Parallel constrained Delaunay triangulation

Narcís Coll and Marité Guerrieri[*]

Geometry and Graphics Group. Universitat de Girona

Abstract

In this paper we propose a new GPU method able to compute the 2D constrained Delaunay triangulation of a planar straight line graph consisting of points and segments. The method is based on an incremental insertion, taking special care to avoid conflicts during concurrent insertion of points into the triangulation and concurrent edge flips.

Introduction

The constrained Delaunay triangulation (CDT) is one of the fundamental topics in Computational Geometry and it is used in many areas such as terrain modelling, finite element method, pattern recognition, path planning, etc. A CDT of a planar straight line graph (PSLG) can be constructed in the following way: begin with an arbitrary triangulation of the PSLG points; examine each PSLG segment in turn to see if it is an edge; force each missing PSLG segment to be an edge; then flip non-locally Delaunay edges until all edges are locally Delaunay with the provision that PSLG segments cannot be flipped. There are two ways to force a PSLG segment to be an edge of the triangulation. The first consists of deleting all the edges it crosses, then inserting the segment and retriangulating the two resulting polygons (one from each side of the segment). The second approach consists of iteratively flipping the edges that crosses the segment until the segment is an edge. Qi et al. [2] presented a GPU method for computing the CDT of a PSLG. This method has three phases. The first phase computes the Delaunay triangulation of the PSLG points, while the second inserts the PSLG segments into the triangulation using edge flipping and the third flips the remaining non-locally Delaunay edges. Experimental results show that the method achieves a significant speedup with respect to the best CPU methods. However, since the Delaunay triangulation is constructed using a digital Voronoi diagram computed on a uniform grid, the method needs a huge memory space to allocate the grid and is less efficient when the distribution of the input points is far from being uniform.

Our approach of computing the CDT on the GPU extends the method presented in [1] by simultaneously inserting points and segments into the triangulation.

1 Our approach

Our algorithm is based on an iterative process that finishes when all PSLG elements (points and segments) are inserted into the Delaunay triangulation and no edge needs to be flipped. In each iteration as many elements as possible are inserted with the condition that only one point can be inserted into one single triangle. Each iteration is divided into four steps: location, where the triangle containing every non inserted point is determined; insertion, where at most one point per each triangle is inserted; marking, where some edges of the triangulation are marked to be a segment or marked to be flipped because they are crossed by a segment or they are non-locally Delaunay; and flipping, where some of the marked edges are flipped thus avoiding conflicts between them.

Let n be the number of points and m be the number of segments. In order to use the GPU's resources as efficiently as possible the following arrays allocated in the global memory are used:

`Points`. Positions (x, y) in 2D. Its first three positions corresponds to the three vertices of a large auxiliary triangle that contains all points. Size $n + 3$.

`Segments`. Indices to `Points`. Each two consecutive indices correspond to a segment. Size $2m$.

`Inserted-P`. Binary flags to determine whether a point has been inserted or not. Size $n + 3$.

`Inserted-S`. Binary flags to establish whether a segment has been inserted or not. Size m.

`Triangles`. Indices to `Points`. Each three consecutive indices correspond to a triangle. Position zero of this array corresponds to the auxiliary triangle. Size $3(2n + 1)$.

`Neighbours`. Indices to `Triangles`. Each three consecutive indices correspond to the current neighbours of the triangle. Size $3(2n + 1)$.

`FutureNeighbours`. Indices to `Triangles`. Each three consecutive indices correspond to the neighbours that the triangle will have after executing a point insertion or a flip which affects its current neighbours. Size $3(2n + 1)$.

[*]Email: (coll,mariteg)@imae.udg.edu. Authors research supported by TIN2010-20590-C02-02.

`ContainingTriangle`. Indices to `Triangles` to record which triangle contains a point, in the case that the point has not been inserted, or otherwise, a triangle incident to the point. Initially, all points are contained in the auxiliary triangle. Size $n + 3$.

`PointToInsert`. Indices to `Points` to record which the next point to be inserted in each triangle is. Size $2n + 1$.

`Flip`. Flags (0, 1, 2, 3) to discern whether an edge has to be flipped or not. Each three consecutive flags corresponds to a triangle. Flag 0 indicates the edge is not a segment and does not have to be flipped. Flag 1 indicates the edge is not a segment and has to be flipped because it is non-locally Delaunay. Flag 2 indicates the edge is a segment and does not have to be flipped. Flag 3 indicates the edge is not a segment and has to be flipped because it is crossed by a segment. Size $3(2n + 1)$.

`EdgeToFlip`. Flags (0,1,2,3,4,5,6) to record, for each triangle, which is the edge that will be flipped (0,1,2), or no edge will be flipped (3) or the edge that will be flipped by an adjacent triangle (4,5,6). In the latter case, it is necessary to subtract 4 to determine the edge that really will be flipped. Size $2n + 1$.

Next we explain in detail each step of the algorithm.

1.1 Location step

For each point p the triangle `ContainingTriangle`$[p]$ is updated as follows: If point P=`Point`$[p]$ has not yet been inserted into the triangulation (`Inserted`$[p] = 0$), a walking process is launched until the triangle t that really contains P is reached. If P lies on an edge, t is taken as the triangle adjacent to the edge with the lowest index. Then, `ContainingTriangle`$[p]$ is updated with t and `VertexToInsert`$[t]$ is updated with p. Note that `VertexToInsert`$[t]$ can be updated simultaneously by distinct processors. In this manner, `VertexToInsert`$[t]$ contains the last point reaching t.

1.2 Insertion step

For each triangle t, the point of index p =`VertexToInsert`$[t]$ is inserted into the triangulation as follows: Let p_i =`Triangles`$[3t + i]$ ($i = 0..2$) be the vertex indices of the triangle t. Triangle t will be triangulated to triangle t with vertex indices p_0, p_1 and p, triangle $2p + 1$ with vertex indices p_1, p_2 and p, and triangle $2p + 2$ with vertex indices p_2, p_0 and p. To properly determine the neighbours of these three new triangles, this step needs to be subdivided into two parts.

In the first part, for each triangle t with a point to be inserted p, the future neighbours of the triangles `Neighbours`$[3t + 1]$ and `Neighbours`$[3t + 2]$ are updated according to the insertion of p in t.

In the second part, for each triangle t, if t has a point p to be inserted, the arrays `Triangles`, `Neighbours` and `Flip` corresponding to the triangles t, $2p + 1$ and $2p + 2$ are updated according to the insertion of p in t. Otherwise, the neighbours of t are simply updated with the triangles previously stored in `FutureNeighbours`$[3t..3t + 2]$.

```
foreach $p < n + 3$ do                    /* in parallel */
    if $Inserted\ [i] = 0$ then
        $t$ =ContainingTriangle[$p$];
        $P$ =Point[$p$];
        Found=false;
        while $Found=false$ do
            $[P_0, P_1, P_2]$ = Points[Triangles[$3t..3t + 2$]];
            if $P$ contained in some segment $P_jP_{j+1}$
            and Neighbours[$3t + j$]<$t$ then
                Found=true; $t$=Neighbours[$3t + j$];
            else if $P$ contained in triangle $P_0P_1P_2$
            then Found=true;
            else
                $C = (P_0 + P_1 + P_2)/3$;
                Seek for $j$ such that segment $Cp$
                intersects segment $P_jP_{j+1}$;
                $t$ =Neighbours[$3t + j$];
        ContainingTriangle[$p$]=$t$;
        VertexToInsert[$t$]= $p$;
```
Algorithm 1: LOCATION

```
foreach $t < 2n + 1$ do                   /* in parallel */
    $p$=VertexToInsert[$t$];
    if $i \neq -1$ then foreach $j = 1..2$ do
        $t'$=Neighbours[$3t + j$];
        $j'$=opposite($j$);
        FutureNeighbours[$3t' + j'$]=$2p + j$;
```
Algorithm 2: INSERTION. PART 1.

```
foreach $t < 2n + 1$ do                   /* in parallel */
    $p$=VertexToInsert[$t$];
    if $i \neq -1$ then
        Inserted-P[$i$]=1;
        $[p_0, p_1, p_2]$=Triangles[$3t..3t + 2$];
        $[n_0, n_1, n_2]$=FutureNeighbours[$3t..3t + 2$];
        $[f_0, f_1, f_2]$=Flip[$3t..3t + 2$];
        Triangles[$3t..3t + 2$]=$[p_0, p_1, p]$;
        Triangles[$3(2p + 1)..3(2p + 1) + 2$]=$[p_1, p_2, p]$;
        Triangles[$3(2p + 2)..3(2p + 2) + 2$]=$[p_2, p_0, p]$;
        Neighbours[$3t..3t + 2$]=$[n_0, 2p + 1, 2p + 2]$;
        Neighbours[$3(2p + 1)..3(2p + 1) + 2$]=
        $[n_1, 2p + 2, t]$;
        Neighbours[$3(2p + 2)..3(2p + 2) + 2$]=
        $[n_2, t, 2p + 1]$;
        Flip[$3t..3t + 2$]=$[f_0, -1, -1]$;
        Flip[$3(2p + 1)..3(2p + 1) + 2$]=$[f_1, -1, -1]$;
        Flip[$3(2p + 2)..3(2p + 2) + 2$]=$[f_2, -1, -1]$;
    else
        Neighbours[$3t..3t + 2$]=
        FutureNeighbours[$3t..3t + 2$];
```
Algorithm 3: INSERTION. PART 2.

1.3 Marking step

In this step the edges corresponding to a PSLG segment and the candidate edges to be flipped are marked. An edge can be a candidate due to being crossed by a segment or being non-locally Delaunay. Accordingly, this step is subdivided into two parts:

In the first part, for each non inserted segment s whose both endpoints have already been inserted into the triangulation, if the endpoints of s are connected by an edge, this edge is marked as constrained. Otherwise, a walking process starting from the first endpoint is launched until a flippable edge crossed by s and not flipped in the previous flipping step is found. Then, this edge and its opposite edge are marked to be flipped. The same is done starting from the other endpoint.

```
foreach $s < m$ do                                  /* in parallel */
    if Inserted-P[s]=0 and Inserted-P[Segments[2s]]=1 and Inserted-P[Segments[$2s+1$]]=1 then
        for $j = 0..1$ do
            if $j = 0$ then
                [$i_0, i_1$]=Segments[$2s..2s+1$];
            else [$i_1, i_0$]=Segments[$2s..2s+1$];
            [$P_0, P_1$]=Points[$i_0..i_1$];
            starting from ContainingTriangle[$i_0$] seek for the triangle of index $t$ incident to $i_0$ intersecting segment $P_0P_1$;
            if $P_1$ is a vertex of $t$ then
                $j$=edge of $t$ connecting $P_0$ and $P_1$;
                $t'$=Neigbours[$3t+j$];
                $j'$=opposite($j$);
                Flip[$3t+j$]=Flip[$3t'+j'$]=2;
                Inserted-P[$s$]=0;
            else
                Found=false;
                while Found=false do
                    $j$=edge of $t$ intersecting segment $P_0P_1$;
                    if $j$ is flippable and different to EdgeToFlip[t] then
                        Found=true;
                        $t'$=Neigbours[$3t+j$];
                        $j'$=opposite($j$);
                        Flip[$3t+j$]=Flip[$3t'+j'$]=3;
                    else
                        $t$=Neigbours[$3t+j$];
```

Algorithm 4: MARKING. PART 1.

In the second part, for each triangle t not crossed by any segment (`Flip`[$3t$] mod 2+`Flip`[$3t+1$] mod 2+`Flip`[$3t+2$] mod 2=0) the non constrained edges that are non Delaunay are marked to be flipped.

```
foreach $t < 2n+1$ do                               /* in parallel */
    if $\sum_{k=0}^{2}(Flip[3t+k] \; mod \; 2) = 0$ then for $j = 0..2$ do
        if Flip[$3t+j$]$\neq 2$ and no Delaunay(j) then
            Flip[$3t+j$]=1;
```

Algorithm 5: MARKING. PART 2.

1.4 Flipping step

During this step, at most one edge of each triangle t is flipped. To avoid conflicts between concurrent flips, this step needs to be subdivided into three parts.

In the first stage, a marked edge whose opposite edge is the only marked edge in its triangle t' is sought. If this edge exists, let nc be the number of marked edges of t. Then, if $nc \geq 2$ or ($nc = 1$ and $t < t'$) the edge is stored in `EdgeToFlip`[t].

In the second stage, if t has an edge to be flipped (`EdgeToFlip`[t]≤ 2), the future neighbours of the triangles adjacent to the quadrilateral determined by the edge are updated accordingly to the future flip of the edge. Otherwise, if the opposite edge of an edge j of t is to be flipped, $4+j$ is stored in `EdgeToFlip`[t].

In the third stage, if t has an edge to be flipped (`EdgeToFlip`[t]≤ 2), the arrays `Triangles`, `Neighbours` and `Flip` corresponding to the triangles t and t' are updated according to the flip. Otherwise, if any adjacent triangle has an edge to be flipped, the array `Neighbours` is updated.

```
foreach $t < 2n+1$ do                               /* in parallel */
    EdgeToFlip[$t$]=3;
    $nc$=$\sum_{k=0}^{2}$(Flip[$3t+k$] mod 2);
    if $nc > 0$ then
        $j = 0$;
        Found=false;
        while $j \leq 2$ and no Found do
            if Flip[$3t+j$]=1 then
                $t'$=Neigbours[$3t+j$];
                $j'$=opposite($j$);
                $nc'$=$\sum_{k=0}^{2}$(Flip[$3t'+k$] mod 2);
                if Flip[$3t'+j'$]=1 and $nc'=1$ and ($nc \geq 2$ or ($nc = 1$ and $t < t'$)) then
                    EdgeToFlip[$t$]=$j$;
                    Found=true;
            $j++$;
```

Algorithm 6: FLIPPING. PART 1.

2 Results

The algorithm was implemented using OpenCl on a computer equipped with an Intel(R) Pentium(R) D CPU 3.00GHz, 3,5GB RAM and a GPU NVidia GeForce GTX 580/PCI/SSE2. Each one of the algorithm's parts was written in a kernel. The algorithm was executed ten times on two different models (Chairs32×32 and Holland4×4) and compared with *Triangle* [3], one of the most popular computational geometry software. The model Chairs32×32 is formed by 1024 copies of the model showed in the Figure

```
foreach $t < 2n+1$ do                                /* in parallel */
    if $EdgeToFlip[t] \leq 2$ then
        $j$=EdgeToFlip[$t$]; $j1=j+1$ mod 3;
        $n$=Neighbours[$3t+j1$]; $nj1$=opposite($j1$);
        FutureNeighbours[$3n+nj1$]=$t'$;
        $t'$=Neighbours[$3t+j$]; $j'$=opposite($j$);
        $j1'=j'+1$ mod 3;
        $n'$=Neighbours[$3t'+j1'$]; $nj1'$=opposite($j1'$);
        FutureNeighbours[$3n'+nj1'$]=$t$;
    else
        for $j=0..2$ do
            $t'$=Neighbours[$3t+j$];
            if $EdgeToFlip[t'] \leq 2$ and
            $Neighbours[EdgeToFlip[t']]=t$ then
                EdgeToFlip[$t$]=$4+j$;
```

Algorithm 7: FLIPPING. PART 2.

```
foreach $t < 2n+1$ do                                /* in parallel */
    if $EdgeToFlip[t] \leq 2$ then
        $j$=EdgeToFlip[$t$];
        $t'$=Neighbours[$3t+j$]; $j'$=opposite($j$);
        $j1=j+1$ mod 2; $j2=j+2$ mod 2;
        $j1'=j'+1$ mod 2; $j2'=j'+2$ mod 2;
        $p2$=Triangles[$3t+j2$]; $p3$=Triangles[$3t'+j2'$];
        Triangles[$3t+j1$]=$p3$; Triangles[$3t'+j1'$]=$p2$;
        ContainingTriangle[$p3$]=$t$;
        ContainingTriangle[$p2$]=$t'$;
        $n1$=FutureNeigbours[$3t+j1$];
        $n2$=FutureNeigbours[$3t+j2$];
        $n1'$=FutureNeigbours[$3t'+j1'$];
        $n2'$=FutureNeigbours[$3t'+j2'$];
        Neigbours[$3t+j$]=$n1'$; Neigbours[$3t+j1$]=$t'$;
        Neigbours[$3t+j2$]=$n2$;
        Neigbours[$3t'+j'$]=$n1$;
        Neigbours[$3t'+j1'$]=$t$;
        Neigbours[$3t'+j2'$]=$n2'$;
        FutureNeighbours[$3t..3t+2$]=
        Neighbours[$3t..3t+2$];
        FutureNeighbours[$3t'..3t'+2$]=
        Neighbours[$3t'..3t'+2$];
        if $Flip[3t+j]=3$ then EdgeToFlip[$t$]=$j1$;
        else EdgeToFlip[$t$]=3;
    else if $EdgeToFlip[t] = 3$ then
        Neighbours[$3t..3t+2$]=
        FutureNeighbours[$3t..3t+2$];
    else
        $j$=EdgeToFlip[$t$]-4; $j1=j+1$ mod 2;
        if $Flip[3t+j]=3$ then EdgeToFlip[$t$]=$j1$;
        else EdgeToFlip[$t$]=3;
    for $j=0..2$ do if $Flip[3t+j] \neq 2$ then
    Flip[$3t+j$]=0;
```

Algorithm 8: FLIPPING. PART 3.

1 arranged in a 32 by 32 array, while the model Holland4×4 is formed by 16 copies of the model showed in the Figure 2 arranged in a 4 by 4 array. The mean running times are presented in Table 1. Our future task is to study the performance of our algorithm versus [2]. However, our approach does not make use of the huge memory space needed by [2] to store the required digital Voronoi diagram.

	Chairs32×32	Holland4×4
Num. Vertices	1444864	1015344
Num. Segments	739328	1007488
Mean time (ms)	4028	2306
Triangle (ms)	12835	10254

Table 1: Behaviour of the proposed algorithm.

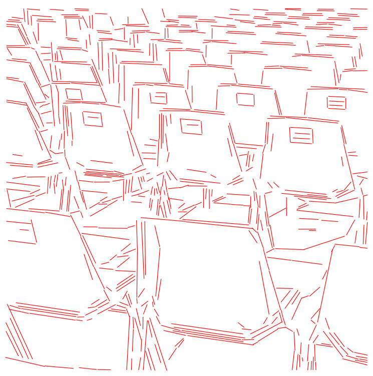

Figure 1: Cell of the model Chairs32×32

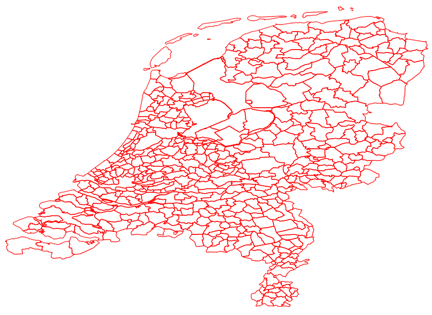

Figure 2: Cell of the model Holland4×4

References

[1] N. Coll, M. Guerrieri, Parallel Delaunay triangulation based on Lawson's incremental insertion, in: *Proceedings of the XIV Spanish Meeting on Computational Geometry*, CRM Documents, 8, Centre de Recerca Matemàtica, Bellaterra (Barcelona), 2011, 169–172.

[2] M. Qi, T. Cao, T. Tan, Computing 2D constrained Delaunay triangulation using the GPU, in: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, ACM, New York, NY, USA, 2012, 39–46.

[3] http://www.cs.cmu.edu/ quake/triangle.html.