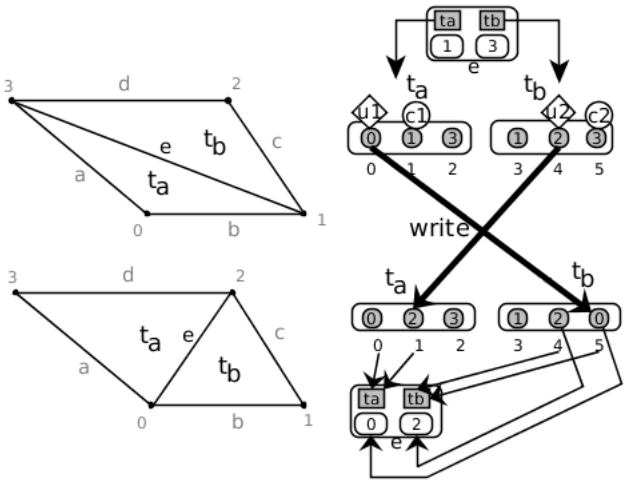

Figure 5: The edge-flip is applied as a rotation of triangles.

4.4 Update

After an edge-flip, consistency problems might appear on the Edges array, specifically on the edges surrounding the flipped edge e. On the previous phase, the algorithm updated only the values of the flipped-edges in the Edges array, but the other affected edges did not receive any update; they can now store references to triangles to whom they don't belong anymore (obsolete ta, tb values). We say that an edge is inconsistent when its vertex indices obtained through ta, tb differ from the ones stored in $v1$ and $v2$. It is important to mention that in our data structure model, $v1$ and $v2$ will always contain correct indices to the vertices array of any non-flipped edge e, no matter how many edge flips were performed. This means that edges never rotate, therefore they remain in the same original place but belonging to different triangles t_a, t_b. In the example mesh S_e used for the previous phase, inconsistent information appeared at edges b and d right after flipping e (see Figure 6).

The detection of an inconsistent edge is trivial; Given an edge e, compare the vertex indices accessed via ta and tb against $v1$ and $v2$. If the values are not the same, then the edge is partially inconsistent (just the indices of one triangle are not correct) or completely inconsistent (the indices of both triangles are not correct). In any case, the indices $v1$ and $v2$ stored on each inconsistent edge become the target values it needs to be recovered through the Triangles array. The key point here is to observe that if the access through ta is wrong, it means that the correct values are in the triangle that was rotated together with t_a (while the edge shared by both was flipped). The solution is then to provide a way that each edge can access the other triangle involved in the edge-flip (rotation). This information can be obtained from the Selection phase: Each time an edge is added to the subset A, a pair of triangle indices t_i, t_j is stored in the array R[] in the form of R[t_i] = t_j and vice-

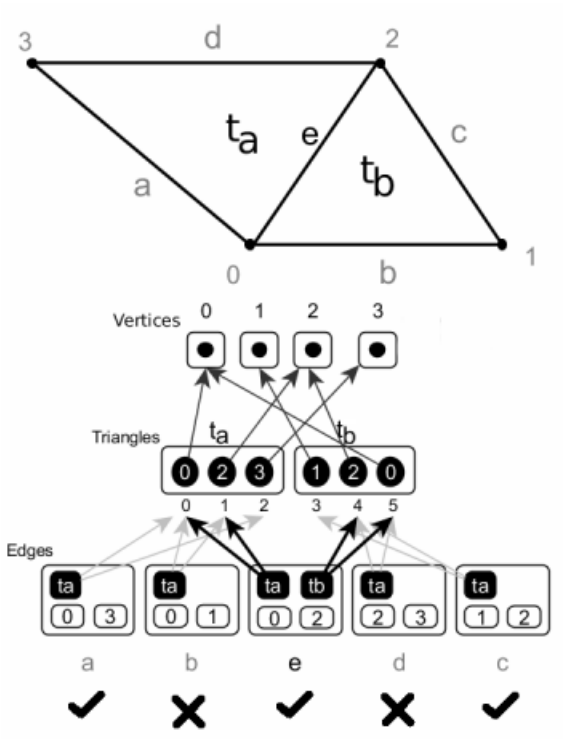

Figure 6: Edges marked with a cross indicate inconsistent information.

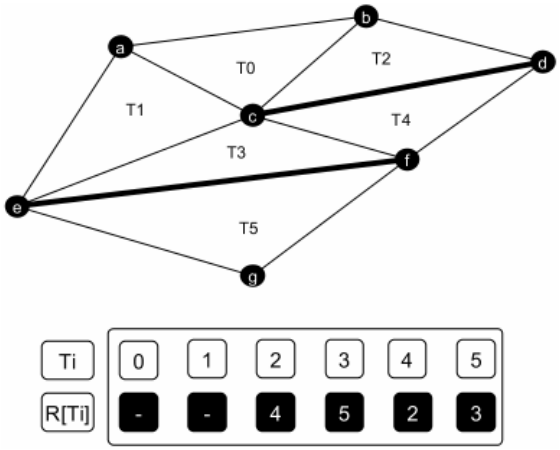

Figure 7: R array holds the relations of all involved triangles.

versa (see Figure 7). Then, if ta contains inconsistent information, $t_i = ta[0]\ div\ 3$ is used as the triangle index in the R array to find $t_j = R[t_i]$, the other triangle that must contain the indices stored in $v1$ and $v2$. For edges completely inconsistent, the update process is done for t_a as well as for t_b.

Since the update of a given edge does not affect its neighbors, it is possible to model a parallel solution where each execution thread can update one particular edge, covering the entire mesh domain in one single kernel call.

Each phase plays an important role for the global algorithm. In fact, each one uses the results from the previous one. When all four phases complete their tasks, we say that an iteration occurred and a subset of edges has been flipped completely in parallel. Each time a new iteration of the algorithm is applied, other subset of edges is flipped and a better triangulation is achieved. Finally, the algorithm ends when all edges become labeled as $d = 1$, which is an indicator that the mesh has become a 2D Delaunay triangulation and or close to a 3D Delaunay triangulation if some edges were not flipped because of geometry restrictions.

Figure 8 shows a flow diagram of the algorithm and how the four phases are integrated.

5 Implementation

Nvidia's CUDA architecture and libraries were chosen to implement the kernels, while OpenGL was chosen to render mesh surface. Thanks to the C type structures used for the mesh model, it is possible to represent vertex and triangle data via OpenGL buffer objects (VBO and EBO respectively). Additionally, CUDA is capable of mapping these OpenGL buffer arrays into its kernels with no need of resending data from Host to Device, increasing efficiency. For the edges, there was no need to keep them on the Device side, because they do not make part of the OpenGL rendering pipeline. Therefore, they are

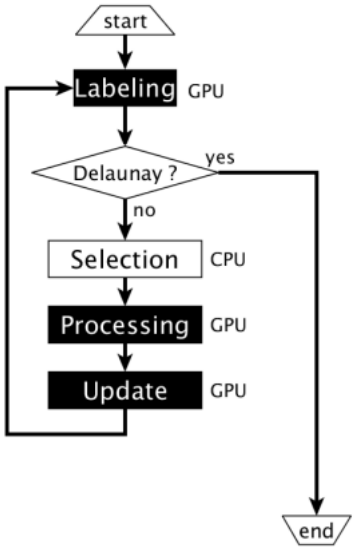

Figure 8: Quadriphasic Iterative algorithm.

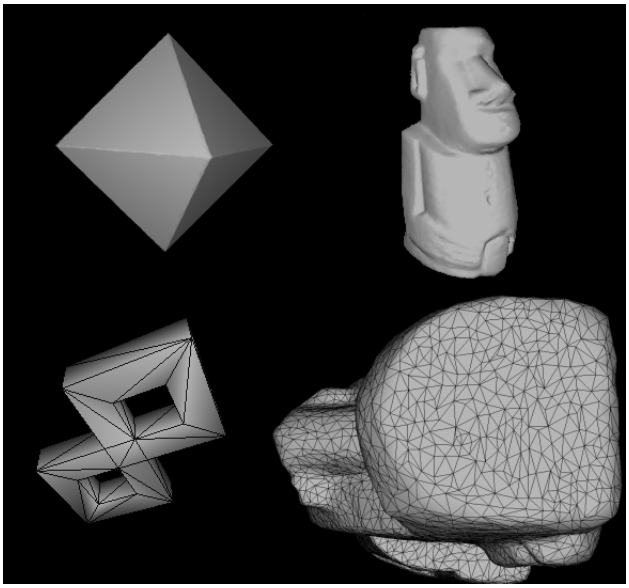

Figure 9: Different Meshes loaded into our implementation.

sent from Host to Device each time the kernel needs them. Figure 9 shows how meshes are visualized in our software by sending the mesh data directly into GPU memory via OpenGL.

6 Tests and results

Each test consists on measuring the time needed to process a given mesh. For each mesh, the test is repeated four times, and the average value is kept for the record. Because the standard deviation became too insignificant, we chose not to include it as part of the results. The same tests are also performed to a CPU-based edge-flip algorithm. Table 1 shows the testing platform used for all tests.

Table 1: Testing platform

Hardware	Detail
GPU	Nvidia Geforce 9800GTX+, 512MB
CPU	AMD Phenom x4 850
Mem	4GB RAM DDR2 800Mhz
OS	Linux Ubuntu 64-bit

Table 2: Meshes used as test examples

Mesh ID	Vertexes	Edges	Triangles
1	32	68	102
2	1,222	2,448	3,672
3	10,002	30,000	20,000
4	23,490	70,290	40,648
5	360,002	1,080,000	720,000

Table 2 presents the test meshes with their vertex, triangle and edge number. Table 3 shows the results in seconds for both CPU and GPU implementations. Symbol I is the improvement factor obtained by using the GPU implementation.

For mesh number 5, we show how the number edge-flips performed relate to each iteration of the algorithm (See Figure 10). As expected, the first three iterations concentrate most of the parallel operations and the last ones are in charge of flipping the remaining edges to reach the Delaunay triangulation. However, not all meshes will have this behavior, in fact,

Table 3: Performance comparison

Mesh	Iterations	T_{GPU}[s]	T_{CPU}[s]	$I=\frac{T_{CPU}}{T_{GPU}}$
1	1	0.0044	0.0025	0.5619
2	4	0.0220	0.1833	8.3412
3	5	0.0584	0.8993	15.397
4	1	0.1405	4.1937	29.8544
5	7	1.3884	55.678	40.1022

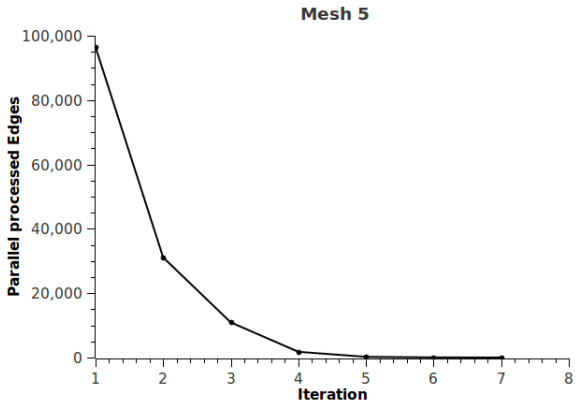

Figure 10: Parallel processed edges at each iteration.

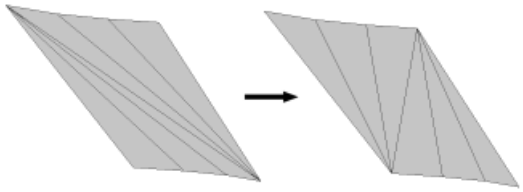

Figure 11: One of the worst cases for the edge-flip algorithm.

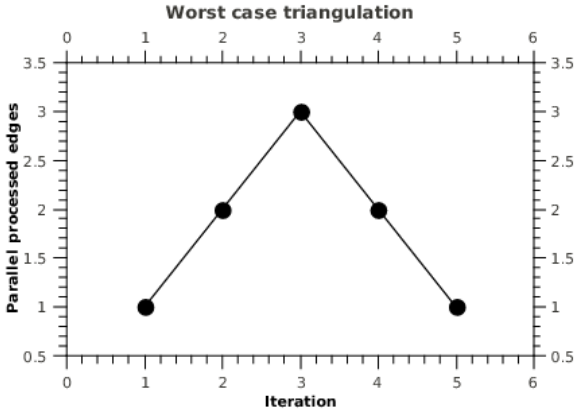

Figure 12: Behavior of our proposed algorithm on the worst case.

the behavior depends on the mesh topology. For special meshes like the ones presented on the Edelsbrunner's book[9], our proposed algorithm concentrate most of its edge-flips on its central iterations. Figure 11 shows an example with one of the worst cases for Delaunay transformations based on the edge-flip algorithm. Figure 12 shows that the number of edge-flips is maximum on the middle iterations, and minimum on the first and last ones. The test was applied also to equivalent meshes with 12 and 16 nodes distributed in the same way as the 8-node mesh shown in Figure 11, and the same behavior was observed.

7 Discussion and Conclusions

To our knowledge, no parallel GPU solution has been proposed for this problem, making the present work a first proposal for the implementation of the edge-flip strategy on a Graphics Processing Unit. We have presented a parallel approach for managing the triangle neighbor dependency, the edge flipping and the detection and update of inconsistencies. With the proposed mesh model structure, it was possible to achieve full integration with the graphics rendering pipeline (OpenGL) and with the GPU hardware.

Our proposed algorithm is best suited for meshes composed by more than thousand edges. For the test of Mesh $N^o = 5$, our implementation took approximately 1.4[s] while the CPU classic implementation took at least 55[s] obtaining a 4000% performance improvement. Although the results indicate that the GPU solution is faster, the sequential CPU algorithm still becomes useful as a complement for processing small meshes, because its performance is approximately two times better than our GPU implementation. The reason of this behavior is because the cost

of processing a low quantity of edges on the GPU is not enough to justify the cost of hardware initialization and data transfers through the PCI-Express bus. For one of the worst case meshes, our algorithm presented a particular behavior maximizing the number of edge-flips that can be executed in parallel at the middle iterations instead of at the beginning ones. This fact was not initially expected.

The proposed algorithm can also be useful to improve the performance of previous works on Delaunay triangulations. For example, the authors of the algorithm described in [5] could improve their results by considering our work for their step "C4" which is computed completely on the CPU.

Nowadays, with actual and modern GPU hardware, it is possible to achieve better results because the CUDA architecture scales performance automatically as more stream processors are available. Additionally, with the recent introduction of the new GPU architecture based on MIMD, it is possible to take advantage of the multi-instructional feature and get higher performance by launching parallel kernels.

In the near future, we would like to get information about at which mesh size n (number of edges) will be always convenient to use a GPU-implementation independent of the number of the edge flips that must be done and for meshes which less than n edges, at which number of edge-flips would be useful. Finally, if we could find a way to reduce the CPU computing time at the Selection phase, the performance could increase considerably.

8 Code Listings

Phase 1: Labeling.

```
//#Labeling (GPU)
__global__ void labeling(vertex* vert, int* tri,
                                        edge* e){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if( i<NUM_EDGES ){
      if( e[i].internal() ){
          //e[i] is an internal edge
          dAngle = delaunayAngle(vert, tri, e[i]);
          gAngle = geometryAngle(vert, tri, e[i]);
          e[i].d = isDelaunay(dAngle) +
                isGeoRestricted(gAngle, ANGLE_LIMIT);
          if( e[i].d == 0 ){
              endFlag = false;
          }
      }
      else{
          //e[i] is a boundary edge
          e[i].d = 1;
      }
  }
}
```

Phase 2: Selection:

```
//Selection (CPU)
void selection(Edge* edges){
   int selSize=0;
   int triRel[NUM_TRIANGLES], triUsed[NUM_TRIANGLES];
   int selection[NUM_TRIANGLES];
   for(int i=0; i<getNumEdges(); i++){
      e = &edges[i];
      if( e->delaunay == 0 ){
         if( triAvailable(triUsed, e->t1, e->t2) ){
             //triangle available, mark as used.
             markUsed( triUsed, e->t1, e->t2 );
             markRelation( triRel, e->t1, e->t2 );
             //add edge index to selection
             addSeleccion(selection, i);
             selectedEdges++;
         }
      }
   }
}
```

Phase 3: Processing.

```
#Procesamiento (GPU)
--global__ void edgeFlip(vertex* vbo, GLuint* eab,
                         Edge* e, int* eIndexes){

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if( i<SELECTION_SIZE ){
    //eIndexes has the selected edges from phase 2
    int myEdge = eIndexes[i];
    T1 = e[myEdge].t1;
    T2 = e[myEdge].t2;
    //Search opposite vertexes
    getOpposites(eab, T1, T2,
                 &opposite1, &opposite2);
    //Search first common vertex
    searchCommon(T1, opposite1, NULL, &common1)
    //Search second common vertex
    searchCommon(T2, opposite2, c1, &common2)
    //flip -- exchange data
    eab[common1] = eab[opposite2];
    eab[common2] = eab[opposite1];
    //update flipped edge's values.
    updateEdge( myEdge, e, i, eab,
                opposite1, opposite2,
                common1, common2);
  }
}
```

Phase 4: Update.

```
#update (GPU)
__global__ void update( GLuint* eab,
             int* trirel,
       Edge* edges,
                           int numPares){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if( i<numPares ){
        int index;
        if( !cons(e[i], eab, e[i].t1) )
             int Tr = trirel[ e[i].t1 ];
             update( e[i], eab, Tr );
        }
  if( e[i].count < 2 )
      return;
        if( !cons(e[i], eab, e[i].t2)){
      int Tr = trirel[ e[i].t2 ];
      update( edge[i], eab, Tr );
        }
    }
}
```

References

[1] J. R. Shewchuk, "Triangle: Engineering a 2d quality mesh generator and delaunay triangulator," in *First Workshop on Applied Computational Geometry* (ACM, ed.), pp. 124–133, (Philadelphia, Pennsylvania), 1996.

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

[3] R. G. Healey, M. J. Minetar, and S. Dowers, eds., *Parallel Processing Algorithms for GIS*. Bristol, PA, USA: Taylor & Francis, Inc., 1997.

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

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

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

[7] C. L. Lawson, "Transforming triangulations," *Discrete Mathematics*, vol. 3, no. 4, pp. 365 – 372, 1972.

[8] S. Fortune, "A note on delaunay diagonal flips," *Pattern Recognition Letters*, vol. 14, no. 9, pp. 723 – 726, 1993.

[9] H. Edelsbrunner, *Geometry and Topology for Mesh Generation (Cambridge Monographs on Applied and Computational Mathematics)*. New York, NY, USA: Cambridge University Press, 2001.

[10] N. Corporation, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.