Utrecht University

Master Thesis (ICA-3565025)

Computing High Resolution Explicit Corridor Maps using Parallel Technologies

Author:
R. Bonfiglioli

Supervisor:
dr. R. J. Geraerts

July 22, 2013

Abstract *— This work investigates the approximated construction of Explicit Corridor Maps (ECMs). An ECM is a type of Navigation Mesh: a geometrical structure describing the walkable space of an environment that is used to speed-up the path-finding and crowd-simulation operations occurring in the environment. Additional geometrical routines that take advantage of the GPGPU model are presented, which improve the current construction method by increasing the number of computations performed on the GPU and reducing the amount of data transferred back to the CPU. At the same time, a multi-tiled construction approach is presented, which almost frees the geometrical computation from its current main constraint, resolution, allowing high-resolution ECMs to be produced; moreover, we show that this approach can benefit optimally from CPU-parallelism.*
Both a GPGPU-aided and a CPU-parallel implementation are tested: experiments show that high-resolution ECMs can be computed, and that the new implementations often outperform the original one.
Acknowledgements. *This research has been supported by the COMMIT/ project (http://www.commit-nl.nl/).*
Keywords: Crowd Simulation, Navigation Mesh, Explicit Corridor Map, GPGPU.

Computing High Resolution Explicit Corridor Maps using Parallel Technologies

Rudi Bonfiglioli
Game and Media Technology Master Thesis, Utrecht University

July 22, 2013

Contents

1 Introduction

An open problem in today's simulations and games is pathfinding. Pathfinding is the basic process that allows agents to effectively navigate a virtual environment, and modern applications need extremely efficient pathfinding routines in order to simulate a high number of characters at interactive rates. Pathfinding solves the basic problem of highlighting the path from one point to another of a map given its description in any form, by listing the ordered sequence of points that have to be traversed. Typically, the description can be a graph or a grid, and many popular search algorithms exist (Dijkstra, A*, D*-lite [KL02]) to solve this problem. However, modern applications deal with complex or with high-quality environments which often can not be described efficiently by a simple graph or a grid. Therefore, the problem of simultaneous computing a proper Navigation Mesh given an environment is introduced: a Navigation Mesh aims at being a compact but rich description of (mostly) the navigable space of an environment, and it will be used to perform pathfinding queries. A Navigation Mesh can be computed at the beginning of the simulation, or even during an off-line phase and stored, and if necessary, it must be modifyable at run-time. A Navigation Mesh subdivides (explicitly or implicitly) an environment into a number of different zones, the union of which will cover the navigable space of the input environment. Information about zones can be "extended", for example by associating an additional value representing the maximum height of the zone, or some kind of value representing the density of agents. How to organize a Navigation Mesh, which information to store and how to update it are all open problems, and they can all greatly influence the quality and the performance of the simulations.

Standard pathfinding algorithms can still be applied to the mesh by using a graph structure "derived" (more or less directly) from it; the nodes of the graph can either lie inside the zones or coincide with (some) vertices of the polygons forming the Mesh. Usually, Navigation Meshes (and the related graphs) deal with a top-down view of the environment (footprint) and thus compute a 2D structure than spans the free/walkable space. The graph associated with such a Mesh that will be queried by the pathfinding routines is sometimes called "Road Map" of the environment, because it intuitively defines some walkable routes that can be used for navigation.

1.1 Project Motivation

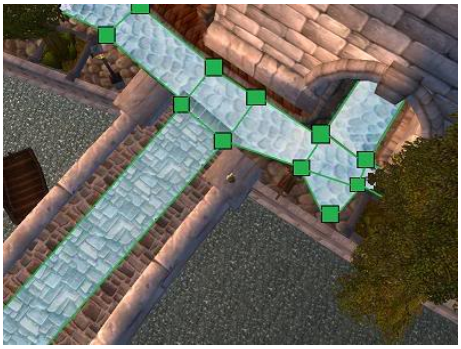
The Explicit Corridor Map [Ger10] (ECM) is a known method that automatically generates a Navigation Mesh from the description of a 2D environment: the ECM is a compact graph that spans the walkable space of the environment. It incorporates information about closest points on the obstacles to each node in the graph and thus provides clearance information and subdivides the walkable space into polygons at the same time. In turn, the ECM is based on the ideas of Medial Axis (and Voronoi diagram), and requires those geometrical structures during construction. Computing a Medial Axis is known to be a computationally expensive operation; since efficiency is a key factor, the ECM employs an approximated computation of a Medial Axis that is dependent on the "resolution" parameter, which correlates with the degree of approximation that will take place. A higher resolution ECM could provide a more detailed description of the input environment but at the cost of increasing computational times. Moreover, a limit exists on the maximum resolution value we can use. Since the resolution in fact derives from the dimensions of the grid that is used during one of the steps of the computation, we can express it with the number of total cells (or pixels) used. At the same time, the input environment mostly uses "meters" as unit of measure, therefore we can describe the precision of an ECM construction in terms of pixels used per squared meter of the environment: in general, to have a detailed ECM we would like to achieve between 20 and 40 pixels per meter.

In this work, we will try to extend the ECM construction approach so that it becomes less dependent on the resolution parameter, and on the related limitations. Practically, we will try to take advantage of parallel technologies, both consolidated ones, like CPU parallelism, and more emergent ones, like GPGPU. Hopefully, our new approach will always be able to

(a) A "waypoint graph", a simple but inefficient description of the walkable space

(b) An example of Navigation Mesh for the areas: if we consider the graph having a vertex per rectangle, we realize how more efficient it is

(c) We can find a path using the Navigation Mesh and applying a natural smoothing



(d) A better example of the use of a Road Map

Figure 1: Navigation structures in a Virtual Environment. Images taken from the online article `http://www.ai-blog.net/archives/000152.html`

produce an ECM at a sufficient resolution no matter what the size of the environment is, and without excessive costs in terms of computational times.

1.2 Project Goals

We are now ready to state the main goals of the project.

- We will analyze the current ECM construction approach, and we will try to shape a new approach that is not constrained by the current limitation on the resolution parameters
- We will investigate whether the new approach can take advantage of CPU parallelism, and how. Eventually, we will produce a new CPU-Parallel implementation that will incorporate the new ideas
- We will then do the same with GPU parallelism (GPGPU)

In the end, we will test the two new implementations and we will perform the right comparisons with the original one.

1.3 Structure of this Work

This work is organized in the following way:

- Chapter 2 introduces some preliminaries and fundamental concepts, together with the related definitions that will be required throughout the rest of the work.
- Chapter 3 introduces and analyzes the current ECM construction approach together with some alternatives
- Chapter 4 introduces GPGPU, and investigates how it could be employed in the construction of an ECM. First, some relevant literature is presented, then some conclusions are drawn and finally some GPGPU-powered routines that perform geometrical computations are shown. We introduce GPGPU first because the current ECM approach already performs some computations on the GPU. Therefore, using GPGPU could appear as a more natural extension
- Chapter 5 introduces a new approach that aims at removing the constraints related to the total resolution by designing a multi-tiled construction. First, the logical and geometrical ideas behind the approach are outlined. Then, we proceed with describing how CPU parallelism could fit this new approach
- Chapter 6 presents and discusses the results of the various tests performed with the two new implementations and the original one
- Chapter 7 wraps up the work by evaluating the benefits and limitations of the new implementations, and by outlining possibilities for future work

2 Preliminaries

In this chapter we will present some preliminary ideas that will serve as a foundation for the rest of the work.

2.1 Generalized Voronoi Diagram

A Voronoi diagram (also called Dirichlet tessellation, or Voronoi tessellation) is a fairly widespread concept, although, as we will see, slightly different definitions for it are present in literature. After having identified a set of objects E in the space that are called "seeds" sites, or "generators", usually for an $e \in E$ both a "Voronoi region" $VR(e)$ and a "Voronoi diagram" $VD(E)$ of E are defined. One possible definition is:

Definition 2.1. *The Voronoi region of a site e of E, denoted by $VR(e)$, is the set of points at least as close to e as to any other site in E. In symbols:*

$$VR(e) = \{p \in R^n | d(p,e) \leq d(p,e'), \forall e' \neq e, e' \in E\}$$

The Voronoi diagram $VD(E)$ then becomes:

$$VD(E) \doteq \bigcup_i \partial VR(e_i)$$

where d is a pre-existing distance metric and ∂ denotes the boundary of a set. More simply, a so defined Voronoi region is a region of the space consisting of points p that are closer to a single site e (the closest site) than to any other site; if this distance is $d(p,e)$, the distance from p to any other site will be equal to $d(p,e)$ or greater.

Definition 2.1 can be found in [FEC02], a paper that aims (also) at discussing some fundamental properties of Voronoi diagrams, but it can be commonly encountered in other works such as [Aur91] and [BA92]. It is easy to notice that some points on the boundaries of the regions will have 2 closest sites, while some will have even more than 2: both these kind of points belong to $VD(E)$ but the latter kind of points will be the vertices of the Voronoi diagram (also called Voronoi vertices), contact points of 2 or more Voronoi regions. Instead, the points that are part of $VD(E)$ but are not vertices will form the edges (also called Voronoi edges) connecting those vertices, and will have 2 associated closest sites. Hence, a Voronoi diagram can be geometrically described (and stored) as a graph where the nodes are our Voronoi vertices and the edges the Voronoi edges connecting them.

Figure 2 shows a simple case of Voronoi Diagram where the generating seeds are points in the space: Voronoi edges and vertices can easily be recognized. Moreover, in the picture some additional properties of the tessellation are easy to spot:

- The set of Voronoi edges is made of only bisector segments, because each point of a Voronoi edge is equidistant from 2 generating seeds.
- If the set S of the generating seeds has n elements, the Voronoi Diagram will have n edges and vertices, and therefore can be stored with linear complexity $O(n)$.
- Voronoi edges form a connected component if the free space is also connected, therefore a Voronoi Diagram can be effectively used as a Road Map [GO08].

Although our first definition of a Voronoi diagram was general, Picture 2 depicts a fairly simple case of Voronoi diagram, where:

- Seeds are points.
- We assume the space to be Euclidean, and we use Euclidean distance as a distance metric.
- The space is two-dimensional.

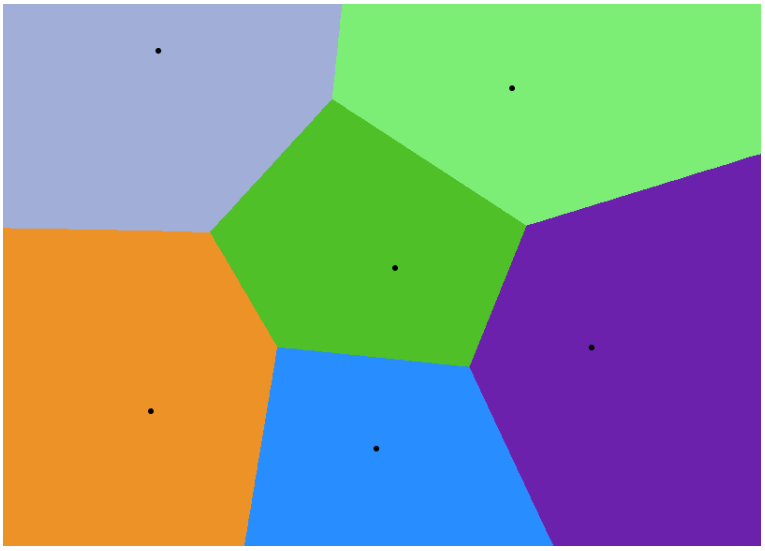


Figure 2: Example of Voronoi Diagram. Black points are the generating sites. Points with the same color belong to the same cell.

However, Voronoi diagrams can be described in a general d-dimensional space and can be generated by seeds which are not just points but segments, polygons, or conics, and different distance metrics can be used, such as a 2D Manhattan distance. When the previous restrictions do not apply, the term "generalized Voronoi diagram" is normally used to refer to the generated Voronoi diagram. In a d-dimensional space, a Voronoi diagram with n generators can be stored with $O(n^{\lceil d/2 \rceil})$ complexity, meaning that in spaces with a dimensionality higher than 2 it could be considered not efficient anymore.

Since this work will make use of 2D Voronoi diagrams where sites that are not just points, it can be useful to outline the general shape of the bisectors in this scenario, depending on the type of the generating site:

- **Lines and segments**: if we consider lines (endless extended), a bisector of two lines is a line that evenly splits the angle between them (Figure 3a) in case of non parallel lines, or if they are parallel, the bisector is another parallel line lying "between" them at the same distance from the generating ones. In case of non parallel lines, the bisectors are actually two, both running through the intersection points (Figure 3b). The bisector of a line and a point is a parabola (Figure 3c), which is in fact commonly defined as the set of point equidistant from a point and a non-intersecting line. In case of line segments, things are trickier, because bisectors can be generated by the segment itself and its two endpoints. Therefore, the bisector of two non-intersecting line segments can contain up to seven parts, either line segments or parabolic arcs. The topology of the part of the bisector changes when it "crosses" the normal vector at a segments' endpoint (Figure 3d).

- **Polygons**: most of the time, we can consider them a closed chain of edges, and treat them as a set of line segments, with non-convex polygons being split into convex ones.

- **Conics or curved segments**: closed or non-closed curved segments seem to be quite complicated to handle, even in the two-dimensional case. In many cases, then, they are approximated to polygons, eventually not losing too many details. The ECM implementation handles them this way.

However, even when generated from more complex scenarios than the one shown in Picture 2, a generalized Voronoi diagram still maintains all the properties listed previously. Hence, we will see how it can be a fairly viable option we can use to create a Road Map of our environment starting from many different footprints.

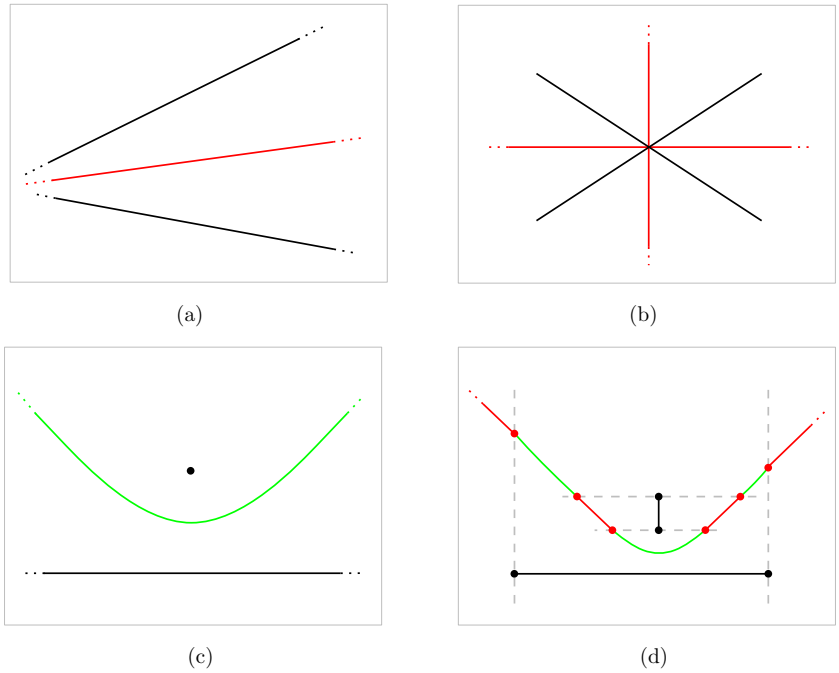


Figure 3: Bisectors in various scenarios.
(a,b) Two non-parallel lines have two linear bisectors (in red), cutting the two angles in between in halves.
(c) The bisector of a point and a line is a parabola (in green).
(d) Two line segments can induce up to seven connected bisector parts: lines (red) and parabolic arcs (green).
Red dots indicate the intersection of a bisector with a normal vector.

We had initially mentioned that the definitions of Voronoi diagrams can differ slightly among the literature. For example, we could employ the following one:

Definition 2.2. *A Voronoi region of a site $VR(e)$ can be defined as*

$$VR(e) = \{p \in R^n | d(p,e) < d(p,e'), \forall e' \neq e, e' \in E\}$$

which defines a region when points are strictly *closer to a site e than to any other. Then a Voronoi diagram $VD(E)$ of the set E will be defined as*

$$VD(E) \doteq \left[\bigcup_i VR(e_i)\right]$$

Definition 2.2 defines $VD(E)$ as the region of the plane complementary to the union of all the previously defined Voronoi regions. It is not difficult to see that all the properties previously discussed still hold for the so generated constructions, but this different definition can have different implications, as we will see int the next sections.

2.2 Medial Axis

As stated also by [FEC02], a common definition for the Medial Axis (MA) is

Definition 2.3. *Consider S as a subset of the n-dimensional Euclidean space and its boundary B.*
An open ball inside B not contained in any other open ball inside B is called a maximal ball.
The Medial Axis of S is defined as the locus of centers of its maximal balls.

Basically, the MA is the locus of the centers of circles that are tangent to S in 2 or more points, with all such circles not "crossing" S at additional points. If S is a simple polygon,

the MA is made only of line segments or parabolic arcs. If the polygon is convex, the MA is completely inside the polygon.

Once again, literature [BA92, FEC02, Aur91] offers different definitions for the Medial Axis, and for example we could employ the following one:

Definition 2.4. *Consider S as a subset of the Euclidean n-space. An open ball B is said to be a maximal ball of S if it satisfies the two following properties:*

- *B belongs to the set G of the open balls that does not have any points of S*
- *B is not a proper subset of any other ball in G*

Definition 2.4 allows to handle some kind of inputs, for example open shapes or disconnected sets of points, which will not have an associated MA according to Definition 2.3. At the same time, all the properties previously outlined (e.g. when input is a simple polygon) are maintained.

Intuitively, if we consider the walkable space of an environment being our polygon P and add the constraint of this space being bounded, the MA of P will be very similar to the set of Voronoi edges and Voronoi vertices of P: Voronoi vertices are clearly centers of empty balls, and points on the Voronoi edges are too since they are equidistant from 2 points on P. Therefore, they are empty balls being tangent to P on those 2 points. But the generalized Voronoi diagram and the Madial Axis of a polygon P do not always coincide. Equality depends on:

- The definitions used for both concepts. Definition 2.1 and Definition 2.2 can lead to 2 different generalized Voronoi diagrams of the same input S when S is a subset of the Euclidean n-space. The same can happen if the input is a polygon, depending on how the input sites are defined. This of course can prevent the Medial Axis of the same input to coincide with the (generalized) Voronoi diagram. Here, it is important to notice that, when it comes to Medial Axes, Definition 2.4 allows to construct Medial Axes of some configurations that are not handled by Definition 2.3, but if Medial Axes are produced according to both definition, they should not differ.
- The way P is defined. Until now, we assumed that segments that are part of our polygons are defined as single segments' entities and not, for example, as discrete collections of points, all being very close to each-other. In the latter case, Definition 2.1 leads to a generalized Voronoi diagram that is different from the Voronoi diagram generated from the former case. This happens because generalized Voronoi diagrams allow arbitrary aggregation of points to form the "generating sites", therefore non-local characteristics are introduced, since the evaluation of proximity (performed during the construction of the regions) takes into account the whole extension of the sites and not just specific portions. On the opposite, a Medial Axis is alway "local", being intrinsic to the data set independently of any grouping. If the sites of the input are defined as a connected set of discrete points, the Voronoi diagram is constructed using Definition 2.2 and a MA exists for that same input, then they coincide [FEC02].

If our input set is a polygon *not* specified as a set of connected, discrete points, its generalized Voronoi diagram constructed according to Definition 2.1 or Definition 2.2 usually includes some edges that are not present in the MA of the same input. Similarly, if the input is a polygon and is specified as a set of connected, discrete points, but Definition 2.2 is *not* employed, the resulting Voronoi diagram will present additional edges not part of the MA. Figure 4 shows an example of this.

In conclusion, the Medial Axis of a 2D footprint is a generally better defined concept than the one of (generalized) Voronoi diagram that depends only on local properties of the input set, and it is the concept on which our Navigation Mesh structure will be built upon.

2.3 Explicit Corridor Map

Published in 2007, [GO07] introduces the *Corridor Map Method* (CMM) for path planning of many agents simultaneously: this method is based on the notion of "corridor", a subset

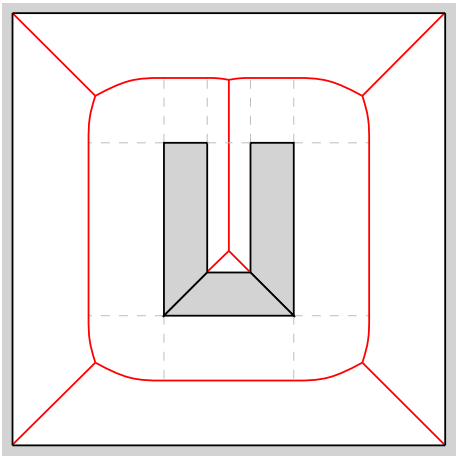


Figure 4: The medial axis (in red) given a 2D environment (with black borders). Obstacles (or holes in the "free space" polygon) are shown in gray and outlined in black. Dashed lines indicate edges that can occur in a Generalized Voronoi Diagram of the same environment, depending on its definition.

of the collision-free walkable space which is in fact the union of many, collision-free balls centered on some points inside it. More in detail, given a 2D footprint, a "backbone graph" G is constructed, composed of a set of edges E and of vertices V. The edges of the backbone graph will be sampled (divided into a finite and discrete sequence of points) and for each point the radius of the largest, collision-free ball will be computed: for each $e \in E$, the sequence of points with the associated radii of the ball will represent a corridor. Similarly, a vertices $v \in V$ will be associated with the radius of the largest collision-free ball centered on it.

In the CMM, a path-finding query is answered by first connecting both the starting point and the ending point of the query belonging to the walkable space to the closest vertices $v_1, v_2 \in V$ of the backbone path with a short, collision-free path; then, a global path can be found on the graph itself by employing any existing path-finding algorithm working on graphs such as A*. The so defined paths will be guaranteed to be collision-free. Characters can be steered by using the sampled points of the corridors as attraction points, and thus applying attractive forces that will move them towards their goals. Moreover, corridors encode a clearance information: clearance can be used to add variation/smoothing to characters' paths, or to allow characters to avoid local obstacles inside the corridor, for example by making sure that local obstacles apply repulsive forces to moving agents. If the edge that is being followed by a character contains a sufficient number of sampled points, this, together with the radii information, seems to be enough to ensure smooth, collision-free paths that correctly avoid local obstacles.[GO07] defines this whole framework and tests it, finding out that it is capable of computing these paths quite efficiently.

In 2010 [Ger10] introduced the Explicit Corridor Map (ECM) approach. As described in the previous section, the CMM approach did not specify a unique way to compute the needed backbone graph: [GO07], the original paper introducing CMM, cited the "Enhanced Reachability Roadmap" [GO06] method as an option, while [GO08], an updated and improved description of the CMM, proposes the use of the Medial Axis, but the framework is stated generally enough to still allow different algorithms. Instead, with the ECM, corridors are built "around" a Medial Axis of the footprint, chosen as a backbone graph, and these edges are not sampled anymore because the computation of the balls forming the corridors is not needed: all the points belonging to the same edge share the same clearance information, therefore storing them once is sufficient. This results in a much more compact structure (with $O(n)$ storage complexity, n being the number of "event points" retrieved from the MA), more efficient to store and navigate, but coming with no additional drawback.

In [Ger10] it is shown that the edges of the Medial Axis include all the information relevant for any path-finding operation, therefore we can use it as a starting point for our "backbone graph". Nonetheless, these edges carry with them useful information because

they "signal" a change of the closest generating site. In the ECM, edges are defined as sequences of "event points": an event point is a point on an edge where a change of closest points information happens. Every vertex of the Medial Axis is an event point, but edges can exhibit more than 2 event points, because the change of closest point information coincides with a change in the associated input segments' normal (bottom-right part of Figure 3 can be a reference here). For each event point of any edge, the ECM stores the (relative) left and right closest points, together with the two distances. The structure should be clearer when looking at Picture 5: at any point on an edge, we can query the distance from the closest left and right obstacle, having therefore the same clearance information necessary to define our corridor, but without the need of storing an expensive set of sampled points and radii for each of them. In fact, considered that:

- Any obstacle will have a finite number of vertices m that will lead to at most m event points,
- every obstacle can generate at most two edges,
- the storage complexity of a Medial Axis is linear in the number of obstacles' vertices n,

we can conclude that an ECM is an efficient data structure having storage complexity $O(n)$, linear in the number of obstacles' vertices.

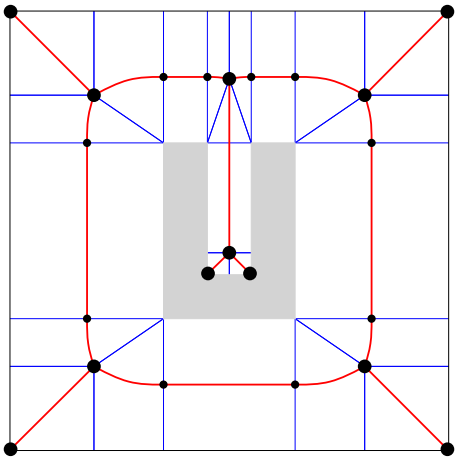


Figure 5: Example of an Explicit Corridor Map. For each edge of the ECM (in red) we store its event points (smaller black dots) and its vertices (bigger black dots). In fact, vertices are event points themselves. Together with vertices and event points we store the closest points (points connected in blue with event points) and the associated distances. Closest points can range from 2 to 4. Notice that the "shape" of edges is effectively defined by the event points and closest points information only (f.e: we have a parabolic arc between two event points if from one side we have 2 closest points and only one on the other side), thus that is the only thing we need to store.

3 Current construction of an ECM

In the previous chapter we have presented the Explicit Corridor Map and the geometrical ideas related to it, like the Voronoi diagrams and Medial Axes. In this section we will discuss some construction algorithms, outlining their advantages and disadvantages.
Given the 2D footprint of an environment, an ECM can be derived from the Medial Axis of the same footprint: we have already discussed how Definition 2.3, and even better Definition 2.4, allow to construct Medial Axes that can be "augmented" into ECMs which are optimal in terms of storage complexity. Vertices of the MA do not always become vertices of the ECM: some of them "signal" a change of closest points' information, but not a change among the associated generating obstacles, and therefore are just stored as event-points on the edges of an ECM. This happens because our input set can be made of polygons, which are grouped in sets of segments and points. We recall that the MA is always "local", therefore an event-point of an ECM corresponds to a "local" change in the MA, and thus with new associated closest points, but those points can lie on segments which are part of the same input site (for example a polygon). If there is no change in the obstacles (for example: the polygons) that generated the point, the point will be considered only an event point (although with new closest points information) and not a vertex of the ECM. Moreover, the geometrical description of the MA provides us with all the necessary points that we need to construct our ECM, no matter if they will be vertices between edges or event points. Only, we have to go through all of them, check their associated generating sites, and compute the closest points information on those obstacles. Hence, constructing an ECM from a MA is a computationally efficient operation.

Hence, one way to construct an ECM is to compute a MA of the footprint and then "arrange" it as an ECM. Again, we can remember that the idea of Medial Axis is deeply related to the one of Voronoi diagram, and that in fact by constructing a Voronoi diagram according to Definition 2.4 and from an input specified as a set of connected, discrete points, our structure will coincide with a MA. But even in case we employ a different definition, or if our input set is made of sites that are groups of segments or points (and not only single, connected points), we can still construct a MA from the (generalized) Voronoi diagram: basically, because of the non-local properties that will influence the construction of a Voronoi diagram, additional edges can be constructed. [FEC02] shows that this is a well-known fact and provides an overview of the methods to construct a Medial Axis from a GVD: for example, [Lee82] is cited as a work that shows as a MA can be obtained from a GVD of a polygonal input set. [Lee82] shows that we can obtain a MA by constructing a GVD using Definition 2.1 considering as input an open straight-line edge for each edge of the input and a point for each vertex of the polygon, and then removing from the GVD the edges that incident on the concave vertices of the input. Works as [SN87] and [MS87] extend a similar approach to have it work also with more general polygonal inputs (e.g. with holes). In [Ger10] this same idea is presented, by making clear that the MA used as a basis for an ECM should be pruned by those kind of edges; the paper also claims that they do not represent necessary path-finding information.
Therefore, independently from the definitions that are used to compute Medial Axes or (generalized) Voronoi diagrams and from the way the input is specified, we know how to efficiently construct a MA from the product of the common methods that compute (generalized) Voronoi diagrams. Again, we reduced the problem of computing an ECM to the problem of computing a GVD: having a geometrical description of its regions is now our fundamental step. The next section will elaborate on the computation of this structure, presenting some approaches with common characteristics, discussing their advantages and disadvantages, too.

3.1 Exact and Inexact computation of a Generalized Voronoi diagram

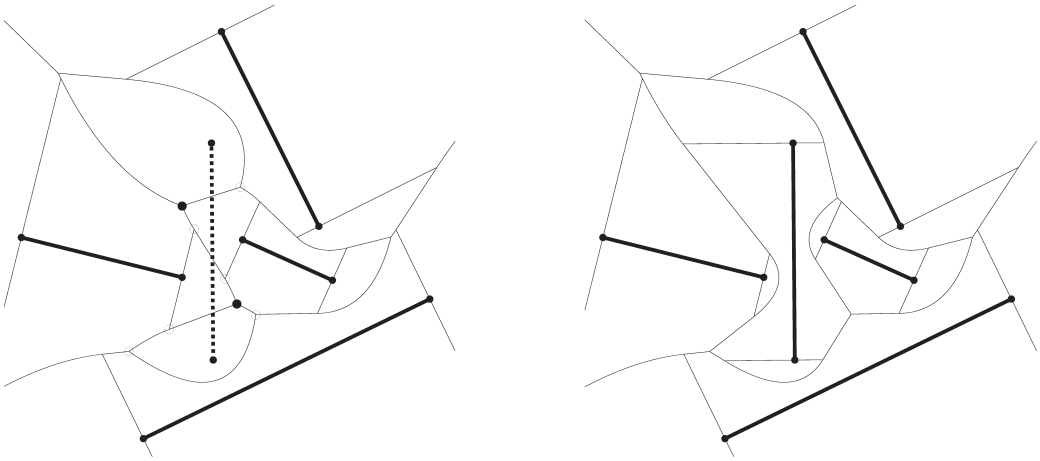
3.1.1 Exact computation of a GVD

In [dBCvKO08] a standard *plane-sweep* algorithm to compute a Voronoi diagram of points can be found: the n points are handled in a sorted order by a line that "sweeps" through

them and handles the so-called "events" that arise, denoting the updates that are performed on the work–in-progress data structures. Since it can be proved that $O(n)$ events arise which can be handled in $O(log(n))$ time, the total complexity of the algorithm becomes $O(nlog(n))$. [dBCvKO08] also shows that the same approach can be extended to compute generalized Voronoi diagrams of non-intersecting line segments, maintaining the same complexity.
Voronoi diagrams can be also constructed using a divide-and-conquer approach, as outlined by [SH75]. Here, the input set is recursively divided into 2 halves, and 2 smaller Voronoi diagrams are constructed from both sides and then merged together. This algorithm also yields a $O(nlog(n))$ complexity, mostly because it requires sorting the sites. However, this algorithm can be often easier to implement robustly.
Another existing approach is to compute a (generalized) Voronoi diagram incrementally, starting from an empty/basic structure and then iteratively insert each site and update the current tessellation; after all the generating sites have been inserted, we will have the (generalized) Voronoi diagram of the entire input set. The insertion of input sites can be randomized, and therefore, as it is common when randomized incremental search is employed, we can reason about expect running times. A fairly advanced randomized incremental construction algorithm which can compute generalized Voronoi diagrams of curved segments, too, with an expected running time of $O(nlog(n))$ can be found in [ACV05].
Although most algorithms that deal with "general" (non-simple, non-convex) polygons have a complexity of $O(nlog(n))$, a theoretical bound is still under discussion, and some work outline fairly complicated algorithms that achieve lower time complexities, such as $O(nlog*(n))$ by [Dev92] or even $O(n)$ by [CSW95]. However, whether those algorithms can lead to stable and manageable implementations has still to be evaluated.
In theory, we could easily adapt all the algorithms described above so that they take advantage of Definition 2.1, or we could pre-process the input to split it into a set of connected points, computing thus a Voronoi diagram that is in fact the Medial Axis we need. Unfortunately, in practice, these exact geometric algorithms rely on exact geometrical computations, especially to handle degenerate cases, and ensuring stable and correct geometrical computations is difficult because of the finite numerical precision of computer. To specifically address these needs, geometric libraries like CGAL [cga] exist: CGAL aims at offering a wide range of common geometric algorithms that can be extended and geometric data structures to work with. The fundamental idea behind CGAL is to abstract geometric operations (such as intersection tests) which will be treated as "exact computations" by the underlying library, exploiting numerical integrations and other techniques, making sure they always return the correct result and preventing overflow/underflow errors at the same time. By using CGAL, the above algorithms could be implemented, and a correct MA could be obtained, but generally the resulting code would be too slow to serve any practical scenario: exact arithmetical operations are quite expensive, and their cost can become prohibitive when the geometrical "degree" of the problem involved is high. A study of the geometrical degree of a Voronoi diagram computation can be found in [LPT98].

More recently, the so-called "topology-oriented" methods drew some attention: a topology-oriented implementation tries to specify a limited set of geometrical properties that must hold at particular stages of the computation, properties that can be at the same time verified without relying on geometrical operations. It can be shown that a topology-oriented approach always leads to a correct result without the requirement of checking if the arithmetic is exact: (correct) geometrical computations are needed only to reach the right results given the specified input. By decreasing the number of operations that require numerical correctness and by formally separating them from the ones that instead do, clean and stable implementation can be obtained. In fact, algorithms are primarily described in terms of combinatorial and topological computations that deal with the properties selected, and numerical computations are used as secondary information. Thus, the inconsistency issue is completely separated from the numerical error issue. Some examples of topology-oriented geometric algorithms can be found in [SIII00], which also includes a brief description of an algorithm to compute Voronoi diagrams of set of points. Upon this last method, the VRONI approach was built [Hel01]: the motivation behind VRONI is exactly to develop a

stable implementation capable of computing correct Voronoi diagrams of sets of points fairly efficiently, since it became clear how producing such an implementation was harder than it seemed. VRONI adopts definition 2.1 and constructs tessellations incrementally while continuously checking for the presence of degenerate or "troublesome" topological cases, which are then correctly handled; numerical sanity checks are also constantly performed, and the system is designed to run "back-up" geometrical routines every time any of the standard ones fail, and to automatically choose and relax the ϵ constraint bounding the geometrical routines. Notice that VRONI, as the other topology-oriented approaches, aims at computing correct structures using the standard float arithmethics. VRONI was later extended to construct generalized Voronoi diagrams of input sets including also circular arcs [HH09, Hel11].



(a) Voronoi nodes to be deleted (b) Voronoi diagram updated

Figure 6: Example of an incremental topology-oriented construction of a Voronoi diagram of a set of segments (VRONI framework - figure from [Hel01]).
(a) In order for the dashed line segment to be inserted as seed, its endpoints are marked for insertion and topology-printed checks are performed to identify the seeds that will be eliminated (circled in gray).
(b) Once the new endpoints are inserted, the Voronoi diagram is updated, and additional topological checks are performed to evaluate the correctness of the insertion and of the new Voronoi diagram.

3.1.2 Approximated computation of a GVD

As we have previously discussed, achieving a stable and correct implementation that computes GVDs employing exact algorithms is not simple; moreover, issues like numerical mistakes can be difficult to spot (and therefore to handle), and preventing them would often imply to perform additional "sanity checks" and "adjustment" operations on the input set, which would introduce additional overhead. Some implementations deal with this by imposing constraints on the input set, limiting in fact the configuration that will be correctly handled by the approach. Doing so hardly fits our case, since the ECM approach aims at computing Navigation Meshes from real-life scenarios. The ECM approach employs instead an approximate method to compute a GVD initially described in [HCK$^+$99] which largely exploits some properties of the graphics hardware. [HCK$^+$99] defines a number of so-called "distance meshes" for common type of obstacles that can appear in the footprint of environments. In case of a 2D environment, 3D distance meshes can be associated with the 2D obstacles depending on their types, and once they have been correctly shaped, positioned, and rendered in a 3D space (using an orthographic top view), we will have a representation of the Voronoi Diagram in the GPU frame buffer. Therefore, the computation of the diagram is bounded to the size of the framebuffer, which will be our total "resolution". Some example of distance meshes will follow:

- A point in the 2D space is associated with a 3D circular cone (approximated as a triangle fan) whose radius at the base needs to be $M\sqrt{(}2)$ for a resolution of $M \times M$

(Figure 7a). A higher number of triangles approximating the cone will lead to a reduced error (if M is 1024, 85 are considered enough f.e.). The cone will have its bottom part touching the top part of the view, at the x, y coordinates of the corresponding 2D point.

- A segment in the 2D space is associated with a pair of cones (at the position of the 2 endpoints) plus a "tent" distance mesh: two quadrilaterals sharing an edge, in correspondence of the 2D segment (Figure 7b). Again, the upper part of the cones will touch the top plane of the view, as well as the edge shared by the quadrilaterals. Apart from the error induced by the cones at the endpoints, no additional error is involved with this distance mesh.

- A polygon in the 2D space is considered a less general case of any kind of site which is approximated as a collection of pre-defined features, namely, points and segments. Therefore, a polygon is considered a collection of line segments (edges) and points (vertices) in the same way as a curve arc is approximated with a number of segments and considered as a non-closed polygon (Figure 7b, 7c). Positioning follows the guidelines already defined for these features. The paper outlines that at vertices of polygons, only "partial" cones are needed to be rendered, decreasing the number of triangles rasterization operations.

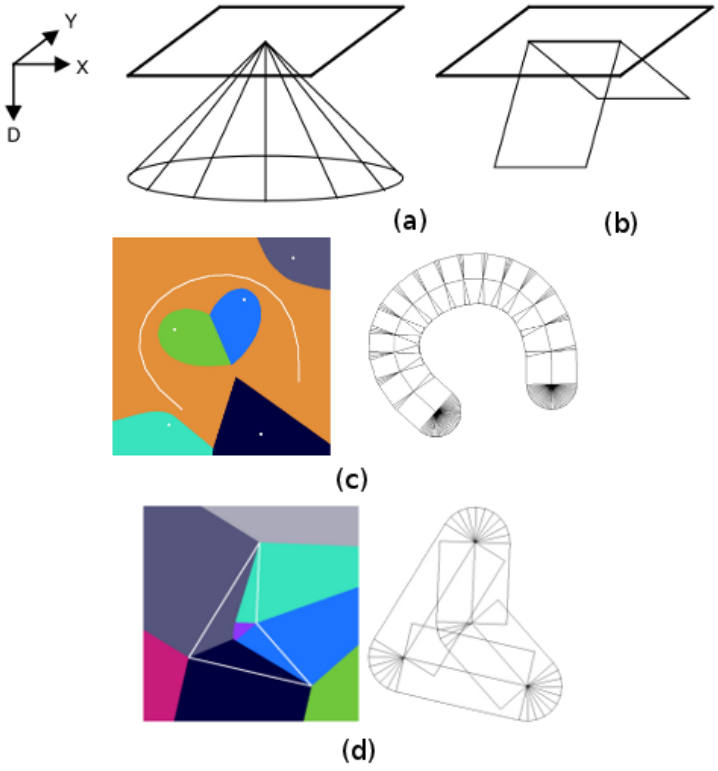


Figure 7: Example of 3D distance meshes (pictures from [HCK+99]. Axis D is the depth of the 3D view.)
(a) A cone distance mesh, from a 2D point.
(b) A "tent" distance mesh, from a 2D segment. Cones must also be added, in correspondance with the two end points.
(c) A curved segment can be approximated with a set of segments, and thus treated as a set of segments and points.
(d) The same applies to a polygon, which is a closed chain of segments. Notice that in the last two scnarios, only a section (or two) of the 3D cones corresponding to segments need to be rendered given that the remaining ones would be occluded by the quads of the tent distance mesh.

In theory, by assigning to every seed a different color, while rendering the distance meshes, the comparisons with the values in the z-buffer of the GPU will make sure that,

for example, when rendering the distance mesh of color $Color_i$, a pixel at position x, y will be colored with color $Color_i$ only if the mesh originated from a seed closer than the one it currently stores, if the cones part of the distances meshes consist of a high enough number of triangles.
Most approximated methods feature a parameter that regulates the correctness of the computation: in our case this parameter is the total resolution of the framebuffer, since our input geometry, and therefore the distance meshes, are discretized in a grid having the same resolution of the framebuffer. Intuitively, a higher resolution of the framebuffer implies a higher resolution grid, and thus a better, more fine-grained discretization.
It is important to notice that this method computes a "visual" representation of a GVD, and not directly a geometrical description of the structure. Therefore, we still can not express the GVD as a graph consisting of Voronoi vertices plus the Voronoi edges connecting them. To do so, we need to transfer all the framebuffer data from the GPU to the CPU, and to perform an additional "tracing" step on the CPU that searches through the whole color buffer on the CPU and creates the geometrical structure.

3.1.3 Problems of the current approximated ECM construction

The errors in the current ECM construction method mostly derive from mistakes that can occur in the approximate GVD construction approach previously described.
Part of these mistakes can occur because of the very nature of the algorithm: originally, [HCK+99] states some upper-limits to the number of triangles that should be used at some standard resolution for each 3D distance meshes to ensure correctness, but a further investigation realized that those upper bounds could not be enough to ensure correctness in some troublesome cases, as described in [Den03, SOM04, SGM05]. Those corrections alone seem to be not enough to ensure correctness: [Den03] further addresses some cases where using "correct" distance meshes still leads to problems, as in the case of two seeds lying both very close to a corner and at the minimum distance of 1 pixel. Therefore, if the seed S_i is located at $0, 0$ (bottom-left corner) and a different seed S_j is located at $1, 0$, the bisector could still be traced incorrectly (Figure 8) with many pixels could get the wrong color.
Additional mistakes occur when the input presents, for example, intersecting segments as seeds, or point seeds that lie on segments seeds. These kind of scenarios can be considered "degenerate" cases, and can eventually not be allowed in the description of the footprint, but still, detecting them implies a careful pre-processing of the input data. Moreover, it can happen that a point seed S_{point} and a segment seed $S_{segment}$ are specified as not intersecting objects according to the input footprint, but end up intersecting once they are discretized on our grid. This is likely to generate mistakes in the GVD since the cone spawned by S_{point} will maybe not be perfectly overlapped by the cones that originated from the distance mesh of $S_{segment}$. As discussed in [SOM04, SGM05], in general, errors arise when the generating sites are too close on the grid, because in those cases issues similar to z-fighting show up when Voronoi regions have to be determined, and colors can be assigned wrongly. Hence, correctness can be improved by subdividing the grid into small sub-tiles and then estimating more accurately the "influence" of each site in the description of the Voronoi regions inside the tile. Since the approach is in fact a GPU implementation, checking influence of the sites means performing GPU occlusion queries during additional passes, and eventually adjusting the color values. Mistakes can also show up when the discretization on the grid of the input data changes the topology of the input set, for example by changing a small line segment into a single point.

3.1.4 Final discussion on the current ECM computation process

In this chapter we have shown how the current ECM implementation computes a Navigation Mesh by "augmenting" a Medial Axis that is correctly computed pruning an approximated GVD of the input set. The GPU construction of the GVD is a central step since:

- Although it computes a "visual" representation of the GVD, it does not provide any usable geometrical description of the Voronoi graph itself. This graph needs to be

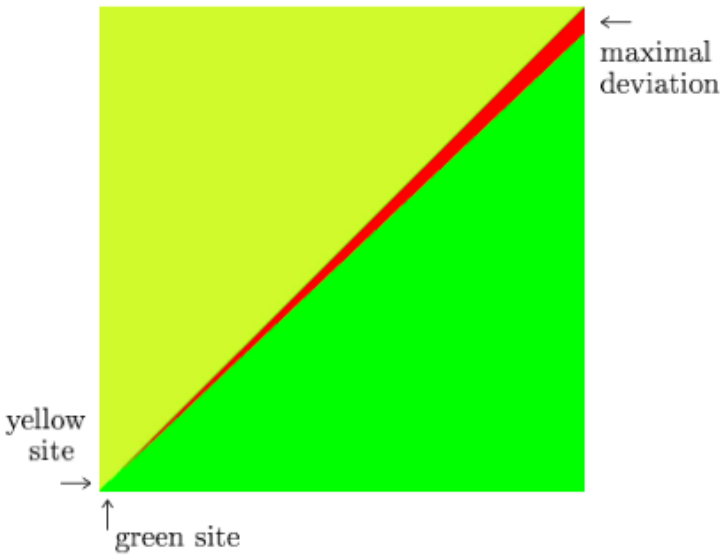


Figure 8: Issues with the method proposed in [HCK+99]: the pixels close to the bottom-left corner at coordinates (0,0) and (1,0) are our 2, but the 2 Voronoi cells generated can in fact be wrong since the pixel colored in red in the image can randomly get assigned to the wrong cell (depending on the unstable depth comparisons). Image from [Den03].

computed on the CPU. Therefore, the whole color data needs to be transferred back from the GPU. The transfer and the subsequent CPU tracing take up together a considerable amount of time.

- Tracing the graph requires having the whole color data on the CPU. Constraints exist on the size of a single buffer that can be allocated on the CPU, therefore this limits the resolution that can be used by the GPU step.
- Literature shows that the step can induce various mistakes.

Beginning to address the latest point, we can notice that literature describes also a number of additions that improve the correctness of the original method. However, those additions are quite complicated to implement and to extend, and although they seem to improve the general scalability, they negatively impact the performances. If we better analyze the method proposed by [HCK+99] and its improvements, we can notice that it seems to consider the GVD computation as a naturally discretized computation. In fact, this could also explain why no importance is given to the tracing of the geometrical structure itself: the GVD is just displayed, and this is considered good enough as a result. Subsequent work shows applications such as interactive 3D visualization of 2D/3D Voronoi diagrams, or other applications in the image processing domain. Path-planning is also considered as an application, but in fact it was not investigated in depth.
In our opinion, the algorithm, together with its evolution, was not always developed to "serve" a geometrical purpose. If we assume instead the visualization domain as the main field of application, the effort put into solving the geometric problems *inside* the discrete GPU "framework" makes more sense, since it allows for example the output data to be treated again as input set and it does not involve data transfers to the CPU.

In the ECM framework, instead, we are interested in computing a geometrically correct description of the Medial Axis, and the step described in [HCK+99] was mostly adopted because of its efficiency and its overall stability. However, our input set is specified as primitives in (double) decimal format, and the grid discretization is performed just because it is needed for the GVD computation. In fact, after the Medial Axis has been traced, coordinates are converted again to the decimals domain. What we can observe is that the

(a) (b) (c)

Figure 9: Approximating the medial axis graphically. (a) In the GPU framebuffer, each coloured zone corresponds to a unique closest obstacle. (b) Z-buffer of the GPU holds single channel data (greyscale). Smaller values map to darker colors. Hence, pixels closer to their closest obstacles have almost black values, while pixels close to Voronoi edges have more greyish/white values. (c) From the GPU information, a Medial axis can be computed. Note that the small black dots denote the 'event points'. Images by Geraerts [Ger10].

discretization of the input set over the framebuffer grid is what can cause many of the tricky scenarios to arise. Since our input set always specifies a bounding box, i.e. a rectangle which strictly includes all the other primitives, we can call $BBoxWidth$ and $BBoxHeight$ the dimensions of the bounding box, while $ResWidth$ and $ResHeight$ represent the dimensions of our framebuffer, therefore the resolution of our grid is $ResWidth * ResHeight$. Hence, each point of the grid is mapped to a corresponding decimal interval of size

$$(BBoxWidth/ResWidth)x(BBoxHeight/ResHeight)$$

Given that the bounding box is fixed by the description of the footprint, increasing $ResWidth$ or $ResHeight$ increases the resolution of the grid, and therefore the interval size mapped. This means that, for example, a footprint having $BBoxWidth == BBoxHeight == 4$ and two point seeds $P_1(x, y) = (0.1, 2.0)$ and $P_2(x, y) = (0.3, 2.0)$, when discretized over a grid with $ResWidth = ResHeight = 12$, will feature the two point seeds neighboring on the X-axis, since the grid coordinates of the input primitives are integers and are computed as

$$GridPoint_{ix} = round(Point_{ix} * ResWidth/BBoxWidth)$$

$$GridPoint_{iy} = round(Point_{iy} * ResHeight/BBoxHeight)$$

and therefore

$$GridPoint_{1x} = round(0.1 * 12/4) = round(0.3) = 0$$
$$GridPoint_{1y} = round(2.0 * 12/4) = round(6.0) = 6$$
$$GridPoint_{2x} = round(0.3 * 12/4) = round(0.9) = 1$$
$$GridPoint_{2y} = round(2.0 * 12/4) = round(6.0) = 6$$

assuming that round is defined as

$$round(f) = floor(f + 0.5)$$

Being so close, $GridPoint_1$ and $GridPoint_2$ can lead to mistakes in the computation of the GVD. However, if we increase the resolution of the grid so that $ResWidth = ResHeight = 48$, the new coordinates of the points on the grid become

$$NewGridPoint_{1x} = round(0.1 * 48/4) = round(1.2) = 1$$

$$NewGridPoint_{1y} = round(2.0 * 48/4) = round(24.0) = 24$$
$$NewGridPoint_{2x} = round(0.3 * 36/4) = round(3.6) = 4$$

$$NewGridPoint_{2y} = round(2.0 * 48/4) = round(24.0) = 24$$

and their distance on the $X - axis$ is not anymore 1 but 3. Trivially, increasing the resolution can prevent some troublesome scenarios to appear in the input grid of the approximate computation. Eventually, the input set could be pre-processed to check the minimum resolution that should be used in order to avoid the presence of problematic configurations in the approximated grid.

This being said, increasing the resolution has some consequences. Since the method described in [HCK+99] is based on rendering 3D primitives, computing 2D GVDs scales well with the increase in the resolution because graphics hardware is extremely optimized for this operations. However, increasing the resolution has a major negative impact on the time spent on the transfer of the framebuffer data from the GPU to the CPU, and at the same time the performance of the subsequent CPU tracing operation does not scale well with the resolution. Moreover, the total resolution is limited both by the frame buffer resolution of the GPU and by the constraint on the allocation of a single buffer on the CPU. Luckily, the GPUs have evolved significantly in the last 10 years and they are today capable of handling very big (High-Definition) resolutions, while the second limitation could be overcome by changing the CPU tracing step so that the big CPU buffer containing the framebuffer data is not needed anymore.

Having noticed that the use of higher resolutions can lead to ECMs with less mistakes, we will now describe a modified approach that will aim at removing the limitation on the resolution that can be used for the computation of a Navigation Mesh, and that will employ modern GPU-parallel and CPU-parallel technologies and related ideas, in order to compute higher resolution, and thus more correct, Navigation Meshes with similar or better performance.

4 GPGPU to improve ECM construction

In this chapter we will investigate how the construction method of an ECM could be changed in order to have an approach that allows higher resolutions and, eventually, performing at the same time better than the original one. Ideally, after having computed a "visual" representation of the GVD in the framebuffer, we will try to perform additional operations on the GPU so that it will not be necessary anymore to transfer the entire color data back to the CPU; moreover, since modern GPUs implement an efficient parallel programming model, usually addressed with the term "GPGPU", we hope to increase the efficiency of the procedure, too.

4.1 General-purpose computing on graphics processing units

The acronym GPGPU stands for "General-Purpose [Computing] on Graphics Processing Units" and is generally used to refer to programs that use the graphics hardware to perform more general and non-strictly graphical calculations. GPGPU roots back to the introduction of the so-called "programmable pipeline" to GPUs: programmers started using the associated Shader APIs to write non-graphics related programs which would be then executed in parallel on the graphics hardware, observing a performance boost. Eventually, both ATI and NVIDIA, leading companies in the GPU hardware industry, released new APIs that allowed an easier development of general, non-graphics programs that can be run on the GPUs, called "NVIDIA CUDA" and "ATI Streams". A short description of the structure of the API and of the GPU computing model will follow, in which we will use the terminology adopted by CUDA, although all the concepts have their correspondent in other APIs.
GPGPU defines a parallel computing model called "Single Instruction, Multiple Thread" (SIMT). In the SIMT model, some functions, called kernels, are executed on the "device" (the GPU) by a high number of threads at the same time while referring to the same set of input data. SIMT distinguishes itself from the SIMD (Single Instruction, Multiple Data) model mostly because the shared working data, that has to be transferred from the CPU (the "host" component) to the GPU "manually" (which means that transfers must be handled by the programmer), can be arranged in a number of different ways, while instead the SIMT model defines stricter rules and constraints on the format of the input data (such as dimensions and padding). An example of SIMT is the SSE instruction set of modern Intel processors. Notice that CUDA includes the SIMT model too, and also in a quite straightforward way given that it also defines instructions that can operate, for example, on sets of 4 floats packed together, mimicking in this way the SSE instructions.
Threads executing a kernel are logically organized into a "Grid" and divided into "Blocks": Blocks share a limited amount of memory that has a very low read/write latency. Each thread has its "index" mapped to a Grid position, with a different value for each of the dimensions of the Grid (3 at most). At the same time, GPUs can be considered a collection of "Streaming Multiprocessors" (SMs): up to a certain number of blocks can be assigned to each SM (as long as the total number of threads is under a particular threshold). Threads in Blocks are further subdivided into "Warps" (Warp size is 32 on most recent GPUs) which are scheduled to execute concurrently on the assigned SM. Hence, SMs are capable of advancing at the same time 32 threads by executing concurrently the same instruction of a kernel 32 times: warp scheduling has almost 0 latency (usually this is referred as "zero-overhead thread scheduling"). With all this being said, it is easy to understand how GPUs can claim such a high number of GFLOPS per second (Figure 10).

Considered that no assumptions can be made on the order in which Blocks and threads are scheduled, GPGPU is usually well suited to solve problems involving computations that can be subdivided into a big number of fairly independent, all equal operations, performed on memory with strong locality properties. An example could be applying a change to all the pixels of one image according to some external formula, which is in fact exactly what Shader/GPGPU APIs were originally invented for, but not all algorithms can be expressed by the SIMT model, or not entirely [KWmH10].

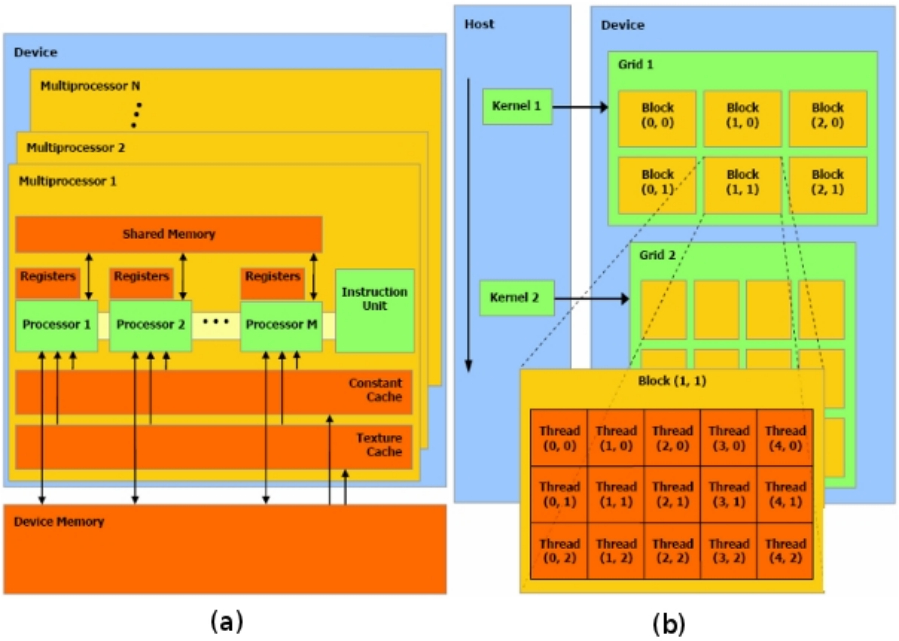


Figure 10: CUDA architecture (simplified). (a) CUDA Streaming Multiprocessor: a SM is composed of many Streaming Processors (or CUDA Cores) that will execute the Blocks. CUDA Cores share texture units, shared memory and constant cache. Device memory instead is shared among all SMs. (b) CUDA execution model: A Grid is defined before the execution of a Kernel. Blocks are created inside, and threads are split into these Blocks. Blocks and threads are assigned IDs. Blocks will be effectively scheduled to Streaming Multiprocessors.

4.1.1 Common GPGPU characteristics and related issues

Developing kernels for the GPU involves getting acquainted with some basic ideas and the common issues they raise; to not understand them correctly can prevent important optimizations from occurring.

Proper use of the various kinds of memory. The standard GPU DRAM memory is referred usually as "Global" memory. Global memory can be read and written by all threads of a Grid executing a kernel, but it is also the slowest memory available. Blocks all share a smaller amount of read/write "Shared" memory: shared memory is much faster that the global one, but is quite limited (around 16KB on most GPUs). On the device we can also have Constant memory, which can be read very quickly, but not written, by all the threads of a Grid, and is also quite limited (65 KBs usually). Threads also have their own private memory, which is mapped to registers of SMs. A common "pattern" that kernels can follow is to start by initializing the shared Block memory, usually with some values read from the Global memory, and then to perform the computations reading from and writing to the fast shared memory, updating the Global memory only at the end of the process by reading the partial result(s) from the Shared memory. To ensure consistency in read/write operations of different threads belonging to the same Block, the instruction `__synchronize()` can be used: a kernel encountering the `__synchronize()` function will be halted until all the threads of the same Block will also reach that instruction. On most platforms, if a kernel performs at least 2/3 reads from the global memory, using sharing memory in-between should be enough to already grant a speedup (Figure 10).

Together with the idea of using all the different kinds of memory correctly comes the concept of "Memory Coalescending": basically, given that warps execute together, they can also read from memory together, if memory locations happen to be contiguous. Thus, if threads from Thread 0 to Thread 15 read respectively from the locations `global_array[0]` to `global_array[15]`, the read operation will occur as a single memory load. Kernels usually have to be carefully designed to trigger coalescing, especially when moving data from

global to shared memory (or the inverse).

Dynamic (re)redistribution of the resources. GPGPU APIs usually provide functions to query the underlying hardware on which the program is being executed, thus it is possible to know what the maximum number of threads/Blocks that we can run are, and all the related information. As programmers, our goal is to maximize the use of all the resources: every time we start a GPGPU computation, parameters such as the Grid dimensions and the amount of Shared memory to allocate must be selected, thus all these resources are dynamically allocated, and they should be chosen with maximization in mind, remembering at the same time that, depending on the current status of the computation and the current hardware usage, the same kernel could require to be executed with (quite) different dynamic parameters (such as: the Blocks' number and Blocks' size) every time to achieve the best performance.
Dynamic resources sometimes are not explicitly declared when launching a kernel, but implicitly, and we have to keep in mind that this could still impact performances. For example, the hardware defines a limit on the number of total registers that can be allocated at the same time on a SM. Since threads' private variables are usually stored in registers, we have to take the number of these variables into account when deciding our Block/Grid size: just by adding one variable to a kernel, we could end up with a Block dimension that does not allow maximum utilization anymore (because of the new total number of registers requested by the Blocks of threads), even though at the same time a quick accessed private variable can save time on memory fetching parts. Therefore, a number of tradeoffs arise, and evaluation through extensive testing can be the only solution. Nvidia released a useful "Occupancy Calculator" in the form of a simple spreadsheet to help dealing with these issues.

Diverging Threads. When threads of the same warp follow different paths of control flows, we say that these threads diverge. Divergence has a severe impact on performance, because the instructions that the threads of the same warp do not "share" can't be executed in parallel, but are instead serialized. The situation can become even worse if we combine this scenario with non-optimized memory accesses. Typical constructs that can lead to divergence, and thus should be employed carefully, are if/then/else constructs and for loops. Divergence can be avoided, for example, by re-designing algorithms, by adjusting the Block and Grid sizes taking into account warp size, and by "padding" the memory accesses depending, again, on warp size.

Latency Hiding. Given that GPUs have so many cores and can execute so many mathematical instructions at the same time, bottlenecks in GPGPU applications can easily be represented by the time spent in creating/destroying a kernel and in the idle time that occurs when all kernels have to wait for a read operation to complete in order to continue. A simple but important solution to deal with this is to increase the number of mathematical instructions performed by threads, for example by having each thread modifying the values of many pixels instead of just one. This way, a SM will better employ the time that it would probably instead spend waiting for a read operation to be finished, thus "hiding" (not avoiding) the latency of the read operations. Also, the number of threads spawned and destroyed will then probably decrease, too, granting an additional improvement.

4.2 Computing Medial Axis on Graphics hardware: Relevant literature

Computing a Medial Axis or a (generalized) Voronoi diagram on the GPU seems to be a fairly active applied research field. We have already introduced the paper by Hoff et al. [HCK+99] and the updates that followed it: the method takes advantage of the sole GPU rasterization operations, with no involvement of any GPGPU techniques, nor any use of the GPU programmable pipeline. Moreover, it is important to remark that the method only "draws" a generalized Voronoi diagram on the framebuffer, without computing in fact any useful geometrical description of the structure.

A technique named "Jump Flooding" was introduced in 2006 by [RT06] to compute 2D Voronoi tessellations (even generalized) on the GPU. Basically, the x and y coordinates

of each seed are stored in the correspondent point of a big texture. Points of the texture that are not seeds will be initialized with the pair of values (nil, nil). A value $stepsize$ is also picked. Then, a pixel-shader program is launched, spawning a GPU thread per pixel: each thread will read the values of the 9 pixels at coordinates $(x+i, y+j)$, where $i, j \in \{-stepsize, 0, stepsize\}$, and in case any of them yields a value (a pair of coordinates), the thread will compute the distance between pairs of coordinates and the current (x, y) pair identifying the pixel. If any of these distances is smaller than the distance between the pair (x, y) identifying the pixel and the pair stored in the pixel, then the correspondent pair will be the new pair stored at position (x, y). After an entire round of GPU threads, the values $stepsize$ is updated: the paper shows, if the resolution of the texture is $n \times n$, by starting with $stepsize_0 = n$ and then subsequently updating $stepsize$ so that $stepsize_{t+1} = log_2(stepsize_t)$, that we end up with a fairly correct propagation of the values among all the pixels in about $log(n)$ rounds. This approach is expanded in [RT07] that shows how to generalize it to obtain 3D Voronoi tessellations, how to reduce the errors significantly with little cost, or how to speed up the whole computation by "downsampling" the texture data. The approach eventually evolved in the "Parallel Banding" approach [CTMT10] that computes the "Euclidean Distance Transform", a concept similar to the Voronoi diagram that instead aims at storing for each point the closest distance to a site, rather than the position of the seed itself. The novelty of the method is the use of a GPGPU technology as CUDA, and thus the attempt at creating an algorithm that fits the related parallel model. The "Jump Flooding" and "Parallel Banding" approaches seem to scale better with the number of input seeds than the approach by Hoff et al., but again, these approaches compute no geometrical description of the structure.

The interesting fact is that, originally, basic algorithms such as the "Marching Cubes" and its 2D counterpart "Marching Squares", that aimed at computing the isosurface (or isoline, when in 2D) of a scalar field were considered "embarrassingly parallel": an isoline of a 2D scalar field is similar to a Voronoi diagram of only 2 seeds, and thus such algorithms could be easily extended to compute a Voronoi diagram. The Marching Squares algorithm subdivides the input into groups of 8 neighboring "voxels" and computes the parts of the isosurface that each group would generate; it is clear that, given the locality of the approach, the algorithm is easily parallelizable. However, at the end of the process the various parts have to be "merged" into the final isosurface (or isoline): originally, the algorithm only displays the final mesh, therefore acting again in parallel by having each "working unit" drawing a part of the final mesh, but no geometrical description is computed.

In fact, "cell-by-cell" approaches that try to take advantage of the massively parallel GPUs, as the ones shown above, are known to face issues when they have to assemble a final description of the geometrical structure, incurring in a double penalty: they can hardly take advantage of parallelism, and are therefore required to use a smaller number of threads, threads that at the same time suffer from the irregular memory access patterns that a phase of graph re-construction induces. This has been already discussed in the literature, and for example [Ble10], the paper that describes the way the current NVIDIA "GPU AI - Path Finding" framework computes Navigation Meshes, notices how the GPU graph search that has to mark which samples of the 3D space form a good coverage of the free-space is the step that takes up, on average, 78% of the computational times alone, exhibiting as well fairly un-coalesced (and thus unoptimized) read and write operations. A more recent article [HKKOO11] from the Computer Systems Laboratory of Stanford University offers a good overview on the problem, and explains more carefully why the GPU programming model pays a performance penalty when it comes to graph-based algorithms, mostly because it leads to frequent threads' branching and to irregular memory accesses. The paper also shows that good GPU parallel implementations of algorithms such as Breadth-First Search benefit from the SIMT model when used on large graphs (order of magnitude $\geq 10^3$), probably because the advantage of using so many independent threads at the same time overcomes the performance penalty caused by branching/irregular accesses. Moreover, [HKKOO11] also presents a new GPGPU programming technique that aims at executing with "thread granularity", and thus at the same time by many GPU threads, only those operations that are clearly not branching (such as copy/read of memory buffers), while performing the others with "Warp

granularity": since branching and coalescence are measured only "per thread", the negative effect is minified. Clearly, this means that some groups of (up to) 32 threads (32 is the warp size of most CUDA capable GPUs) will execute the same set of instructions at the same time, thus reducing the number of simultaneously active working units by a factor of 1/32 (at maximum), but reducing threads' branching in fact seems to grant anyway a performance boost.

Taking into account the considerations above, it seems clear that working on a GPU parallel algorithm that computes an ECM, or even one of its "basis" structure such as a GVD, will be at the same time difficult and probably not beneficial, considering also that currently ECMs, being compact descriptions of the walkable space of an environment, exhibit graph having only up to a couple of hundreds of edges. Therefore, in our case, the penalties that can originate from branching and irregular memory accesses will maybe not be counterbalanced by the advantage of having a huge amount of working units. Moreover, as opposed to the case of a simple BFS, we would like not to simply visit and mark the nodes of a graph, but also to store them in a common organized structure, which can pose the additional read/write synchronization issues.
Therefore, the approach that we will present will focus on:

- Performing additional steps on the GPU that seem to naturally fit the GPGPU model, while basically leaving out those that are shown not to do so (for example assembling the final graph).
- By keeping in mind the geometrical nature of the ECM structure, we will try to compute the biggest possible amount of information required by an ECM construction on the GPU, ending up with transferring back only relevant information.

4.3 Overview of the new Approach

As a starting point, we assume to have a visual representation of a GVD on the GPU. This can be the output of the method by Hoff et al. or could be also generated using the Jump Flooding technique. Both have advantages and disadvantages in terms of speed, with the former one scaling better with the resolution of the grid, while the second one scales better with the number of input primitives. We choose to adopt the algorithm by Hoff et al., since the errors it can generate are clearer, and because its property of scaling better with the resolution resonates with our goal of reaching very high resolutions.
Given the color data representing a GVD on the GPU, we define a 2D $grid(x, y)$ of points, with $0 \leq x < bufferWidth$ and $0 \leq y < bufferHeight$; $bufferWidth$ and $bufferHeight$ are the dimensions of the framebuffer that define the total resolution. Points are placed so that they have, when possible, 4 touching pixels around them, apart from those touching any of the borders which exhibit instead less than 3 neighboring pixels. The origin of the axes is located at the topmost left corner; the x axis runs from left to right while y runs from top to bottom. In case of a grid point having 4 neighboring pixels, and thus colors, we call the colors $ColorLU$, $ColorRD$, $ColorLU$ and $ColorRU$. Pictures 11a and 11b show these assumptions. To each pixel of the grid we can assign an $ID = (y * bufferWidth) + x$, a single value that identifies the 2D position in the grid.

Then, we observe that transferring the whole color data to the CPU for further tracing is a big waste, since most of the information on the GPU is in fact useless. Considering the definition of the ECM, the information that we actually need is:

- The locations of the Voronoi vertices, because a subset of them will become the vertices of our ECM.
- The locations of all the edge-points, which are the points that will form the edges connecting our ECM vertices.
- Among the edge-points, we need to know which ones will be event-points of the ECM edges, according to the definition of [Ger10]. They can be either Voronoi vertices or edge-points in our GVD.

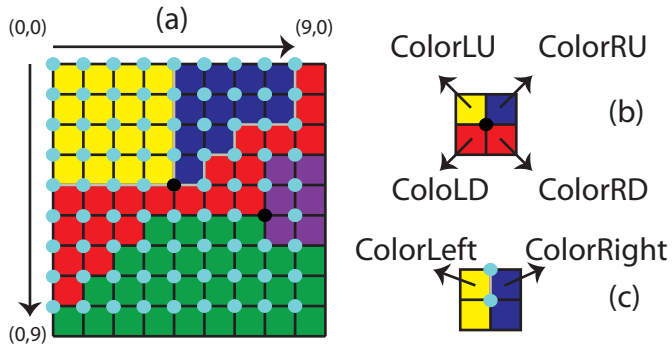


Figure 11: Example of the adopted grid.
(a) A grid, together with the associated pixels' data.
(b) Color data of a vertex with 3 different pairs of neighboring color values.
(c) An edge-point, with its pair of different color values (to its left and to its right)

- For each vertex and event-point of the ECM, the location of the closest points to them that lie on obstacles.

Trivially, the information specified above is a smaller subset of all the points of the grid. If we could compute it on the GPU efficiently, then we would save a considerable amount of time in the data transfer, and at the same time we can expect an easier tracing routine on the CPU, since there will be no need to compute any additional information.
In the following section, we will describe how to compute all the needed information on the GPU.

4.3.1 Locating Voronoi vertices and Voronoi edge-points

With the color data on the CPU, computing a list of all the locations of the Voronoi vertices is trivial, and is shown in Algorithm 1. Since each point has (up to) 4 neighboring colors (squares of the grid), we can use the function GetDirections to check the number of pairs of colors that

- share a side of the grid
- are different from each-other
- are both different from black (color assigned to obstacles of the footprint)

Points that yield 3 or 4 "local directions" are degree 3 or 4 points and are thus Voronoi vertices. Similarly, points with degree 2 are Voronoi edge-points. However, Algorithm 1 calls the function GetUniqueDirections, that also checks whether to a pair of different neighboring colors are also associated two distinct Closest Points before counting an additional direction. This function allows to select only the vertices of the Medial Axis, but performs the additional expensive computation of the Closest Points. At the end of the routine, we will have all the locations of vertices and edge-points, respectively, in lists *verticesList* and *edgePointsList*.

Locating Voronoi vertices and Voronoi edge-points directly on the GPU poses some additional challenges. For each point, the original CPU function can call either the single GetDirections function or both GetDirections and GetUniqueDirections, and these two very different control flows can rapidly increase the branching of the routine, deteriorating the GPGPU performance. At the same time, GetUniqueDirections includes the computation of the Closest Points, a fairly expensive geometrical computation that, instead, we would prefer to treat more carefully separately, to try to achieve maximum efficiency. Therefore, at this point, we will not perform any operation similar to GetUniqueDirections, thus limiting the routine to the computation of non-unique directions. This means that we will also detect vertices of the GVD that will eventually prove not to be vertices of

Algorithm 1 FindVoronoiVerticesAndEdgePoints($colorbuffer$, $bufferWidth$, $bufferHeight$, $verticessList$, $edgePointsList$).
Input: A CPU buffer $colorbuffer$ with width and height $bufferWidth$ and $bufferHeight$ holding the color data representing our GVD, two empty lists $verticessList$ and $edgePointsList$ that will hold the results.
Output: Output is appended to lists $verticessList$ and $edgePointsList$.

```
1: for x = 1 → bufferWidth do
2:   for y = 1 → bufferHeight do
3:     nDirs ← getDirections(colorbuffer[y * bufferWidth + x])
4:     if nDirs > 2 then
5:       nUniqueDirs ← getUniqueDirections(colorbuffer[y * bufferWidth + x])
6:       if nUniqueDirs > 2 then
7:         sample ← CreateSample(x, y, colorbuffer)
8:         verticessList.Append(sample)
9:       end if
10:    else if nDirs > 1 then
11:      sample ← CreateSample(x, y, colorbuffer)
12:      edgePointsList.Append(sample)
13:    end if
14:  end for
15: end for
```

Algorithm 2 FindVoronoiVerticesAndEdgePointsGPGPU($colorbuffer$, $bufferWidth$, $bufferHeight$, $verticesMarkers$, $edgePointsMarkers$).
Input: A CPU buffer $colorbuffer$ with width and height $bufferWidth$ and $bufferHeight$ holding the color data representing our GVD, two buffers $verticesMarkers$ and$edgePointsMarkers$ as big as $colorbuffer$ that will be written.
Output: Output is written to the buffers $verticesMarkers$ and $edgePointsMarkers$.

```
1: {This function is being executed by multiple threads on the GPU}
   {Threads have an ID that depend on the distribution of the workload}
2: x = GetXCoordinateFromGPGPUID()
3: y = GetYCoordinateFromGPGPUID()
4: nDirs ← getDirections(colorbuffer[y * bufferWidth + x])
5: if nDirs > 2 ∨ (nDirs > 0 ∧ (x == 1 ∨ x == bufferWidth − 1 ∨ y == 1 ∨ y ==
   bufferHeight − 1)) then
6:   verticesMarkers[y * bufferWidth + x] = 1
7: else if nDirs > 1 then
8:   edgePointsMarkers[y * bufferWidth + x] = 1
9: end if
```

the Madial Axis, and edge-points that are parts of edges that will not appear in the final ECM. Filtering out those points that will not be part of the ECM will be performed during subsequent steps, as well as the computation of Closest Points. The GPGPU version of FindVoronoiVerticesAndEdgePoints will launch a CUDA kernel spawning an independent thread for each grid point, and each thread will examine the 4 colors $ColorLU$, $ColorLD$, $ColorRU$ and $ColorRD$ neighboring the assigned point. Spawning a thread per grid point, and thus a thread per pixel, means anyway subdividing the work-load in the standard, parallel-friendly CUDA entities (shown in Figure 12).

Re-designing Algorithm 1 for the GPGPU model poses a more subtle but additional challenge: Algorithm 1 stores the vertices/edge-points detected in two separate lists that are continuously updated. Since GPGPU defines a parallel model, and considering that is desirable to have the highest possible number of GPU threads performing non-interfering computations, we face the problem of multiple threads updating some shared buffers. On the GPU, because of the high number of threads and all the issues related to divergence,

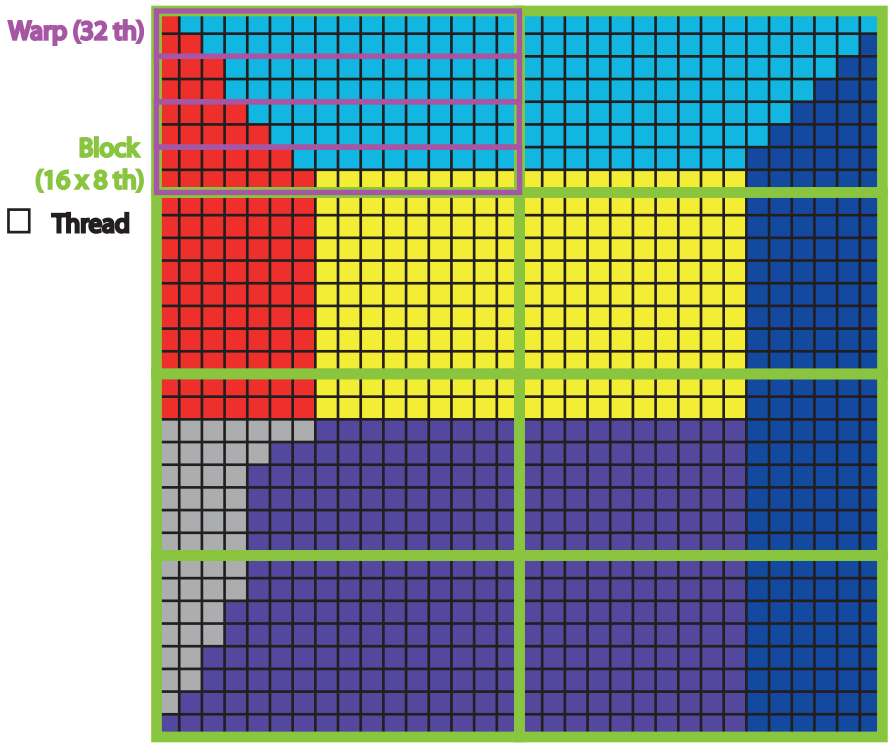


Figure 12: Example of input color data, and its CUDA subdivision for the first step. The GPGPU step that will locate Voronoi vertices and edge-points will spawn a CUDA thread (represented by a single rectangle) for each pixel, checking the neighboring color data through the use of Texture memory. Threads are advanced in parallel in groups called "warps" (depicted in purple), with every thread of the warp executing instructions simultaneously. Warps are then grouped into CUDA Blocks (in green): CUDA Blocks are what is sent to the Streaming Multiprocessors of the GPU. The size of a CUDA Block should be specified: in general, we use bigger Blocks than 16x8, used in the picture.

the problem can strongly impact the performance, and is at the same time harder to solve efficiently because of the different types of memory available, with different advantages and limitations.
A first possible way to solve the problem is to use atomic operations, which are defined by the CUDA API and available on many (not all) CUDA capable GPUs. Atomic operations abstract the use of synchronization mechanisms, adding the cost of such synchronization to the routine. This cost can be quite high for a CUDA routine, especially considering that our buffers are located in Global Memory and that are thus accessed at the same time by all the threads of the kernel, that will eventually have to be synchronized. A known way to address this is to define many different and smaller buffers in Shared Memory to use as lists, and make sure that threads of each CUDA Block will first update their "Block Shared" buffer, copying the content of these buffers to Global Memory only at the end of the whole computation. Threads will synchronize only for the accesses to Shared Memory, decreasing the total cost of synchronization since less threads will concur for the use of each buffer. However, this solution requires a more complex implementation, together with the use of atomics that act on Shared Memory, which are defined by the CUDA API but available on a significantly smaller number of CUDA capable GPUs than the number of GPUs supporting atomics that act on Global Memory.

In the end, we decided to adopt a different solution, adapted from the techniques usually employed to perform "stream compaction" of "sparse" data on the GPU. Two big output buffers $verticesMarkers$ and $edgePointsMarkers$, as large as the grid, are allocated on the GPU, with all their values initialized with 0. During the GPGPU FindVoronoiVerticesAndEdgePoints routine (Algorithm 2), if a Voronoi vertex is recognized by a thread, a 1 is written at the correspondent position of the buffer holding the "markers", and the same

happens for the edge-points (Figure 13). This way, we have no concurrence involved if we spawn one thread per point, since each point has a separate and defined location to perform its write operation to. Then, we perform an exclusive "prefix sum" on the two output buffers that "mark" the positions of vertices and edge-points.

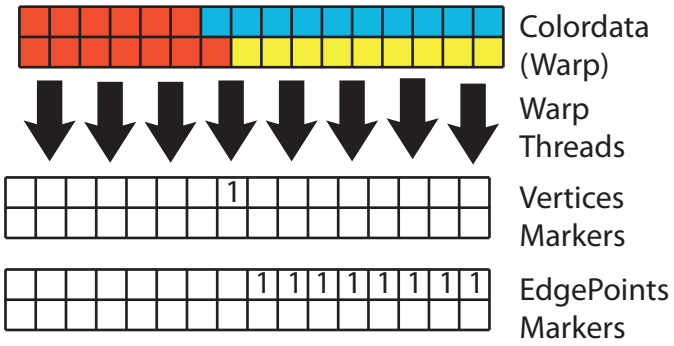


Figure 13: Example of Voronoi vertices and edge-points detection on Warp basis: each CUDA thread checks neighboring pixel values associated to point of the grid, and eventually writes a 1 at the corresponding position of **either** the $VerticesMarkers$ or $EdgePointsMarkers$ arrays.

The prefix sum (also "prefix scan" or just "scan") of an input sequence of values $\{x_0, x_1, \ldots, x_n\}$ is defined as the sequence $\{x_0, x_0 + x_1, \ldots, \sum_{i=0}^{n} x_i\}$. A Pefix sum is easy to compute sequentially, but a parallel computation is not trivial and used to be the subject of many fundamental studies concerning parallel algorithms [Ble90], since its generalization can be considered an important "building block" for many parallel algorithms, such as sorting. Prefix sums are now known to be a strongly parallelizable algorithm on the GPU [HSO07], too. We will perform an "exclusive" prefix sum, which is simply a standard prefix sum "shifted" right by one position, with a 0 as first element. If we compare, for example, the array marking the vertices with the "scanned" one we can notice that if we consider the sequence of values located only at the positions where there used to be ones, then we have an increasing sequence of values starting with 0 (Figure 14). Both $verticesMarkers$ and $edgePointsMarkers$ will be "scanned" and the results will be written to the 2 additional buffers, $verticesMarkersScanned$ and $edgePointsMarkersScanned$, having the same size. At this point, we can also notice that if we sum the last element of the scanned buffer with the last element of the original correspondent one (this latter element can be either 0 or 1) we have the total number of vertices (or edge-points) detected. Therefore, we can then allocate two more GPU buffers $verticesIDs$ and $edgePointsIDs$ of the right size that will host the IDs of the detected vertices and edge-points, and then call another GPU kernel that will spawn multiple threads, one per pixel, each thread checking if the associated pixel identifies a vertex (or an edge-point) by looking at the buffers with markers $verticesMarkers$ and $edgePointsMarkers$; if it does, the thread will write the ID of the associated pixel at index i of $verticesIDs$ or $edgePointsIDs$, where i is the value that can be read from either $verticesMarkersScanned$ or $edgePointsMarkersScanned$ (shown by Figure 14, too). In addition, threads of this kernel will also take care of copying the color data associated with the points to the 3 additional arrays $EPColorsLeft$, $EPColorsRight$ and $VPColors$. The reason why the edge-points' color information is split in 2 separate arrays will be clear later in this work.

The described approach involves 4 separate operations (1 one kernel that marks positions, 2 prefix-scans and 1 additional launch that re-organizes the IDs of the points and downloads color data), thus 4 separate launches of CUDA kernels, and 4 additional big but temporary buffers (2 used for the markers and 2 more to hold the results of the scans) on the GPU, but it allows to launch at each step the maximum possible number of threads (one per pixel of the grid) that always perform independent or parallel-friendly (as in the case of

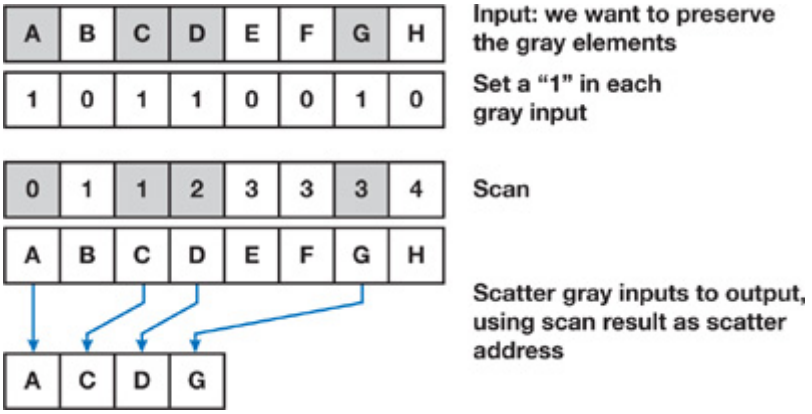


Figure 14: Stream compaction steps (taken from [HSO07]). First, the elements that we want to preserve are "marked" with 1s, in a separated array, which we then scan (performing an exclusive prefix sum). In the scanned array, at the positions of the elements we have marked previously, an increasing sequence of numbers starting with 0 can be observed. Moreover, if we add the last element of the scanned array to the last value of the "markers" array we obtain the number of values marked (4 in the picture). Therefore, we can allocate a new array of size 4 and then copy the element i of the input array (with letters) only if it is marked (1 in the second array) and at the location stored at position i of the scanned array. In our implementation, we perform the same exact operations but on 2D arrays.

the prefix sum) operations. The first kernel reads 4 values per thread, all at spatially-neighboring locations; the same applies for the fourth kernel which has to download up to 4 color values per pixel. The first and third kernels write values at consecutive, contiguous and non-overlapping locations. Therefore, the write operations should be coalesced/optimized. At the end of the three launches, the GPU buffers $verticesIDs$ and $edgePointsIDs$ will contain the IDs of the n Voronoi vertices and the m IDs of the Voronoi edge-points, while the arrays $VPColors$, $EPColorsLeft$ and $EPColorsRight$ will contain, respectively, $4n$, m and m color values.

4.3.2 Computing Normals to the Obstacles' Segments

For the following steps of an ECM construction we need to compute the normals generated from the primitives' data, which is the set of normals perpendicular to each segment defining a primitive. An obstacle is defined and stored by the ECM framework as a list of points $(P_1, P_2, \ldots, P_i, \ldots P_n)$. Therefore, n normals to segments $(P_1P_2, \ldots, P_iP_{i+1}, \ldots, P_nP_1)$ are generated per obstacle; each normal N_i starts at obstacle point P_i and points outwards. Normals are needed during the computation of an ECM to locate the event-points on the edges [Ger10].

We can compute the primitives' normals directly on the GPU if have the primitives' data on the device, too. In the current ECM framework, primitives' data is not explicitly loaded on the GPU, therefore we will have to transfer it: since any primitive can be expressed as a set of segments, the whole primitives' data can be represented as a long set of segments, hence as a long set of pairs of points. We will store primitives points in 2 separate arrays $Primitives_1$ and $Primitives_2$ that will hold consecutive points of the segments: if segment S defined by the 2 points Sp_1 and Sp_2, $Primitives_1$ will hold Sp_1 and $Primitives_2$ Sp_2. We will also store the number of segments per primitive in the array $NSegsPerPrim$.

The type of each primitive $prim$ can be:

- **Point**: a point primitive $PPoint$ can be considered a segment with two coincident points. In this case, we will add $PPoint$ to both $Primitives_1$ and $Primitives_2$
- **Line Segment**: a line segment $PSeg$ is defined by the two points $PSegP_1$ and $PSegP_2$. Moreover, a line segment generates 2 normals, in both orientation. Therefore, we will store the segment two times, in inverted order: $Primitives_1$ will get $PSegP_1$ and $PSegP_2$ while $Primitives_2$ will receive $PSegP_2$ and $PSegP_1$.

- **Border**: the ECM framework defines the "border" primitive, which is a segment part of the bounding box enclosing the footprint. Unlike standard segments, borders "generate" only one normal, pointing inside the footprint. However, to keep things simpler, we will just treat borders as standard segments and have them generate an additional (unused) normal.
- **Polygon** a polygon primitive $PPolygon$ is defined by $n > 2$ points

  $$\{PolP_1, PolP_2, PolP_3, \ldots, PolP_n\}$$

  and therefore by the n segments

  $$\{(PolP_1, PolP_2), (PolP_2, PolP_3), \ldots, (PolP_n, PolP_1)\}$$

  We will then store the segments of the primitive by adding $\{PolP_1, PolP_2, \ldots, PolP_n\}$ to $Primitives_1$ and $\{PolP_2, PolP_3, \ldots, PolP_n, PolP_1\}$ to $Primitives_2$.

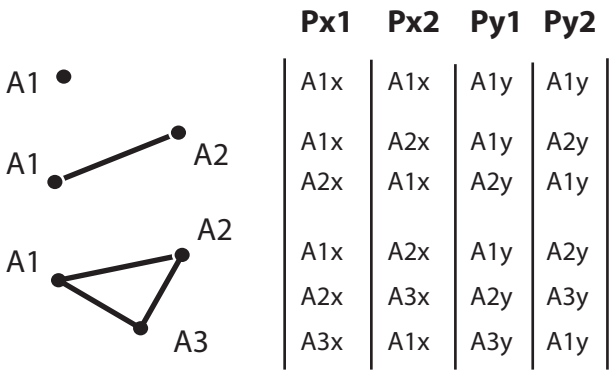


Figure 15: Organizing Primitives' data using the Structures of Array principle. For each primitive, its coordinates are added to the arrays that describe obstacles data, which are $Primitive1X$, $Primitive2X$, $Primitive1Y$, $Primitive2Y$.
A **point** will be considered as a segment with two coincident endpoints, therefore $Primitive1$ and $Primitive2$ will all receive the same pair of values at X and Y positions. Points will not spawn any normal, but they have to be stored f.e. for rendering.
A **segment**, or a **border**, will add 2 values to all the 4 arrays, thus 2 normals will be generated (in both directions).
A general **polygon** follows the generalization of segment, therefore an n-sided polygon will add n values to all the 4 arrays, spawning n normals, 1 per segment, in the outward direction.
Reading vertically, we can see the sequence of values that each array will store at the end of the process. This organization of the data conforms to the SoA principle more and allows more efficient GPGPU reads.

Additionally, we will split the points of $Primitives_1$ and $Primitives_2$ in 4 additional lists $PrimitivesX_1$, $PrimitivesY_1$, $PrimitivesX_2$ and $PrimitivesY_2$, in order to store x and y coordinates separately. These choices allow us to have each segment i belonging to the primitives' data defined by the points ($PrimitivesX_1[i]$, $PrimitivesY_1[i]$) and ($PrimitivesX_2[i]$, $PrimitivesY_2[i]$). Storing the data in those 4 arrays complies to a "Structure of Arrays" (SoA) organization of the data which leads to more efficient memory accesses for GPGPU applications when compared to an "Arrays of Structures" (AoS) organization, typically used in standard softwares (as discussed f.e. in [Wm11], Chapter 31). All 4 arrays holding primitives' data will have the same length m, therefore we can (pre) allocate on the GPU the 2 output buffers $outputNormalsX$ and $outputNormalsY$ of m elements that will hold the normals data.

At this point, the 4 arrays need to be transferred to the GPU, with the consequent consumption of time. However, primitives' data on the GPU could be used also for rendering during the construction of the GVD using the method described in [HCK+99].

With this organization, we can compute the normal $Normal_i$ as

$$Normal_i = (PrimitivesY_2[i] - PrimitivesY_1[i], PrimitivesX_1[i] - PrimitivesX_2[i])$$

and therefore we can compute all the normals with a CUDA kernel that spawns m GPU threads, each thread computing a normal i according to the formula above. Figure 16 illustrates the behavior of this kernel. Accesses of this kernel are optimized since the input data is organized as a SoA, and memory writes are optimized since each thread only write two values at position i of $outNX$ and $outNY$, therefore accessing a contiguous region of GPU memory.
To retrieve the normals for each primitive, we can perform an exclusive prefix-sum of the array $NSegsPerPrim$ and store it in the new array $segmentsOffs$ that will then hold, at each position i, the starting index of the normals' points. Therefore, primitive i will have its normals stored in the range

$$outNX[segmentsOffs[i]], \ldots, outNX[segmentsOffs[i] + NSegsPerPrim[i]]$$

and

$$outNY[segmentsOffs[i]], \ldots, outNY[segmentsOffs[i] + NSegsPerPrim[i]]$$

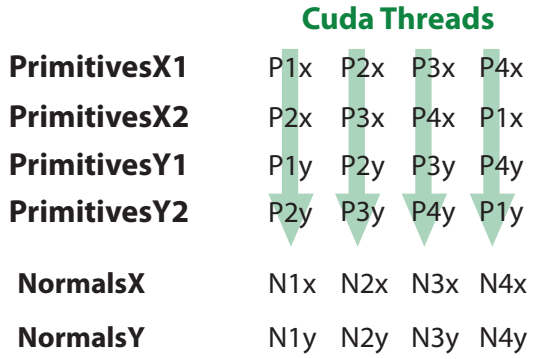


Figure 16: Computation of primitives' normals. Given our organization of the obstacles' data, each thread can read a pair of points from the same position i of the 4 arrays and then compute a normal. Each thread then will write the 2 values computed at the same position i of both $outNormalsX$ and $outNormalsY$. Thanks also to a SoA organization, each CUDA thread works independently from the others and performs efficient memory read/writes.

4.3.3 Computing Normals' Indices Associated to Points

From [Ger10] we can understand that the positions of the event-points on ECM edges, which inform us on the geometrical topology of the curves describing the edge, depend on the change of the normal from the Closest Point to an edge-point; each edge-point of the Voronoi diagram is generated by 2 obstacles, hence it is associtated with 2 Closest Points on 2 different obstacles, and an event-point is located wherever either of the 2 previously defined normals change. Originally, [Ger10] describes how to compute the set S of event points on one edge (e_i) as follows:

> First, we compute for each obstacle its normals. We associate normal n_i with edge e_i and vertex v_i with normals n_i and n_{i+1}. The event points then correspond to the points on the edge which intersect with the half lines starting at v_i with directions n_i and n_{i+1}, respectively.

The index i that refers to a normal of a closest obstacle to a point is what we call "normal index". In the current computation of an ECM, at this stage of the construction, an edge e is represented as a discrete list of neighboring points in our grid, hence we can consider e being be parameterized by $E[t]$, where $0 \leq t \leq 1$, having 2 closest points on its left and right $l[t]$ and $r[t]$ and 2 normals denoted as $n_l[t]$ and $n_r[t]$ and formed by vectors $E[t] - l[t]$ and $E[t] - r[t]$. For each edge-point, we consider the closest obstacle on its left and we assign to

the point an index $ni_l[t]$ if it lies in the vector space spanned by normal n_i and n_{i+1}, and the same we do for the index $ni_r[t]$ and the obstacle on the right of the point. Then, during tracing, the event-points on the edge occur when two consecutive points have either indexes different, which means, when $ni_r[t] \neq ni_r[t+\epsilon] \vee ni_l[t] \neq ni_l[t+\epsilon]$.
We can now recall that we have already computed a list of edge-points belonging to the edges, together with their color information and thus associated obstacles. We also have a description of the obstacles' data in terms of 4 arrays, $PrimitivesX_1$, $PrimitivesY_1$, $PrimitivesX_2$ and $PrimitivesY_2$, that sequentially list the x and y coordinates of the points defining the segments, and a description of the normals they induce in terms of the 2 arrays $outNX$ and $outNY$ that sequentially list the x and y coordinates of the normals associated to each obstacles' segments. Therefore, we notice that the computation of of the normal indices ni_l and ni_r for a point P involves

1. Checking the left color of P, and thus the obstacle associated
2. Iterating over the $1 \leq i \leq n$ normals associated with this obstacle, and for each pair of consecutive normals check if P lies inside the space they span; if it does, we assign i as left index normal, without continuing our iteration
3. Repeating the process from step 1 but checking the right color of P instead

Normal indices relative to "left" obstacles are written to $outNILeft$ while the ones relative to "right" obstacles to $outNIRight$. Assigning normal indices to m edge-points can be performed in parallel by m threads that read from array $edgePointsIDs$ at position i the (encoded) coordinates values of an edge-point, then read the $PrimitiveID$ of the left and right closest obstacles, respectively, from $EPColorsLeft[i]$ and $EPColorsRight[i]$, and then use $PrimitiveID$ to know at which positions to read the obstacles segments in the 4 arrays that describe the obstacles data, thus starting f.i. with the segment defined by points $(PrimitivesX_1[i], PrimitivesY_1[i])$ and $(PrimitivesX_2[i], PrimitivesY_2[i])$. In the end, each thread will perform the check of step 2, and then write 2 values, at position i of $outNILeft$ and $outNIRight$. Figure 17 illustrates the behavior of this kernel.
Adapting this parallel algorithm for GPGPU poses the problem that different threads will want to read at the same time the same obstacles segments values, and that in general the normals' coordinates are read not sequentially but in a sparse order, and both factors can prevent coalesced reads from happening, thus slowing down the whole process. This issue could be solved by "unrolling" the primitives segments in a couple of long lists with repeated elements to make sure that thread i processing edge-point at position $edgePoints[i]$ would read different memory locations from the ones read by thread $i+1$, although they can consist in the same segments data; the same should be done with the normals' coordinates. The so-organized data could be arranged on the CPU, but this would cost additional memory transfers between CPU and GPU, or it could be created in parallel directly on the GPU, but once again, since the values would be then read in the same sparse manner, we will incur in the same penalties. Therefore, it would make sense to simply perform, at the same time, all the other remaining operations.
Until now, we described the computation of normal indices associated to edge-points, but in fact we have to perform the same operations also for vertices: in this latter case, the procedure is totally similar, thus the same parallel algorithm will be used, with the only difference that every GPU thread will read 4 colors and thus assign 4 indices; moreover, normals indices for vertices will all be written to the same array $outNIVertices$.

4.3.4 Computing All the Needed Closest Points on Obstacles

To compute the Closest Points on each obstacle to a point P belonging to the Voronoi diagram we have to recall that the color data associated to P indicates which primitives contributed in "generating" the point, which means that P is the point with the minimum distance to those primitives, according to the definition of the Voronoi diagram. Hence, the Closest Points to P should lie on those obstacles. A point P can have up to 4 different

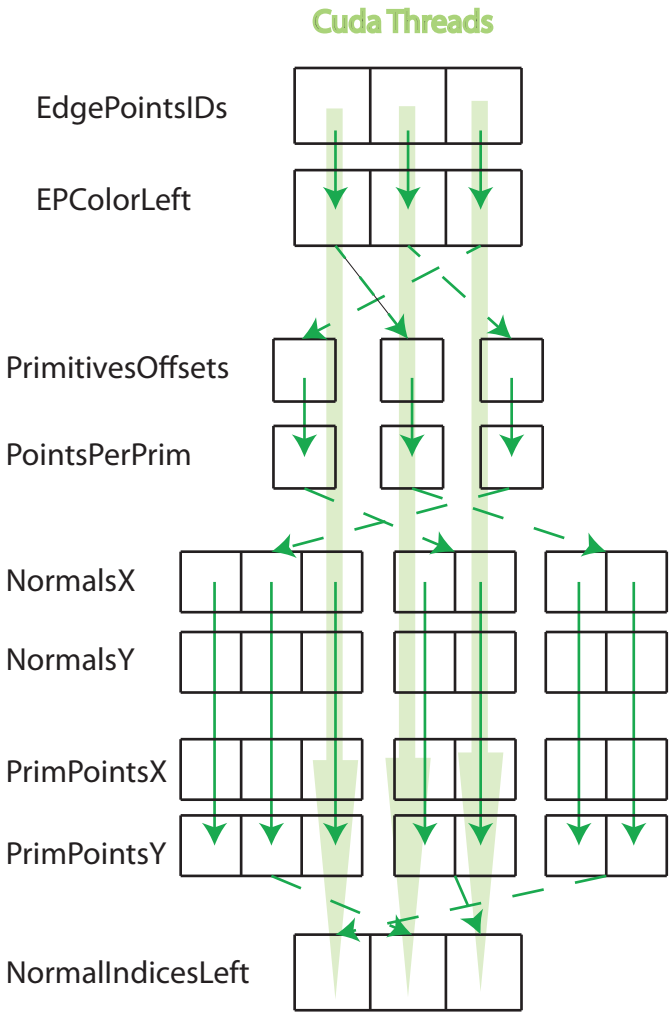


Figure 17: Computation of normals' indices associated to edge-points. Each CUDA thread first reads an input point (grid ID) and an associated color (left, for example). From the color ID, the thread extrapolates where to find the coordinates of the primitive associated with the color by reading a $primitiveOffset$ and a $nPrimitiveSegments$ from 2 separate arrays: now the thread knows the memory locations to read the primitive's segments and normals from. Normals and segments are then read and the computation of the associated normal index is performed. The index is then written to an output array. The same process is then carried out for the other colors associated to the point (1 more in case of an edge-point, 3 more in case of a vertex.
In the picture, continuous arrows indicate optimized CUDA reads, while dashed ones indicate scattered, non-parallel friendly reads (such as reading the obstacles data depending on the varying $primitiveOffset$ and $nPrimitiveSegments$.

neighboring colors, and thus up to 4 different points on those obstacles. The original implementation would then compute closest points by querying the obstacles' data corresponding to the colors information associated to the point P, iterating over the l lists of segments describing the l associated obstacles and then computing the closest points to P on each segment of the l lists, eventually keeping it as closest point if closer to P than the one considered as the closest until that moment; in the end, we will have the l closest points to each of the l primitives as a result.

It is clear that the whole computation of the closest points' to P depends only on the segments belonging to the l lists of segments $segmentsList_{i=1...l}$ representing the obstacles' data, which then can be then all merged into a single list $segmentsCP$ having m elements, where m is the sum of the elements in each list $segmentsList_i$. We can then observe that

the whole computation can be reduced to:

1. Compute the Closest Point on every segment of the list $segmentsCP$ to the point P together with its distance. These will be the lists $outputCP$ and $outputDistances$, both with m elements.

2. Group the $outputCP$ points and the $outputDistances$ values according to the primitive they refer.

3. For each group, select the Closest Point by considering the lowest distance value stored in $outputDistances$.

The above idea still holds if we had initially n lists $segmentsCP$, and therefore a total collection of $totalNSegments$, where $totalNSegments$ is the sum of the elements in all the initial lists. Step 1 can be executed in parallel by $totalNSegments$ threads at the same time, all reading and writing to distinct and ordered positions of input and output arrays. The 2 other steps are in fact reduction operations [MG11] and can benefit from parallelism too. However, we have to compute the closest points to not only just a point P, but for example to all the points in the set $pointsCP$: if $|pointsCP|$ is the number of points in $pointsCP$, we can still perform a parallel computation as outlined above by just increasing the number of threads. For each P in $pointsCP$ we will have a number n of associated primitives, equal to the number of distinct colors among the neighboring colors, and thus a total $totalNSegments$ associated to P. The input set of the entire computation is then the set $inputCP$ with total number of segments:

$$|inputCP| = \sum_{i=1}^{|pointsCP|} totalNSegments_i$$

and thus $|inputCP|$ is the total number of threads that we would require in order to perform a totally parallel computation.

Since we have all the Voronoi vertices and edge-points IDs computed, $inputCP$ can be easily derived: however, in practice, creating $inputCP$ has a cost, because we would like to instantiate it on GPU memory as well, and using GPU parallelism. Therefore, we could reconstruct it by executing $|inputCP|$ threads that in parallel take care of copying the right "portion" of segments data from $PrimitivesX_1$, $PrimitivesY_1$, $PrimitivesX_2$ and $PrimitivesY_2$, to the right position of the output arrays of unrolled data for Closest Points computation $OutDataCPX_1$, $OutDataCPX_2$, $OutDataCPY_1$ and $OutDataCPY_2$. At the same time, 2 additional output arrays will be created, $oudPID$ and $outPoint$, containing respectively, at position i the primitive ID associated with the segment defined by values at position i of the 4 arrays describing the primitives, and the original associated point (grid ID) from $inputCP$. Figure 18 illustrates the behavior of the kernel that unrolls the data in preparation of the computation of all the Closest Points. Figure 19 instead illustrates the following CUDA steps required to compute all the Closest Points.
The parallel implementation we have just described could suffer on the GPU because the values that describe the primitives' segments are not read in sequential, contiguous "bulks" (which allows coalesced reads) but sparsely (depending on the primitives associated to each point of $inputCP$), but this is un-avoidable. Moreover, since we are forced to perform sparse reads of the input $primitivesSegments$ for each $P \in inputCP$, incurring into thread serialization, the question of whether to perform already the computation of the closest point to P on any of the segments of the obstacles associated to it, instead of subdividing the computation in 2 separate steps (unroll/compute), arises. Therefore, we will keep the possibilities separated and then test both of them.

What is left to be discussed is for which points we should compute the closest points on obstacles, which means, what are the members of the set $inputCP$. All the vertices of the Voronoi diagram should be included, since we want to compute their closest points to either add them to the final ECM (if those points are also vertices of the Medial Axis) or to realize that those points are in fact not vertices of the Medial Axis (and thus of the final ECM). In addition, 2 edge-points that are part of the same edge, neighboring, but with a

different associated normal to an obstacle segment should also $\in inputCP$ since one of the 2 will be an event-point of the final ECM, and thus will have some associated closest points information.
We recall that our data is specified in the following arrays:

- $edgePointsIDs$ having n elements representing the x and y coordinates (encoded) of the point in the grid;
- $EPColorsLeft$ and $EPColorsRight$ having each n elements representing, respectively, the left and right color information of each edge-point;
- $outNILeft$ and $outNIRight$ having each n elements representing, respectively, the indices of the segments of the obstacle associated with the left, or right, color that generated the normal.

Then, we can proceed as follows:

1. We can sort all the vectors (they all have n elements) using the X coordinates encoded by the values of $edgePointsIDs$ as keys.
2. For each pair of consecutive values in the sorted $edgePointsIDs$ at positions i and $i+1$ we can check whether they are neighboring (consecutive x coordinates), if they belong to the same edge-point (same pairs of values from $EPColorsLeft$ and $EPColorsRight$) and if they have either of the indices stored in $outNILeft$ or $outNIRight$ different. If all these 3 conditions are satisfied, then the pair is marked to be inserted in $inputCP$.
3. Then, the 2 above points are repeated but the sorting is now performed using the y coordinates encoded by the values of $edgePointsIDs$ as keys.

Sorting, as well as sorting more vectors using the values of one of them as keys, is known to be a parallel friendly operation, also on the GPU [HSO07]. Regarding step 2, it can be performed in parallel by n threads, each checking a pair of consecutive values in the various arrays. On the GPU, in order not to have an overlap between the values that are read by different threads, it could make sense to first "unwrap" the values of the arrays (after they have been sorted) so that all the original pairs of values at position i and $i+$ are mapped to different but consecutive positions in the new arrays, that will then have $(n-1)*2$ elements. Once the edge-points are marked, they can be eventually compacted using the same stream compaction technique described at the end of Section 4.3.1.

4.3.5 Constructing the ECM

At this point of the construction, we have computed the coordinates of Voronoi vertices and edge-points, the normals to the segments describing the obstacles, the indices of the associated normal to the closest obstacle's segment for each vertex and edge-point, and the location of closest-points for vertices and event-points. According to [Ger10], this is all the information we need to compute an Explicit Corridor Map from a GPU-rasterized GVD of the footprint information, and thus we proceed to transfer this information to the CPU from the GPU. It is important to notice that, if we exclude the transfer of the primitives' data from the CPU to the GPU (required also for rendering of the Voronoi diagram on the GPU), this is the first and only data we will transfer between the CPU and the GPU. Hopefully, this amount should be less than the entire size of the frame buffer (as it was instead in the original implementation that did not take advantage of GPGPU) and thus the whole transfer should be quicker.

On the CPU, we will fill an instance of the new object ClosestPointsCache with all the freshly transferred information. We will describe ClosestPointsCache better later in this work, but essentially, it uses a hash-map to associate points (in the form of integer grid coordinates) with their "local directions", color information, closest points on obstacles and normal indices. Therefore, to construct the final ECM, the same operations that occurred in

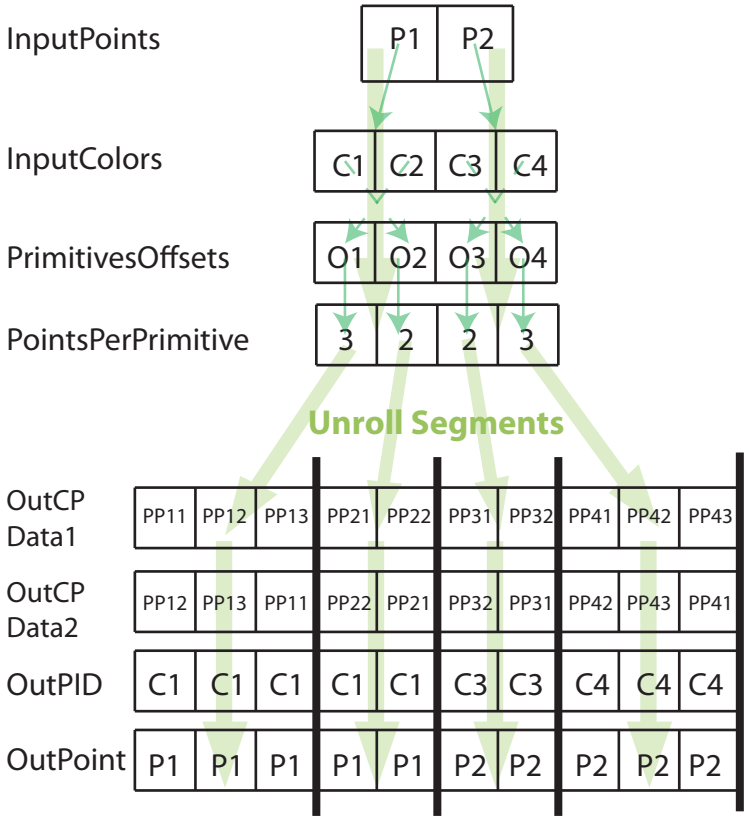


Figure 18: Unrolling data to prepare a Closest Points computation. Each CUDA thread reads an input point (edge-points in the picture) and the associated color data (2 colors per point in the picture). Each color value identifies a primitive; therefore, both a $primitiveOffset$ and a $nPrimitiveSegments$ are read from the respective arrays, in order to be able to locate the primitives segments in the arrays that hold the description of primitives. The goal of this kernel is to fill up the 4 arrays $outCPData1$, $outCPData2$, $outPID$ and $outPoint$: the first 2 arrays will receive the points describing the primitives associated with the colors read by the thread, while $outPID$ will receive the associated primitive ID (mostly the color bound to it) and $outPoint$ the ID describing the original input point. It is clear how to those 2 latter arrays some repeated sequences of input points/primitive IDs, all equal, will be written; this data repetition will allow uniform memory accesses during the following CUDA step.
In the picture, the green dashed lines indicate scattered (and thus not efficient) memory accesses, while continuous ones indicate uniform memory accesses to the buffers.

the original framework will be performed, but nothing more will be computed, only a hash-map will be queried. In the end, we are maintaining the naturally "branching" algorithm that assembles an ECM on the CPU, but having nonetheless already performed all its major related computations on the GPU, and keeping all the necessary data in a memory-inefficient but fast to query structure such as a hash-map.

4.4 GPGPU optimization notes

The design and implementation of the various computational steps described in the previous section on the GPU using the GPGPU paradigm derived (and took advantage) from a number of important characteristics of the model. Although some of them have already been introduced at the beginning of this chapter, in this section we will elaborate on them, and at the same time we will outline in which parts of the computation they are used. Notice that the CUDA-specific considerations refer to the SDK version used for the development and testing of the implementation, which is, version 5.0 of the NVIDIA CUDA Toolkit.

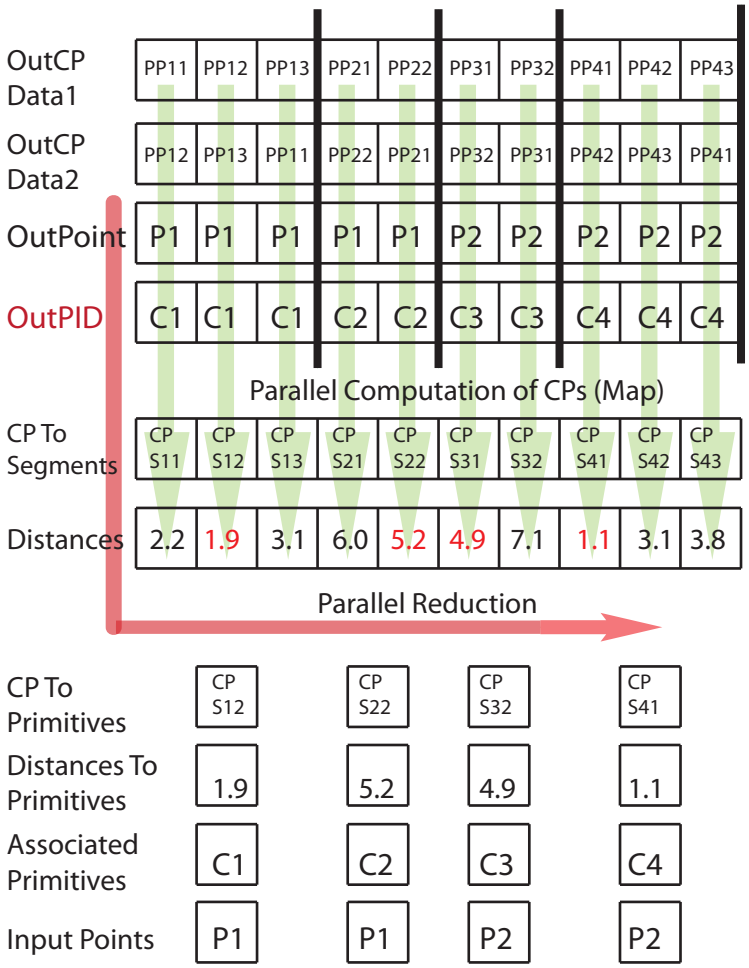


Figure 19: Computation of closest points. Each CUDA thread starts with reading 4 values at position i from the 4 arrays arranged during the previous unrolling step: the first 2 values read are 2 endpoints of a segment S, the third value was the original input point P and the fourth is the ID of the primitive S belongs to. Hence, the thread can now perform a closest point computation, computing CP, the closest point to P on S, and the distance D, which will be written at position i of $CPToSegments$ and $Distances$, respectively. We recall that the new distance and closest point can still be easily associated with the original point P (position i of $outPoint$) and primitive ID (position i of $outPID$).
Since, for each input point P, we are interested in the closest point to all the associated primitives, we can now sort the values of $outPoint$, $outPID$, $CPToSegments$ and $Distances$ over the values of $Distances$. Moreover, we will perform a "segmented sort" operation using the content of $outPID$ to delimit our segments: performing a segmented sort will make sure that we will order, independently, all the values that belong to the same primitive, having thus the closest point (and the associated distance) to each primitive. We can further process the output to extract only the first elements of each segment, ending up with the 4 final arrays shown in the picture. The dimensionality m of those arrays will be smaller of the input one, but we have anyway everything we need (input point, primitive ID, output data) to reconstruct the closest points information from them.
As a result of the previous pre-processing, this whole GPGPU computation involves parallel-friendly calculations and memory accesses. We can also notice how this step follows the typical parallel pattern "map" (each threads maps a distance and a closest points to the 4 input values) and "reduce" (the segmented sort, plus filtering of elements with lowest distances per segment).

4.4.1 Use of Texture Memory

Textures come from the domain of computer graphics. They can be bound to memory buffers and instantly provide some benefits:

- Texture queries are cached, and optimized for 2D spatial locality, although Texture contain only 8 KB of cache per SM

- Textures have some limited processing capabilities that can efficiently unpack and broadcast data (hence, a single $float4$ texture read is faster than four separate 32-bit reads)

- Texture have separate 9-bit computational units that perform out-of- bounds index handling, interpolation, and format conversion from integer types ($char$, $short$, int) to $float$.

On the other side, GPUs can only use a limited number of Textures at the same time, and can pose additional requirements on the type of memory Textures can be bound to (mostly alignment). Also, Textures are read-only, although newer GPUs provide writing-to-texture capabilities using "Surface" Textures, assumed that a few additional conditions are met. As a GPGPU layer, CUDA comes with a number of functions that allow programmers to take advantage of Textures, which can then be bound only to GPU Global Memory; about the caching capabilities of Textures both CUDA and NVIDIA GPUs specifications are not totally clear, but what is clear is that, for example, a single thread reading neighboring locations of a 2D buffer in Global Memory representing a 2D space, like (x, y), $(x + i, y)$, $(x, y + i)$, $(x - i, y - i)$, should incur into a speed-up if the 2D buffer is read through a 2D Texture fetch, although for which values of i this holds is unclear, and it seems to largely depend on the hardware. Literature shows that a smart use of Texture memory alone can lead to significant improvements [GM07]. Examples of algorithms that usually benefit from the use of Texture memory include graphical filters used for image processing/computer vision, but also simulations in 2D/3D spaces which involve continuous propagation of information to neighbors.

We can now observe that checking the degree of our grid-points after a rasterization of a GVD involves reading, for each thread associated with the point at location (x, y), the color values at memory locations (x, y),$(x - 1, y)$,$(x, y - 1)$,$(x - 1, y - 1)$, and this is one of the memory access patterns which benefit from the use of Textures. Hence, our CUDA implementation will initialize 2D Textures to read the input color data and, in subsequents steps, to read the "markers" from buffers $verticesMarkers$ and $edgePointsMarkers$. It is important to notice that, for example, when marking points of the grid depending on the color data, we can not bind $verticesMarkers$ and $edgePointsMarkers$ to Textures since Textures are read-only.
The GVD color data is rendered to a simple OpenGL renderbuffer: the current CUDA API version defines a rather mature set of calls that allow to directly manipulated memory created inside OpenGL (or DirectX) contexts with limited overhead. Therefore, to use the input color data we only needed to associate the OpenGL context and the OpenGL buffer index with the CUDA context, and then ask CUDA a "handle" to this GPU memory.

Some additional Textures-related optimizations could be tried:

- Surface writing could be added to allow write-to-texture. The current implementation seems to fulfill the requirements for Surface writing; however, only modern GPUs/drivers support it.

- New GPUs and GPU APIs permit what is called "bindless use" of Textures, which allows to fetch Textures' values without performing a bind operation before. However, using bindless Textures seems to save only a couple of nanoseconds per Texture bind, therefore it probably makes a difference only in those scenarios that involve long sequences of kernels operating on Textures.

4.4.2 Coalesced reads/writes and Structure of Array

As we have already explained, a coalesced memory access means that the hardware can coalesce, or combine, the memory requests from different threads that are part of the same warp into a single wide memory transaction. Clearly, coalesced accesses make a big difference in performance. However, coalesced accesses are a somehow complicated subject in CUDA since GPUs have different warp sizes, alignment requirements, and in general newer GPUs are becoming better at performing coalesced transfers even when the memory accesses

of threads are not perfectly aligned, because of the additional cache levels introduced. As a simple rule, threads with consecutive thread indices that read consecutive memory addresses located into a 128-bytes aligned range of addresses should trigger a coalesced access. Figure 20 shows some examples of coalesced memory accesses while Figure 21 shows some examples of non-coalesced memory accesses.

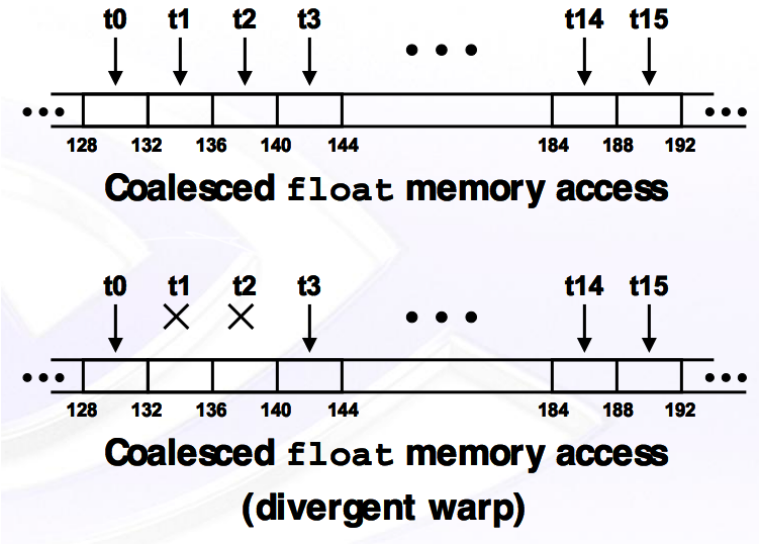


Figure 20: Examples of coalesced CUDA memory accesses (adapted from figures of the CUDA notes of the Oxford Computing lab).
Top: A sequence of consecutive threads belonging to the same CUDA warp read a sequence of contiguous values.
Bottom: On most (recent) GPUs, even if some of the threads belonging to the same CUDA warp do *not* read some of the values of the sequence, the memory reads can still be coalesced.

The benefit of coalesced accesses leads to the necessity of organizing data as a Structure of Arrays (SoA) instead of Array of Structures (AoS): traditionally, for example, when a structure *Point* having two float elements, x and y, is created and then concatenated in a sequence of n elements, we usually end up with a long buffer with the layout $(P_1X, P_1Y, P_2X, P_2Y, \ldots, P_nX, P_nY)$, and thus literally, an array of the various structure elements. Instead, a SoA organization would lead to a memory layout like $(P_1X, P_2X, \ldots, P_nX, P_1Y, P_2Y, \ldots, P_nY)$: this organization is in fact a number of separate arrays (2 in our case) that form a complex structure together, therefore those arrays could be split in different memory buffers (as in our case, where for example $outNX$ and $outNY$ are distinct GPU buffers). With a SoA organization stored in two separate arrays PX and PY, threads $(t_i, t_{i+1}, t_{i+2}$ that want to read each one point location will respectively read positions $(PX[i], PX[i+1], PX[i+2])$ for the x coordinate and $(PY[i], PY[i+1], PY[i+2])$, resulting in both reads coalesced. The same would not happen with the AoS layout since, for example, t_i, t_{i+1} would not read neighboring memory locations when reading the x coordinates alone.

Around the CUDA API an entire ecosystem of libraries exists, both maintained by NVIDIA and the community. The Thrust library [HB10] is a collection of GPU-accelerated primitives and helper functions that also tries to "enforce" a SoA organization of the data; it was used by our implementation in the steps that compute the primitives' normals, normal indices associated to points and Closest Points on obstacles.

4.4.3 Shared Memory and Optimized Scan/Sorting

Again, we have already introduced Shared Memory, which consists in either 16 KB or 48 KB per SM arranged in banks that are 32 bits wide, shared among threads of the same

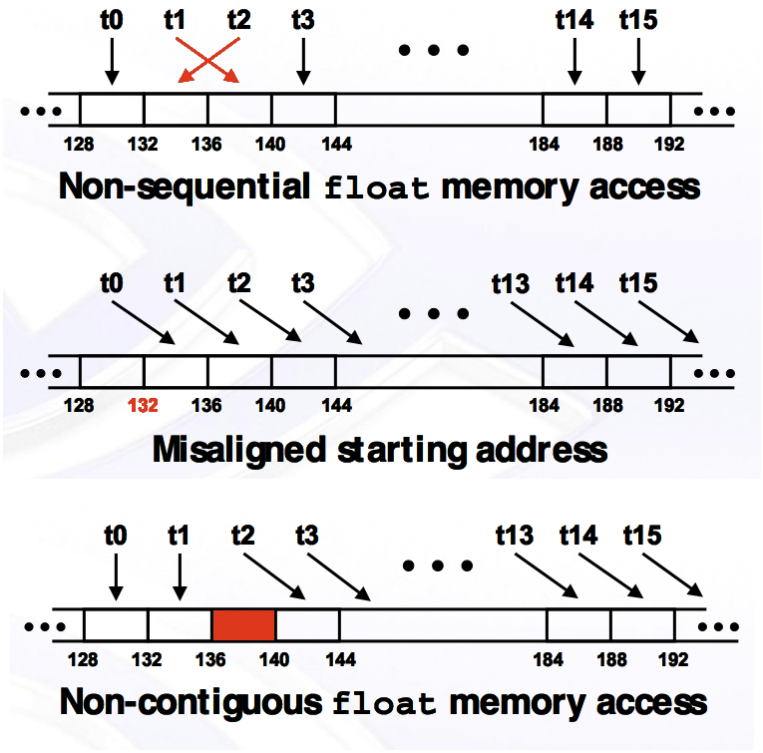


Figure 21: Examples of non-coalesced CUDA memory accesses (adapted from figures of the CUDA notes of the Oxford Computing lab).
Top: Not all the threads of the warp access elements sequentially, breaking the required uniformity. This access can be referred to as "scattered" access of values in a buffer.
Middle: Memory alignment is crucial when it comes to coalescing. A simple guideline to follow (which guarantees coalesced accesses on most CUDA capable GPUs) is to make sure that the address of the value read by the first thread of the warp is located at an address (in bytes) which is a multiple of 32 (in the picture, the location with address 132 is read by t_0 as opposed to the location with address 128, which is a multiple of 32 instead.
Bottom: Thread t_2 breaks the sequence of contiguous accesses by skipping "its" value and reading instead a neighboring value in the buffer. Notice the difference between this case and the one displayed in the lower part of Figure 20.

block, and for which memory accesses from threads of warps belonging to the Block should be always coalesced. However, when multiple requests are made by different threads for data within the same bank, the so-called "bank conflicts" can occur. These requests can either be for the same address or for multiple addresses that map to the same bank. When a bank conflict happens, the hardware serializes the memory read/write operations. Moreover, benchmark show that, contrary to early NVIDIA documentation, Shared Memory is not as fast as register memory, registers which are instead more and more present in newer GPUs. This, together with the difficulty in using it effectively, makes Shared Memory less and less crucial in achieving high performance.
This being said, the use of Shared Memory can be very important in maximizing the work-efficiency of algorithms, especially if they are expressively parallel: a common pattern is to first load data from Global to Shared Memory, then perform many operations per single thread of each block reading/writing to shared memory (thus taking advantage of Shared Memory during the repeated reads/writes) and only at the end writing data from Shared Memory back to Global Memory. This pattern is illustrated, for example, in [HSO07] that describes among others an efficient GPGPU implementation of the parallel friendly "prefix scan" operation.
Operations like "prefix sum" and sort, mostly "radix sort", derive their efficient parallel implementations from the ideas behind parallel prefix scan: they become then important "building blocks" for parallel algorithms, included our implementation. The prefix sum and

sorting implementations we used are in fact fairly advanced GPGPU algorithms that have been carefully shaped and thought to take advantage of various hardware characteristics (as explained in [MG11]), going significantly beyond the sole correct use of the different types of memory. The main and general ideas behind them are:

- **Reducing Aggregate Memory Workload:** many optimized scan algorithms [HSO07] would usually try to distribute the total workload into chunks that would be then independently submitted to SMs of the GPU, since they could be elaborated in parallel. Instead, [MG11] evolves from a simple "two-level reduce-then-scan" approach, that does not scale only by allocating one thread per input element and then partitioning them in chunks, but instead allocates a fixed-size grid of C threads per input element that are kept and reused through the whole computation. Therefore, partial sums are executed with results stored in threads' registers instead of global/shared memory, reducing the cost of reads/writes. Also, less CUDA launches are performed in total since threads are kept and reused, and since each thread performs many different parts of the computation, the total workload results more balanced and resources end up being distributed better.

- **Employing Specialized Local Multi-scan Algorithms:** for the scan phase, a new GPGPU specialized variant of the general multi-scan pass was developed. It features a better use of threads' registers, the encoding and unrolling of indices data that avoids some memory reads, and specific intra-warp communication.

- **Flexible Algorithm Serialization:** this characteristic is mainly a consequence of the previous two. In more classical [HSO07] or histogram-based approaches [DCSM96], different phases are carried out in parallel but subsequently require to "communicate" with the following ones, mostly in terms of writes and subsequent reads of Global Memory. In [MG11] this is not needed anymore, permitting at the same time a more dynamic distribution of the workload, since many dependencies between the "chunks" of data that need to processed cease to exist.

The routines described and tested in [MG11] are the result of a collaboration between the University of Virginia and NVIDIA itself. The code resulted from the research is part is part of the Thrust project since its version 1.3.0. Version 1.5.0 of Thrust is currently shipped also together with the latest CUDA Toolkit and therefore was the one employed by this work.

4.4.4 Use of Pinned GPU Memory

Through the entire work, many words have been spent elaborating on the correct use of GPU memory and its different types. However, it is well-known that moving data from/to the GPU is by itself a very important aspect, often representing a bottleneck of computations. This can be very significant in our case because our process includes a long phase performed on the CPU, and at the same the GPU computations are not "massive" as in many other cases involving GPGPU. Therefore, the un-avoidable cost of memory transfers could end up not being counterbalanced by the parallel speed-up.

GPGPU offers some ways to mitigate these costs, the easiest one being the use of the so-called "Pinned", or page-locked, host memory. The immediate speed-up comes from the fact that in systems with a front-side bus (as many modern ones are) the bandwidth between host memory and device memory is higher if host memory is allocated as Pinned. The downside is that page-locked memory is generally quite scarce, and that reducing the amount of available physical memory can degrade performance of the whole system. Page-locked memory is also required to perform asynchronous transfers between GPU and CPU, or to be mapped directly to the addresses of the device (eliminating then the need to copy it to or from the device).
In our implementation, all the CPU buffers that will at some point hold data transferred from the GPU will be allocated as Pinned host memory, if possible. Pinned memory can be marked as " Write-Combining" memory too, which leads to additional speed-ups if the

memory is only written, but this never applies to any buffer of our implementation that will receive data from the GPU. It is unclear whether the use of Mapped memory leads to speed-ups. especially when it comes to recent hardware or hardware with dedicated memory: in the end, we decided not to add it since it would have required additional work on the code. Another related further optimization that could be to investigate is the integration of asynchronous data transfers in order to allow the execution of memory transfers and CPU code at the same time, or even, on some hardware, the execution at the same time of memory transfers and GPU calculations. However, including asynchronous memory transfer would require to re-think the general structure of the entire approach, and, therefore, it was left out.

5 From a Single-tiled to a Multi-tiled construction

Currently, the ECM framework incorporates a "limited" multi-tiled creation approach: since the total resolution is bound to the original GPU resolution of the Voronoi construction, the method can perform multiple rasterization passes to render high-resolution Voronoi diagrams. However, after each pass, the framebuffer data is moved back to CPU and concatenated with the previously transferred data, if any, in the same buffer. We have already showed that the use of a big CPU buffer is a limit, and how we can remove this requirement by performing for example more operations on the GPU, as the previous chapter describes. This, in turn, fairly naturally leads to the idea of subdividing the computation of a high-resolution ECM into the computation of many, lower resolution Voronoi diagrams, which could be then used to compute a final, high-resolution.
This new approach involves changing the original tracing algorithm since it involves, for example, a more careful treatment of the borders. Also, merging the partial Voronoi diagrams together with their associated information (Closest Points) into a final ECM is not straightforward; this chapter will address these issues.

5.1 General Overview and Preliminaries

We consider a visual representation of a high-resolution Voronoi Diagram as our starting point, representation that can be easily rendered according to the method described in [HCK+99] together with the movement of the OpenGL viewport that allows rendering its multiple parts. Again, visually, we can subdivide this pixel-approximated Voronoi Diagram into neighboring tiles, by just partitioning the pixels into squares, or rectangles. We will use squares for simplicity, in this work. We imagine to define a grid over the entire initial visual representation of a generalized Voronoi diagram in the same way we did in the previous chapter (Figure 11). This pre-defined grid can be applied, in the same way, too separate tiles as well.
At this point, we assume to be able to "trace" (which means, to compute a geometrical definition of the Voronoi diagram) each tile independently, and then to be able to "merge" those descriptions into a final ECM, which can be defined as an "augmented" Voronoi diagram. Therefore, we also compute and store the Closest Points information, thus having at the end a sort of "encoded" description of the ECM of that tile. It is important to notice that the final ECM would be exactly equal to another ECM computed in a single-tiled "fashion" (for example using the original framework); however, a single-tiled computation is in fact often not possible due to technical limitations (maximum CPU buffer dimensions, maximum size of a GPU render target).

A first possible subdivision of the tiles is shown in Figure 22b: there, each pixel is assigned to only one tile. This subdivision has an important problem: some points belonging to the left and bottom borders of the tiles will have 2 or more missing neighboring colors, which is a known problem also in the original method However, in the original framework the left and bottom borders are always "outmost" borders, and thus it is assumed that ignoring that missing information would be harmless. Unfortunately, this is not the case anymore now for most of our tiles, where left or bottom borders are mapped to central points of the initial grid.
Hence, we will adopt a different subdivision, shown in Figure 22c: there, pixels that are located on right and top borders of tiles will be also part of the left and bottom borders of those tiles that will share that common "boundary", if any. If we assume squared tiles with $tileWidth == tileHeight$, and consider two tiles t_0 and t_1 "sharing" a vertical border, the following property will hold for point p_0 located at coordinates $(tileWidth - 1, y_i)$ of tile t_0 and point p_1 located at coordinates $(1, y_i)$ of tile t_1:

$$ColorRU_{p0} == ColorLU_{p1} \wedge ColorRD_{p0} == ColorLD_{p1},$$

and similarly, if the two tiles share instead an horizontal borders, for two points with the same x_i coordinate belonging to different tiles but lying on the border will hold that

$$ColorRD_{p0} == ColorLD_{p1} \wedge ColorRU_{p0} == ColorLU_{p1}.$$

This means as well that point q_1 belonging to tile t_1 located at coordinates $(0, y_i)$ is mapped to the same grid point mapped by p_0 belonging to t_0: however, q_1 would miss at least 2 among the neighboring color information. By mapping some points of the tiles to the same points of the grid we indeed "waste" some space, but this allows us to have no "gaps" in the color data information between neighboring tiles. To achieve this subdivision, the rasterization part of the method should be changed so while moving the OpenGL viewport a stripe of overlapping pixels is maintained.
Once the color data of a Voronoi tile is produced, we can proceed with "tracing" it.

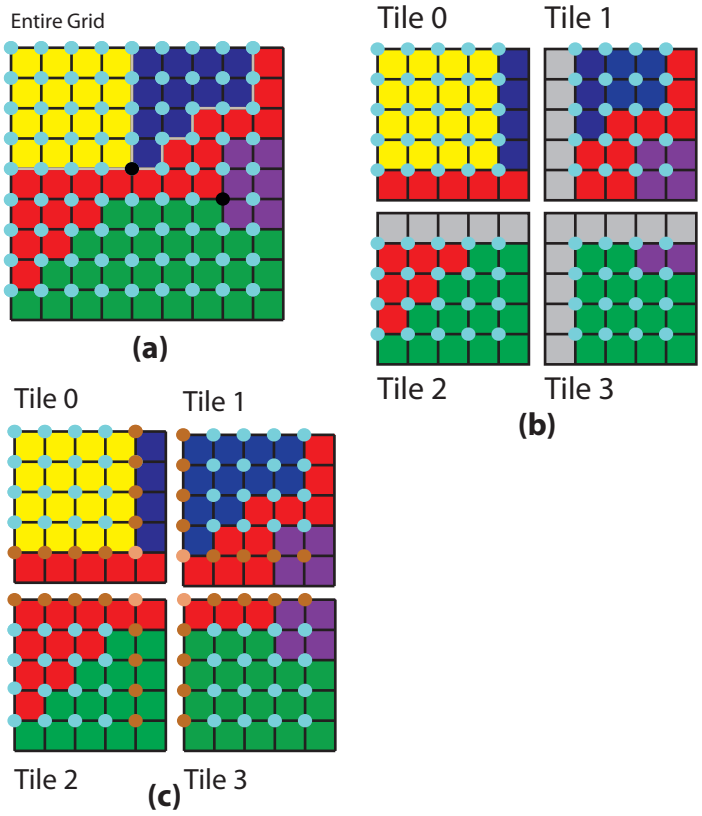


Figure 22: Discrete input grid and subdivision into tiles.
(a) An example of color data representing a GVD, showing the locations of the points. Vertices of the GVD are marked in black. The input is the same as the one depicted in Figure 12.
(b) The same input subdivided into 4 tiles, with no overlapping of pixels among different tiles. We can notice that every point of any tile (light-blue) is mapped to a distinct point of the original input. However, some points lying on the borders of tile 1,2 and 3 have some missing neighboring colors, since those pixels in fact belong to a different tile.
(c) The same input subdivided into 4 tiles, with tiles featuring some "stripes" of pixels in common. We can notice that the points that lie on a border "shared" by more than 1 tile (shown in brown) are in fact *not* mapped to distinct points of the input grid (which means, some input points are mapped to a more that 1 point on more than 1 tile), therefore, we are "wasting" some memory with storing this information more than once. We will not process some of those points (the ones having missing color information) in order to have a one-to-one correspondence between points of the original grid and points of the tiles.

5.2 Tracing a Tile

Originally, tracing consisted of basically two steps: finding the starting tracing points (the Voronoi vertices) and then subsequently tracing from those points the Voronoi edges that would form our resulting graph. The main difference now is that part of a Voronoi edge can lie inside a tile without any of its points being a Voronoi vertex.

The original method that searched for starting points (Algorithm 3) simply iterated over all the points of the grid and stored those that featured at least 3 unique directions,

Algorithm 3 FINDVORONOISEEDS(*colordata_buffer*, *tileWidth*, *tileHeight*).
Input: Pointer to a CPU buffer of size *tileWidth* * *tileHeight* * sizeof(color_type) holding the GPU color data.
Output: The array containing the vertices found.

```
1: outVertices ← []
2: for x = 1 →tileWidth-1 do
3:   for y = 1 →tileHeight-1 do
4:     vertex ← MAKEVERTEXFROMCOLORDATA(colordata_buffer, x, y)
5:     dirs ← GETDIRECTIONS(vertex)
6:     if dirs.length > 2 then
7:       udirs ← GETUNIQUEDIRECTIONS(vertex)
8:       if udirs.length > 2 then
9:         outVertices.APPENDVERTEX(vertex)
10:      end if
11:    end if
12:  end for
13: end for
14: return outVertices
```

which in our discrete case identifies Voronoi vertices. From those points the "tracing" of edges started. An edge was then considered finished when it reached either another point featuring 3 or more "unique directions" (a Voronoi vertex), another point featuring 1 unique direction only (mostly a concave corner of an obstacle) or a point lying on the border of the grid.

The main problem with adapting this method is that we should consider as starting points also normal "edge-points", thus points that feature only 2 unique "directions", but lie on the borders of our tiles, otherwise we could end up not having any point to to start tracing those edges that do not lie completely inside the tile (Figure 23a). Moreover, we will include also those points that feature only 1 unique direction but include "black" among their color information, signaling that they are endpoints of edges running towards an obstacle (because of the color black) and touching a "convex" corner (because they feature an "unique" direction). We can correctly locate all those points since our subdivision of the tiles involves no missing color information. Algorithm 4 shows the new routine FINDVORONOISEEDSUPDATED.
After all those points have been gathered, we can proceed with tracing the edges. Because of the possibility of incomplete edges, we will have to treat the set *borderPoints* separately. Hence, the tracing step needs to be further subdivided into two sub-steps:

- First, we will identify "standard" Voronoi vertices and "border" vertices, points that are, topologically, only edge-points but that are close to a border of our tile. We will use points in both sets to start tracing edges
- Then, we will trace edges in the same way the original implementation did
- In the end, we will trace additional edges starting from "border" vertices. In this phases, edges will be traced different, since additional checks must be performed

5.2.1 Tracing "standard" edges

During this step, each of the points in *outVertices* is used as starting point for tracing an edge of the diagram. This is step is the same as the original one, hence edges are constructed in the same way, with just a difference: when we have to decide upon a stopping point, we treat points that are 1 pixel away from borders as if they were touching the border itself. For example, we consider reaching a point with coordinate $x == 1$ a stopping point as if it was lying on the a left border of a tile, although points that really lie on the left border have of course $x == 0$. This way, edges belonging to different tiles will not include those points

Algorithm 4 FindVoronoiSeedsUpdated($colordata\_buffer$, $tileWidth$, $tileHeight$).
Input: Pointer to a CPU buffer of size $tileWidth$ * $tileHeight$ * sizeof(color_type) holding the GPU color data.
Output: The arrays containing the vertices found and the edge-points touching any of the borders that we should also use as starting points of tracing.

```
1: outVertices ← []
2: borderPoints ← []
3: for x = 1 →tileWidth-1 do
4:   for y = 1 →tileHeight-1 do
5:     vertex ← MakeVertexFromColordata(colordata_buffer, x, y)
6:     dirs ← GetDirections(vertex)
7:     if dirs.length > 2 then
8:       udirs ← GetUniqueDirections(vertex)
9:       if udirs.length > 2 then
10:        outVertices.AppendVertex(vertex)
11:      end if
12:    else if vertex.x == 1 ∨ vertex.x == tileWidth − 1
       ∨ vertex.y == 1 ∨ vertex.y == tileHeight − 1 then
13:      if dirs.length > 0 then
14:        udirs ← GetUniqueDirections(vertex)
15:        if udirs.length == 2 ∨ (udirs.length == 1 ∧ PointHasBlack-
           Color(colordata_buffer, x, y)) then
16:          borderPoints.AppendVertex(vertex)
17:        end if
18:      end if
19:    end if
20:  end for
21: end for
22: return outVertices, borderPoints
```

that are not uniquely mapped to other points of the global grid. Moreover, every time an edge-point is added to any edge, the list *borderPoints*, which is in fact implemented as a C++ std::map, is queried for the existence of that edge-point, and in case it exists, it is removed. We do this because we know that each point can belong to up to one Voronoi edge, hence if any edge-point gets assigned, there is no need to process it further. Unfortunately, it is important to observe that the additional lookup of the *borderPoints* array adds to the tracing a non-negligible computational cost of $log(m)$ for each edge-point found to belong to an edge, where m is the number of elements in the ordered list *borderPoints*.

5.2.2 Handling border points

Points in the list *borderPoints* could be part of partial edges which would not have been traced by the original tracing routine. Since we stored these points, the easiest thing to do now is to iterate over them and to use them as "starting points" for tracing new edges. A point $p_1 \in borderPoints$ features either exactly two unique directions or one unique direction and the color "black" among its neighboring colors: we first ignore the directions that point "outside" the tile (eg. the direction "right" assigned to a point with coordinate $x == tileWidth - 1$). Then, we analyze the remaining directions *dirs*:

- First, if a "perpendicular" direction *perpDir* exists in *dirs*, which is, a direction pointing instead "inside" the tile, we start tracing an edge starting from point p_1 and direction *perpDir*. We finish tracing when any of the original stopping condition is satisfied, and during tracing, we again make sure to remove the edge-point that we encounter from the list *borderPoints*. We will name an edge traced this way *perpendicularEdge*. Figure 23a shows an example of "perpendicular edge" traced from a point $\in borderPoints$.

- Then, we check if a particular "parallel" direction *parallelDir* exists in *dir*: parallel directions are "up"/"down" for vertical borders and "left"/"right" for horizontal borders. We always check the existence of either "right" for horizontal borders **or** "up" for vertical borders. If a parallel direction exists, we start tracing an edge starting from point p_1 and direction *parallelDir*, again removing encountered edge-points from *borderPoints*, until we encounter p_2 satisfying any of the stopping conditions. Eventually, we will name this edge *parallelEdge*. The edge pointing upward in Figure 23 is an example of "parallel edge".

If we end up with **only either** a *perpendicularEdge* or a *parallelEdge*, we add it as an edge of the Voronoi diagram. If instead **both edges** have been traced, we can observe that they are in fact part of the same edge: if they were not, point p_1 would feature 3 unique directions instead, and thus would not be in the *borderPoints* (but instead would be a member of $outVertices$). Therefore, we merge the 2 edges into one and add this final edge to our diagram. Figures 23a and 23b show the first and second scenario, respectively.
It must be noted that, in order to trace a *parallelEdge*, we only checked the presence of **one** particular parallel direction, while the point could in fact feature 2 parallel directions. However, we keep the list *borderPoints* sorted lexicographically, and by checking only the direction in which the border points are sorted, we ensure that all the points that should belong to a parallel edge are processed in an order that allows to all of them to be added, and in the right order. In fact, if we take two points that should belong to a parallel edge p_{i-1} and p_i and we are currently processing p_i, we can observe that:

- If point p_{i-1} featured 3 or 4 unique directions, it would be a member of $outVertices$, which are processed during the previous step, and because of the nature of edge tracing, p_i would be already a part of an edge including also p_{i-1}.

- If point p_{i-1} featured 1 or 2 unique directions, it would have been a member of *borderPoints* since we have assumed it to be also a member of the same parallel edge, and in such edges all points have the same value for either the x or y coordinate. By checking always the "increasing" direction for each border point, we know that in this case p_{i-1} would have been, again, already processed and would be thus already part of an edge that will include also p_i, since border points are stored and processed in lexicographical order.

During the processing of *borderPoints* we make sure not to add any edge made of only one point. Once it is finished, we iterate again over the leftovers (if any) to add as edges with $length == 1$ the points that feature at least 1 unique direction pointing "outside" the tile. Any additional point not satisfying this condition probably signals an error in the generation of the Voronoi color data, since it highlights an edge-point belonging to an edge which should lie completely inside the tile, but that it was not surrounded by any other edge-point/Voronoi vertex with "compatible" color data and thus compatible unique directions information. We can choose to report these points and/or ignore them.
After the second step is completed, we can assume to have a description of all the edges that either completely or partially lie inside the tile.
Regarding the computational complexity of the whole step, we are performing $O(m)$ iterations over the m elements of the STL map *borderPoints* to perform every time the tracing of one edge, to which we assign constant complexity.

5.2.3 Organizing edges of a tile

It is useful to the separate edges that lie completely inside a tile from those that do not, since only the latter ones will eventually have to be processed during merging. Hence, for each tile, we will create 4 new lists of edges, one for each border, where we will store the edges that have one (or both) of the edge points touching the associated border. We will call those lists topBorderEdges", "bottomBorderEdges", "leftBorderEdges" and "rightBorderEdges". Edges of those lists will be indexed by the value of the y coordinate (for the lists "leftBorderEdges" and "rightBorderEdges which are associated to vertical borders) or

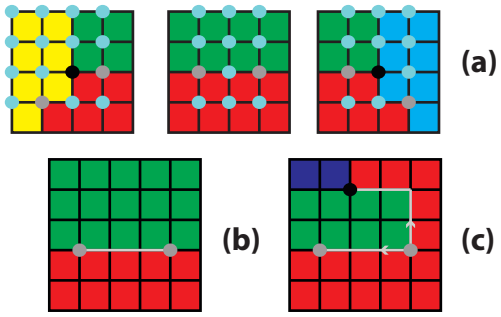


Figure 23: Handling border points.
(a) If we consider the input GVD described by the 3 tiles merged together, we can notice an edge connecting 2 Voronoi vertices locate inside the left-most and right-most tiles (black points), running through all the 3 tiles. However, if we consider only the color values of the tile in-between, we can notice that no Voronoi edge would be normally traced from it, since no Voronoi vertex would be recognized, preventing then an edge to be present in the final ECM. On that tile, the gray points are examples of "border points" that we use as starting points to trace additional edges.
(b) Both points in gray will be considered border points since they line on a border of the tile and feature 2 unique directions. From either of them, a "parallel" edge will be traced, ending at the position of the other point, which will be then not processed any further.
(c) From the border point on the right both a "parallel" edge and a "perpendicular" edge will be spawned. Then, testing the color data associated with the end-point in common will make clear that those 2 edges are in fact parts of the same edge. Hence, they will be merged and added to our list of edges.

x coordinate (for the lists "topBorderEdges" or "bottomBorderEdges" which are associated to horizontal borders) of the point of the indexed edge touching the border: in case of both endpoints touching it, we will index the edge using the lowest value among the possible ones. We recall that no edge can have a $length > 1$ and the starting point that coincides with the endpoint. However, at each position of any list we could have more than one edge indexed because different edges can start/end with the same point touching a border. Because of the properties of Voronoi diagrams, it can be observed that only up to 2 different edges can be indexed by a single value. It can be noticed also that a same edge can touch 2 borders at the same time: in case of an edge touching both an horizontal and a vertical border, we will store that edge twice in the 2 different lists, but in case of an edge touching either 2 vertical borders or 2 horizontal borders, we will choose one of the 2 borders and store the edge only in the list associated with that border. The final graph will be correct independently from this choice. In the end, lists associated with vertical borders could store up to $tileHeight$ indexed pairs of edges, while those associated with horizontal borders could store up to $tileWidth$ indexed pairs of edges. Lists will be kept sorted by their index values. In our C++ implementation, the 4 lists are implemented using STL multi-map containers storing pointers to edge structures.
Populating the 4 new lists for each tile is a matter of iterating over all the edges traced and checking whether the first and/or last points of each edge touches one or more of the borders, eventually inserting the edge in the corresponding lists: edges that do not touch any border will be inserted in another list $internalEdges$. At the end, we will also sort all the edges in any border list. The computational complexity of the entire process is $O(n)$ for the first insertion of the n edges in the right list, and then $O(nlog(n))$ for the sorting, thus $O(nlog(n))$ is a final complexity measure.

5.2.4 About the computation of Closest Points

We have already defined the idea of "unique" directions, and the fact that it implies checking, for a point, the Closest Points on the obstacles that "generated" the point according to the definition of the Voronoi diagram. Unique directions are checked multiple times even during tracing, as it should be clear from Algorithms 3 and the descriptions of the tracing

operations above. In terms of arithmetic operations, checking the unique directions, and thus computing the Closest Points, can be one of the most expensive operations to perform; at the same time, the Closest Points information do not change during a single construction since the footprint is always the same. While the original ECM framework recomputes the Closest Points information and associated unique directions every time they are "queried", a new class called ClosestPointsCache has been created in the new implementation, to represent the colors and Closest Points information of points lying on a tile: when such information is requested for a point, it is computed on-the-fly **only** if the cache does not contain it already, and is then eventually stored. Information is "cached" using an hash-map container indexed by the coordinates of the point: ideally, we hope the query cost of the hash maps to be less than the cost of computing the Closest Points, and this should eventually lead to an overall speed-up since the same information is often needed more than once.

5.3 Merging tiles

Once we have stored correctly in our border lists the edges touching the borders of each tile, we can proceed with merging the content of all the tiles together to have our final Medial Axis. For each pair of tiles having a border in common, we have to process the edges indexed in the 2 lists correspondxent to the 2 borders. Now, we know that

- edges at position i of a single list associated with a border will share an endpoint (and thus its color data);
- both arrays at index i will store a list of edges that will all have at least an endpoint with either the x or y coordinate equals to i.

Algorithm 5 shows the "skeleton" of a function that merges 2 tiles horizontally, sharing a vertical border. Logically, we notice that the two lists taken into account have only at the same indexes the edges that will eventually need to be merged together. We can also notice that two edges that should be merged have no points in common, thus merging their points will result in the right sequence of distinct edge-points. However, the "contact points" of the two edges that will be merged will have some color data in common, but "swapped": we have already called this condition "compatible color data", and we will use it to make sure we are performing a correct merge. If we consider the content of two lists involved in a merge, at index i, being not empty at both sides, only 5 different scenarios can appear, which we will now cover more in depth. Although the following description univocally refers to the "horizontal merge" case, once this has been understood, it can be easily adapted to the "vertical merge".

- **Scenario A (Figure 24a):** This is the case when only 2 edges (one per side) are stored at the same index (and thus having the same y coordinate) and have both only **2** unique pairs of colors among the (common) color data. This scenario shows that, in the new merged tile, the 2 edges should become just one. The new edge will be created by taking the vertex of $edgeTouchingFromLeft$ that does not touch the border, adding all the points (together with the color data) of $edgeTouchingFromLeft$ until the other vertex, then adding all the points, from the vertex touching the border to the non-touching one, of $edgeTouchingFromRight$. This new resulting edge will have a length:

$$\begin{aligned} newEdge.length == & \, edgeTouchingFromLeft.length \\ & + edgeTouchingFromRight.length \end{aligned}$$

- **Scenario A-2 (Figure 24b):** This case is similar to the previous one because we need to merge 2 edges into one, but to figure it out we had to compute first the "unique" directions. Ideally, we would not merge the 2 edges if either of them ended up with a point having 3 or more different pairs of colors, since that would show that one of the edges ends with a vertex of the Medial Axis, but noticing that the "unique" directions are 2 for both endpoints leads to the conclusion that the point is a vertex of the Voronoi

diagram but not of the Medial Axis, therefore we merge as we did in the previous case. The only difference is that now we could have more than one edge indexed per side: since we computed the *uniqueDirections*, we will use this information to discriminate which edges to merge. A pair of edges, then, will become a *newEdge* as described above, while the rest will remain unchanged.

- **Scenario B (Figure 24c):** This is the case when we have one vertex of the Medial Axis as endpoint, touching the shared border and belonging to the $edgesTouchingFromLeft$, while instead the endpoint touching the shared border and belonging to the *edgesTouchingFromRight* is not. Notice that in this case we can assume that the list $edgesTouchingFromRight$ contains only one edge, given that the point vertex touching the shared border exhibits only 2 "unique" directions. Therefore, the edges of $edgesTouchingFromLeft$ will be kept as they are, while instead the edge $edgeTouchingFromRight$ will be extended with the additional point $vertexTouchingFromLeft$ after the current endpoint (becoming a new endpoint itself), together with its color data. In the end we will have a new edge $newEdgeFromRight$ with length:

$$newEdgeFromRight.length == edgeTouchingFromRight.length + 1$$

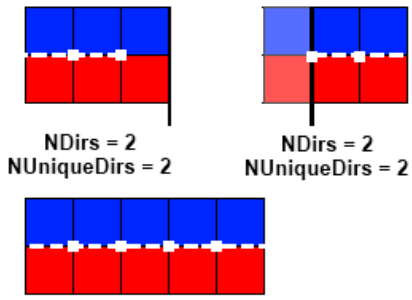
- **Scenario B-2:** This scenario is similar to **Scenario B**, only with "inverted roles". Hence, it will be the edge in $edgesTouchingFromLeft$ that will be extended with the additional point $vertexTouchingFromRight$ as a new endpoint after the current endpoint that touches the border. We will then have a new edge $newEdgeFromLeft$ with length:

$$newEdgeFromLeft.length == edgeTouchingFromLeft.length + 1$$

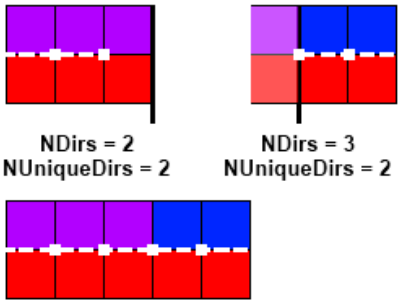
- **Scenario C (Figure 24d):** This is the last case: both lists of edges share endpoints touching the border that feature 3 or more *uniqueDirections*. We will not perform any merge or extension in this case, all the edges will be kept as they are. However, color data of the 2 end points shared by the 2 lists of edges show that this 2 points should be merged into a new edge *newEdge* since they are mapped to different points of the grid of the new tile, are both vertices, and have also compatible color data. Thus, we will create and add the new edge *newEdge* connecting these points, with length:

$$newEdge.length == 2$$

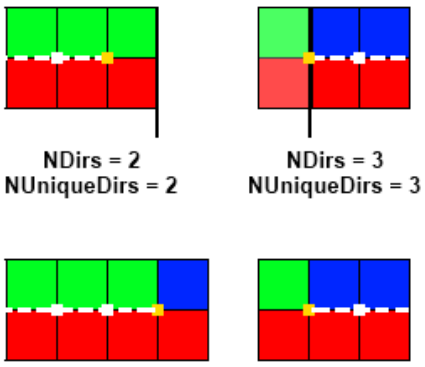
In all of the above scenarios, additional actions are applied to both newly created and untouched edges. First, the values of the coordinates of edge-points are updated since each tile stores points using a "local" reference system. Then, we check the end points of the edges to see if they touch the border shared by the two tiles: in case of an old, unchanged edge, we will check only the endpoint that did not touch the border already, while in case of a newly created one we will check both of them. If an edge has 1 endpoint touching again the border **and** if that point has a value of the coordinate associated with the merge operation (y in case of horizontal merge, x in case of vertical merge) that is greater that the index value currently being processed, then we re-insert the edge in one of the two arrays at the new position. Otherwise, we just put it in the list *newTileEdges*. In our implementation, edges have a flag that specifies if they are updated or not, which is used during this merge step to avoid mis-interpreting coordinate values.
Similar operations have to be performed when, while iterating, the current lists of elements being inspected from both arrays are not indexed by the same value. In this case, we will work only on the list of edges indexed by the lowest value: we will first update the coordinates of those edges, then for each of them we will check if the other endpoint touches again the same border: if it does, we will re-insert it again in the same array, otherwise we will add it to *newTileEdges*. In the end, we will also update the list of edges to process at the next iteration for that side of the border by calling GetNext(). Algorithm 5 shows it.



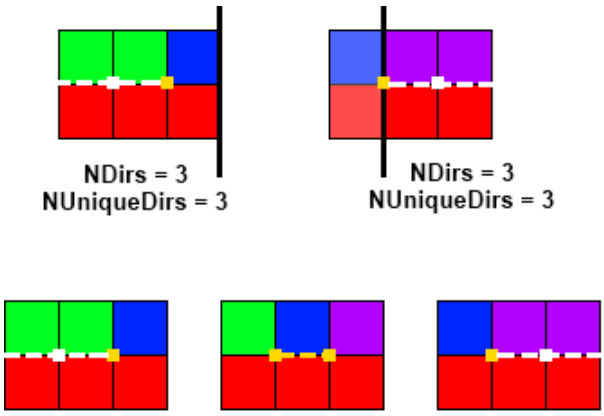
(a) **Connection scenario A**: the endpoint of the edge touching the common border from the left has $NDirections = 2$ and $NUniqueDirections = 2$. The same applies to the endpoint of the edge touching the border from the right; endpoints have compatible color data. Therefore, the connection results in a fused edge out of the two.



(b) **Connection scenario A-2**: the endpoint of the edge touching the common border from the left has $NDirections = 2$ and $NUniqueDirections = 2$. The endpoint of the edge touching the border from the right instead has $NDirections = 3$ but $NUniqueDirections = 2$ (the pair blue/purple does not represent a unique direction). Endpoints have compatible color data, hence again, the connection results in a fused edge out of the two.



(c) **Connection scenario B**: the endpoint of the edge touching the common border from the left has $NDirections = 2$ and $NUniqueDirections = 2$. The endpoint of the edge touching the border from the right instead has $NDirections = 3$ and $NUniqueDirections = 3$. Hence, the endpoint of the second edge is a vertex, and is exactly the vertex that will end the first edge. We will then append the vertex to the end of the edge on the left, while keeping the edge on the right. The result is 2 edges, one old and one new.



(d) **Connection scenario C**: the endpoint of the edge touching the common border from the left has $NDirections = 3$ and $NUniqueDirections = 3$. The endpoint of the edge touching the border from the right instead has $NDirections = 3$ and $NUniqueDirections = 3$. Hence, the endpoints of both edges are vertices, and we should not modify those edges. Moreover, since the two endpoints are mapped to different points of the grid and with compatible color data, we will create an additional edge out of the 2 endpoints. The result will be 2 old edges a newly created one.

Figure 24: Connection scenarios - horizontal merge. During a vertical merge the same scenarios show up, only with color data at different positions.

A merge operation only deals with the edges of both tiles that touch the shared border. Therefore, to have a full description of the edges of the new tile, we have to add the edges from the previous tiles that touched no border/not the border involved in the merge to the new edges $newTileEdges$. Moreover, the coordinate values of these edges have to be

Algorithm 5 MergePartialEdges($rightBorderEdges$, $leftBorderEdges$).
Input: Border array with edges touching the right border of a tile t $rightBorderEdges$ and border array with edges touching the left border of tile $t-1$ $leftBorderEdges$. This border is shared between the 2 tiles.
Output: New edges after merge operation.

```
1: newTileEdges ← []
2: currentEdgesFromLeft ← rightBorderEdges.GetNext()
3: currentEdgesFromRight ← leftBorderEdges.GetNext()
4: while !rightBorderEdges.IsFinished() ∧ !leftBorderEdges.IsFinished() do
5:   vertexTouchingFromLeft ← GetVertexTouchingRightBorder(rightBorderEdges)
6:   vertexTouchingFromRight ← GetVertexTouchingLeftBorder(leftBorderEdges)
7:   nDirsLeft ← GetDirections(vertexTouchingFromLeft)
8:   nDirsRight ← GetDirections(vertexTouchingFromRight)
9:   if vertexTouchingFromLeft.y == vertexTouchingFromRight.y then
10:    if nDirsLeft == 2 ∧ nDirsRight == 2 then
11:      Assert(currentEdgesFromLeft.size ==1 ∧ currentEdgesFromRight.size ==1)
12:      Assert(Compatible Color Data)
13:      Scenario A
14:    else
15:      nUniqueDirsLeft ← GetUniqueDirections(vertexTouchingFromLeft)
16:      nUniqueDirsRight ← GetUniqueDirections(vertexTouchingFromRight)
17:      if nUniqueDirsLeft == 2 ∧ nUniqueDirsRight == 2 then
18:        Assert(Compatible Color Data)
19:        Scenario A - 2
20:      else if nUniqueDirsLeft > 2 ∧ nUniqueDirsRight == 2 then
21:        Assert(Compatible Color Data)
22:        Scenario B
23:      else if nUniqueDirsLeft == 2 ∧ nUniqueDirsRight > 2 then
24:        Assert(Compatible Color Data)
25:        Scenario B - 2
26:      else if nUniqueDirsLeft > 2 ∧ nUniqueDirsRight > 2 then
27:        Assert(Compatible Color Data)
28:        Scenario C
29:      end if
30:      currentEdgesFromLeft ← rightBorderEdges.GetNext()
31:      currentEdgesFromRight ← leftBorderEdges.GetNext()
32:    end if
33:  else if vertexTouchingFromLeft.y > vertexTouchingFromRight.y then
34:    UpdateEdges(currentEdgesFromRight)
35:    CheckEndpointsAndReinsert(currentEdgesFromRight, leftBorderEdges, newTileEdges)
36:    currentEdgesFromRight ← leftBorderEdges.GetNext()
37:  else if vertexTouchingFromLeft.y < vertexTouchingFromRight.y then
38:    UpdateEdges(currentEdgesFromRight)
39:    CheckEndpointsAndReinsert(vertexTouchingFromLeft, rightBorderEdges, newTileEdges)
40:    currentEdgesFromLeft ← rightBorderEdges.GetNext()
41:  end if
42: end while
43: return newTileEdges
```

updated as well. All the border shared by two tiles lead to a merge operation, thus, up to 4 merge operations can occur per tile.

5.4 From the final Medial Axis to an ECM

Once merge operations are finished, we will end up with a big tile representing the whole grid and thus the final Medial Axis. Edges will still be organized between the 5 ordered lists that distinguish between edges touching at least one border and and edges not touching any of them. However, the ECM structure is organized as an array of vertices of the Medial Axis, with each vertex indexing the edges that it originates. Therefore, we will reorganize the content of the final tile in the same way: mainly, we need to iterate over all the edges in all the 5 lists, store their starting points and endpoints as vertices, and then add them to the final list $ECMGraph$ at the position of the spawning vertex. While iterating, we can encounter an edge e that in fact originates from a vertex v that has already been found and added to the new list. Therefore, we should add e to $ECMGraph$ at the index of v: to recognize such vertices we will make sure to store in an additional structure every vertex we find, together with the index where they are located in $ECMGraph$, and we will use this structure for the various look-ups. Since in the implementation we are using hash-tables that can be queried in amortized constant time, we can assume that the final complexity of the reorganization of the edges is $O(n)$ where n is the total number of edges stored in all the 5 lists resulted from the merge operations.

At this point, $ECMGraph$ does not describe an ECM yet, but a few more steps must be performed. As [Ger10] illustrates, an ECM is a graph made of vertices, edges connecting the vertices, and event points lying on the edges; as a Navigation Mesh, it is considered an optimal description of the walkable space of an environment. At this stage $ECMGraph$ already includes all the necessary vertices and edges, but it could miss some event points that lie on our traced edges. Unfortunately, the only way to locate those event points is to iterate over all the edge-points of all the edges while checking if 2 consequent edge-points feature a change into the associated colors, or a change in the index of either of the normals associated in case they are associated to the same obstacles. The final ECM will include the vertices and edges connecting those vertices described as lists of 2 or more event points (an edge is described, at least, by its starting and ending vertices).

An ECM also includes the Closest Points' information, associated to each vertex or event point. In fact, Closest Points information have been already used thorough the tracing of MA edges, therefore they have been already computed and stored in the various instances of the the class ClosestPointsCache associated to the tiles. Two instances of ClosestPointsCache can be easily merged like tiles are, since they mostly require the "fusion" of the hash maps storing the information of the points, and fusing such hash-maps is trivial because they do not contain information that refer to the same points. At this final stage, we should have all the information stored in just one instance of ClosestPointsCache, having in fact all the closest points information that we need already computed, and what is left to do is simply to query this instance for the Closest Points that have to be associated with the corresponding points of our final ECM.

The very final step is to transform the coordinates of vertices and event points from the reference system of the frame buffer grid back to the reference system of the original footprint. The frame buffer grid enforces integer coordinates while of course the original coordinates of the footprint are decimal: we have already discussed a transformation that allows mapping points of the original footprint to frame buffer coordinates (SECTION) and a similar one that allows to map points back. At this stage, we will just apply the latter to vertices, event points and Closest Points in order to have a final ECM that refers to the initial coordinates system.

5.5 Final Considerations concerning a multi-tiled construction

Indeed, subdividing the computation into tiles with partial data that refer to a global Medial Axis and subsequently merge them proved to be quite tricky: although the subdivision we

defined is quite advantageous since it leads to no missing color information for edges of a tile, it still implies dealing with overlapped OpenGL reads of the same frame buffer and with points of different tiles that are in fact mapped to the same coordinates. Additionally, it requires careful treatment of points lying close to borders so that "partial edges" are correctly traced and can be then merged into the final edges of the Medial Axis.
The merging of the edges themselves is also not trivial: however, in this case, we just had to enumerate all the possible configurations that could have been found at the borders of the tiles, and then derive what is the correct way to handle each case. Unless we missed something, all the cases should have been correctly covered. This being said, in the implementation of the routines care must be taken since they involve many different types of containers (and their related operations), and this can easily lead to wrong or poorly performant code.

5.6 Towards a CPU-Parallel implementation

From the previous sections it can be noticed how tracing any of the tiles is a quite self-constrained operation, which could then fit a multi-tiled parallel construction approach. In this section we will outline and discuss such a multi-tiled construction taking advantage of many CPU threads. Algorithm 6 is a first draft that describes a new parallel implementation. It includes a number of assumptions and simplifications:

- the number of tiles per row $NTilesHor$ and tiles per column $NTilesVert$ of the hypothetical grid we subdivided our initial framebuffer data in will be the same. Thus, $NTilesVert == NTilesHor$ and $NTiles = NTilesVert * NTilesHor$ will hold.
- the number of CPU buffers we will employ $NBuffers$ will be equal to the number of tiles in the grid $NTiles$
- the number of threads executing in parallel $NThreads$ will be equal to the number of CPU buffers $NBuffers$

Algorithm 6 ComputeECMMultiTiledParallel($NTilesHor$, $NTilesVert$, $tileWidth$, $tileHeight$).
Input: Number of tiles per row $NTilesHor$ and tiles per column $NTilesVert$ in our grid of tiles, width $tileWidth$ and height $tileHeight$ of every tile.
Output: An ECM mesh.

1: $NTiles \leftarrow NTilesHor$ * $NTilesVert$
2: $NBuffers \leftarrow NTiles$
3: $NThreads \leftarrow NTiles$
4: $CPUBuffers \leftarrow$ AllocateHeap($tileWidth$ * $tileHeight$, $NBuffers$)
5: **for** $t = 0 \rightarrow NTiles - 1$ **do** {For loop executed **in parallel** by $NThreads$ threads}
6: $tx \leftarrow t$ Mod $NTilesHor$
7: $ty \leftarrow t$ Div $NTilesHor$
8: DownloadGPUData($tx * tileWidth$, $ty * tileHeight$, $tileWidth$, $tileHeight$, $CPUBuffers[t]$)
9: $GVDData[t] \leftarrow$ TraceGVDTile($CPUBuffers[t]$)
10: ArrangeEdgesTouchingBorders($GVDData[t]$)
11: **end for**
12: **sync_threads**
13: PerformParallelMerge($NTilesHor$, $NTilesVer$, $GVDData$, $NThreads$)
14: $ECM \leftarrow$ FinalizeECMGraph($GVDData$[0])
15: **return** ECM

The instruction *sync_threads* (used in Algorithm 6 for the first time) ensures that every thread will wait for all the other active threads to reach that point in the code before proceeding with the next instruction. From Algorithm 6 we can observe that:

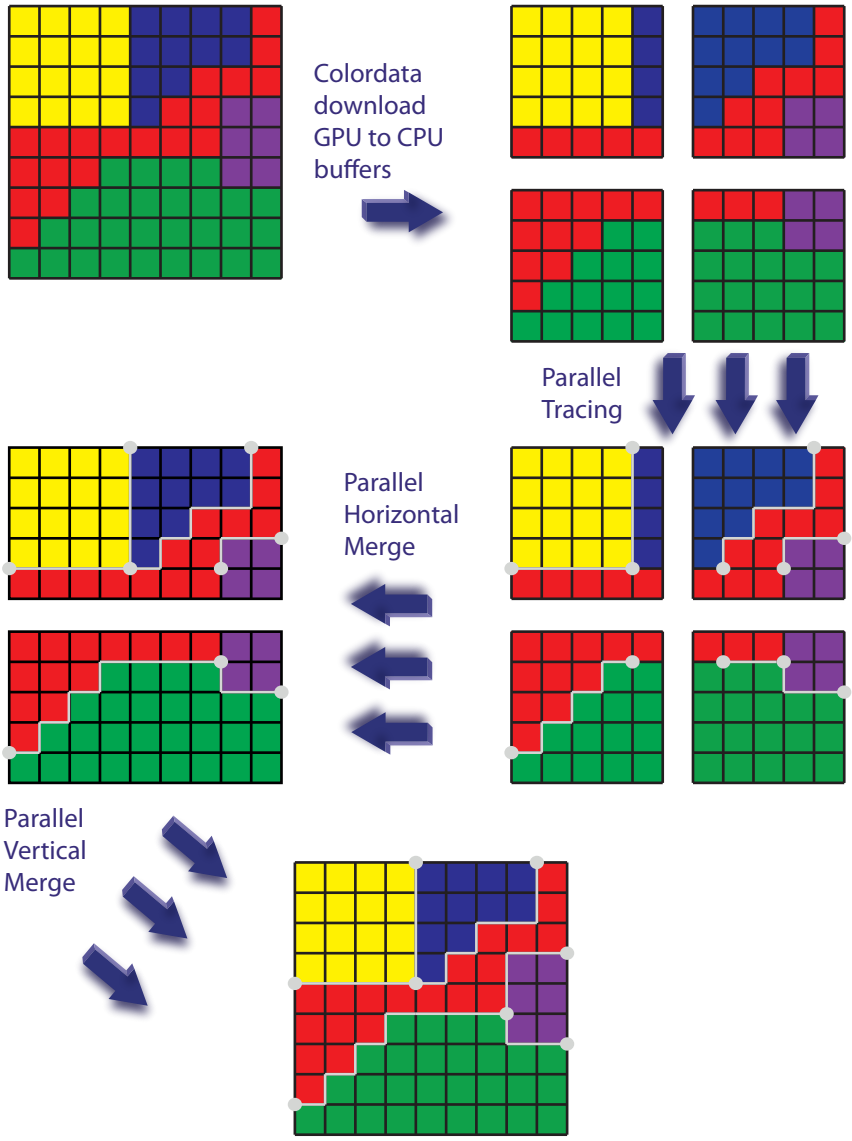


Figure 25: Overview of the parallel approach. Steps which are performed in parallel are outlined with multiple arrows.

- Both the tracing (function TraceGVDTile) and re-arrangement of the edges (function ArrangeEdgesTouchingBorders) of each tile can be executed in parallel by our *NThreads* threads. Moreover, the tracing of an tile also involves computing (and then storing) the closest points information and the index of the normals associated with every edge-point/vertex, hence, we are in fact distributing all the geometrical computations among the various threads.

- Each tile t gets assigned a color data buffer and an instance of the Closest Points' cache, which are used during construction. Since every different thread deals with the processing of a different tile, no read-write conflicts are generated by different threads accessing the same shared data.

- We need to enforce the synchronization of all threads at the end of the main loop since the function PerformParallelMerge, that will merge the partial edges of all the tiles, assumes that all tiles have been already filled. We will cover this function separately later.

However, the implementation described by Algorithm 6 has a number of issues. First, the function DownloadGPUData is called by different threads, and in parallel, but this function performs a GPU memory transfer operation from an OpenGL context: accessing the same OpenGL context in parallel from many threads is quite troublesome, especially if we need to perform memory transfers, and generally not recommended. Therefore, our implementation will perform all the required GPU transfers sequentially as a first thing.
An additional shortcoming of Algorithm 6 is that we simplified it by assuming

$$NTiles == NBuffers == NThreads$$

but this is in general not desirable. In fact, we would like to put a limit on the total CPU memory that our approach will employ during the whole process, which also means putting a limit on the number of CPU buffers holding the color data that we will use, given that they often represent a big majority of the CPU memory used. Nonetheless, we don't want the (limited) number of CPU buffers to constrain the total number of tiles of our construction, because this would in turn limit the total resolution of the construction (the dimension of a tile is constrained by the limit on the size of a GPU frame buffer). At the same time, we don't want the number of CPU buffers to drive the amount of parallel threads we can employ, too. Hence, the construction will proceed by first determining the number of tiles $NTiles$ to use and their dimensions, and then:

- If possible, we will try to instantiate a buffer per tile **and** a thread per tile, if this is allowed by the system and is in accordance with what was specified by the user. In case that was possible, we will have $NTiles == NBuffers == NThreads$, which is in fact the easiest situation, since then we can just loop (in parallel) over all the tiles following Algorithm 6.

- If we end up with $NTiles \neq NBuffers$, and in general $NBuffers < NTiles$, we will try to instantiate a thread per CPU buffer, having thus $NThreads == NBuffers$. In case that was possible, we will proceed with processing the $NTiles$ tiles in multiple "passes", processing up to $NBuffers$ tiles at each iteration in parallel. The $NBuffers$ CPU buffers will be re-used during the various passes by the concurrent threads.

- If $NBuffers < NTiles$ **and** we could not have a thread per CPU buffer, thus $NThreads < NBuffers$, we will further decrease the number of CPU buffers to instantiate so that it matches the number of threads, thus $NBuffers \leftarrow NThreads$, and the rest will proceed as described in the previous point. We will do so because it is pointless to instantiate more CPU buffers than the number of parallel threads that will then work during each of the passes.

The implementation allows the user to specify a value for $NTiles$, $NBuffers$ and $NThreads$, either explicitly or implicitly, and then the policies illustrated above are applied. In addition, we will limit the number of threads $NThreads$ to 1 per processor, or 2 per processor supporting either Intel Hyper-threading or the AMD "Bulldozer" micro-architecture (or a better one): although OSes allow programs to spawn almost as many threads as they wish, too many threads end up overloading the system itself, leading to oversubscription and in general to a saturation of the system and its computational capacities.

In Algorithm 7 we can observe how the issues discussed above led to a modified parallel implementation. We made again some simplifying assumptions, basically that all tiles have the same width and height and that the number of $NBuffers$ picked evenly divides the total number of tiles $NTiles$. These assumptions lead to a cleaner implementation, although extending the whole approach so that they are not required would be a minor adjustment. Each pass processes $NBuffers$ tiles in parallel by $NThreadsToUse == NBuffers$ that are then synchronized at the end of the pass: this is required to perform the data transfer from the GPU for the execution of the next pass (if any), since GPU transfers from a unique OpenGL context to the CPU can not be performed in parallel. This being said, OpenMP abstracts the creation and destruction of threads themselves, which depends on a variety of

Algorithm 7 ComputeECMMultiTiledParallelImproved($NTilesHor$, $NTilesVert$, $tileWidth$, $tileHeight$, $NBuffers$).
Input: Number of tiles per row $NTilesHor$ and tiles per column $NTilesVert$ in our grid of tiles, width $tileWidth$ and height $tileHeight$ of every tile, and maximum number of CPU buffers to use at the same time to hold the color data $NBuffers$.
Output: An ECM mesh.

```
1: NTiles ← NTilesHor * NTilesVert
2: NThreadsToUse ← Min(NThreads, MaxThreadsAvailable)
3: NBuffers ← NThreadsToUse
4: Assert(NTiles ≥ NBuffers)
5: CPUBuffers ← AllocateHeap(tileWidth * tileHeight, NBuffers)
6: GVDData ← AllocateHeap(NTiles)
7: NPasses ← NTiles/NBuffers
8: for p = 0 → NPasses − 1 do
9:    for b = 0 → NBuffers − 1 do
10:      t ← (p ∗ NBuffers) + b
11:      tx ← t Mod NTilesHor
12:      ty ← t Div NTilesHor
13:      DownloadGPUData(tx ∗ tileWidth, ty ∗ tileHeight, tileWidth, tileHeight,
         CPUBuffers[b])
14:   end for
      {For loop executed in parallel by NThreadsToUse threads}
15:   for b = 0 → NBuffers − 1 do
16:      t ← (p ∗ NBuffers) + b
17:      GVDData[t] ← TraceGVDTile(CPUBuffers[b])
18:      ArrangeEdgesTouchingBorders(GVDData[t])
19:   end for
20:   sync_threads
21: end for
22: PerformParallelMerge(NTilesHor, NTilesVer, GVDData, NThreads)
23: ECM ← FinalizeECMGraph(GVDData[0])
24: return ECM
```

different factors, like compiler's flags/hints, the OpenMP version of the machine where the code is being run on and its associated behavior, and the policies of the operating system. Specifically, in our implementation, OpenMP will try to "pool" the threads employed by the construction instead of continuously create/destroy them at the beginning/at the end of the second for loop, avoiding the related unnecessary performance overhead.

5.6.1 Merging Tiles in Parallel

Parallelism can be exploited also during the merge step: Figure 26 shows the (simplified) scenario where we have a big environment subdivided into $NTiles$, all of them having the same $tileWidth$ and $tileHeight$, with $tileWidth$ and $tileHeight$ being equal. Moreover, here $NTilesHor = NTilesVert$, and they are a power of 2 (8).
From Picture 26 we can observe that we can merge horizontally all the tiles sharing a vertical border at the same time. In fact, those merge operations do not depend on each others. They could be executed in any order and the result will still be the same; hence, we can execute them in parallel. After the first "cycle" of horizontal merges we can execute a second cycle of horizontal merges, again in parallel, until no more horizontal merge operations are possible. At that point, we could proceed with merging the resulting tiles vertically, again until no more vertical merges are possible.

We can call our cycles of merge operations "merge steps". A merge step will perform a number of parallel merges, either horizontally or vertically, at the same time. At merge

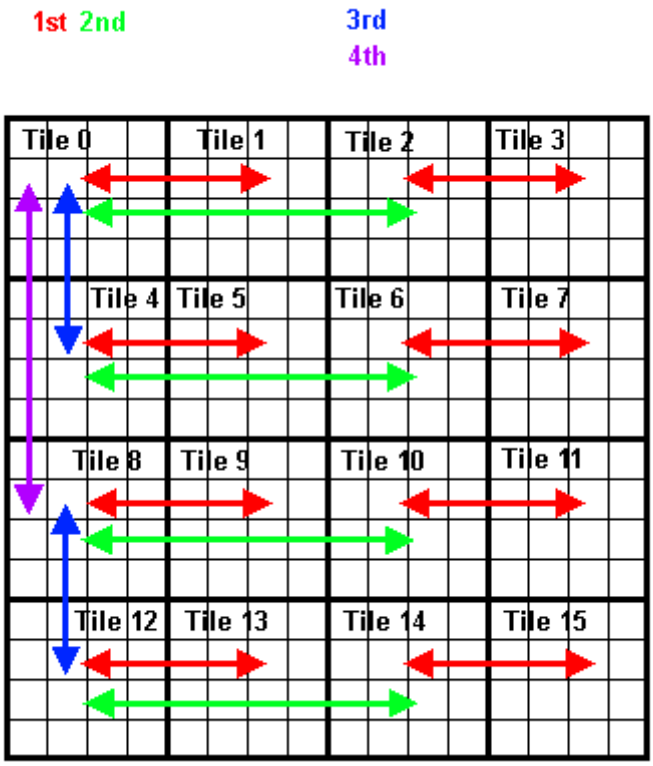


Figure 26: Parallel merge of 16 tiles. Arrows connect tiles that are merged together. Arrows of the same color represent "merge steps": lists of operations that can be performed in parallel by different threads. The order in which the merge steps are performed is outlined above the diagram. All the necessary horizontal merge operations can be performed in 2 "steps" if we have $NThreads/2 = 8$ at our disposal, ending up with $NTilesVert = 4$ tiles. Again, all the vertical merge operations can be performed in 2 parallel "steps". In the end, we will perform $O(log(NTiles))$ merge steps.

step $s = 0$ we will always perform horizontal merges: the number of operations will be $NTiles/2$, therefore we would achieve maximum performance by employing $NTiles/2$ parallel threads. At merge step $s = 1$ we will have $NTiles/2$ tiles in total, and therefore we could achieve maximum performance by employing $NTiles/4$ threads simultaneously. Assuming our $NThreads = NTiles/2$, we can see that the number of horizontal merge steps $NHorizontalSteps$ is

$$NHorizontalSteps = log_2(NTiles) = log_2(NThreads * 2)$$

and at each step s_j with $j = 0 \dots NHorizontalSteps - 1$ we will perform a total number of merge operations $NHorMerges_j$

$$NHorMerges_j = NTiles/(2^{j+1}).$$

At most, we will perform $NHorMerges_j = NTiles/(2^1) = NTiles/2$ operations when $j = 0$, at the first step. Therefore, we should achieve maximum performance in all the steps when

$$NThreads = NBuffers \geq NTiles/2. \quad (1)$$

Eventually, after $NHorizontalSteps$ horizontal steps, we will exhaust all the necessary horizontal merge operations. This means we will end up with $NTilesVert$ tiles with a new $tileWidth = totalWidth$ and $tileHeight$ unchanged.
We will now perform some vertical merges: again, we can derive our $NVerticalSteps$ as

$$NVerticalSteps = log_2(NTilesVert)$$

Algorithm 8 MERGETILESPARALLEL(*tilesGrid*, *NTiles*, *NTilesHor*, *NTilesVert*, *NThreads*).
Input: Merging the GVD data of a a grid *TilesGrid* of *NTiles* organized by *NTilesHor* per row and *NTilesVert* per column in parallel using *NThreads*. In the end, *tilesGrid*[0] will hold the whole new data.
Output: void.

```
1: step = 1
2: while step < NTilesHor do
3:    for i = 0 → (NTiles / 2 * step)-1 do {Loop is executed in parallel by NThreads}
4:       indexSRC ← 2 * step * i
5:       indexDST ← (2 * step * i) + step
6:       MERGERIGHT(tilesGrid[indexSRC], tilesGrid[indexDST])
7:    end for
      single_thread_start
8:    step = step * 2
      single_thread_end
9: end while
   sync_threads
   single_thread_start
10: step = 1
   single_thread_end
11: while step < NTilesVert do
12:    for i = 0 → (NTilesVert / 2 * step)-1 do {Loop is executed in parallel by
       NThreads}
13:       indexSRC ← 2 * step * i * NTilesHor
14:       indexDST ← (2 * step * i) + step) * NTilesHor
15:       MERGEBOTTOM(indexSRC, indexDST)
16:    end for
17: end while
```

and at each step $i = 0 \dots NVerticalSteps$ we will perform $NVertMerges_i$ operation, with

$$NVertMerges_i = NTilesVert/(2^{i+1}).$$

Thus, this time, we need $NThreads = NBuffers \geq NTilesVert/2$ to have maximum performance, a condition which is always fulfilled if we fulfill the previously defined property 1. Algorithm 8 shows the whole approach.
What we notice is that both sequences of horizontal and vertical merges are an application of the parallel reduce idiom, a well-known parallel idiom introduced by [Ble90], with the only difference that merge operations are performed instead of simple sums. In case $NTilesHor == NTilesVert$ is a power of 2, we can observe that the number of vertical merge operations that have to be performed during the first vertical merge step is simply $NTilesVert/2 = NTilesHor/2$, therefore we can organize the vertical merge steps following the horizontal ones in a binary tree, in the same way [Ble90] does. As the paper outlines, if we have to reduce n elements using $p = n/2$ processors working at the same time, the complexity of the reduction is

$$O(log_2(n))$$

which is

$$O(log_2(NTiles)$$

in our case.
The cases in which $NTilesHor \neq NTileVert$, or either of them is not a power of 2, are anyway correctly handled in our implementation, although those scenarios require a more careful treatment: for example, in case $NTilesHor\%2 \neq 0$, a thread that wants to perform an horizontal merge between tile t and tile $t+1$ has to check in advance whether tiles $t+1$ shares a vertical border with tile t. Often, this leads to less merge operations in total, but in

fact they not always lead to better performance since threads can end up being under-used, for example. In addition, the theoretical complexity of the whole merge changes depending on the number of threads $NThreads$ available, decreasing, but as [Ble90], the speedup granted by the parallelization is always optimal.

6 Experiments

The previous chapters described the ideas behind 2 new implementations that aim at constructing high-resolution ECMs. In this chapter, we will test them together with the original one. We will mainly compare the difference in running times between the 3, and we will show that the new implementations allow to build ECMs of environment at resolutions that are higher that the maximum one reachable by the standard implementation, illustrating also some of the additional benefits.

6.1 Experimental Setup

We will test 3 different implementations:

- **GPGPU** is a multi-tiled implementation that uses CUDA to perform some additional computations on the GPU, as described in Chapter 4. Given the input parameters total resolution r and tile resolution t, it produces ECMs at resolution $r \times r$ by sequentially processing a number of tiles of resolution $t \times t$ which are then merged together into the final ECM. Due to the nature of the GPGPU operations, which at some point require 3 buffers with a size of $t * t * 4$ bytes of pinned GPU memory, t can be at most 4096 on most GPUs. This imposes a constraint on the resolution of the tile, but not on the final resolution r of the ECM. Moreover, since it includes some CUDA instructions, this implementation requires an NVIDIA GPU and a fairly recent driver version.

- **CPU Parallel** is a multi-tiled implementation that employs OpenMP to spawn many CPU threads and parallelize the construction of an ECM, as described in Chapter 5. Given the input parameters total resolution r, tile resolution t and number of threads to use T, it produces ECMs at resolution $r \times r$ by processing a number of tiles of resolution $t \times t$ in parallel using up to T CPU threads, merging them together in parallel into the final ECM. The resolution of a single tile is subjected to the same constraint that applies to the standard ECM implementation (due to the maximum size of a buffer than can be allocated on the CPU, buffer required to host and process the frame buffer data from the GPU), therefore t can not be greater than 16000. If no value for T is specified, the implementation uses the value returned from the OpenMP function $omp\_get\_num\_threads()$.

- **Original** is a standard version of the ECM implementation. Given the input parameters total resolution r, it produces ECMs at resolution $r \times r$ using the approach described in [Ger10]. Due to reasons already outlined, r can not be greater than 16000. "Original" is a not totally updated version of the ECM framework, and it uses 32-bit color values for the GPU rasterization part, while more recent versions use 16-bit color values. Lowering color-depth lowers the number of maximum obstacles supported, but increases the total resolution that can be achieved and improves efficiency. However, since the GPGPU implementation uses 32-bit color values to perform its computations, we decided to test "against" a version of the ECM framework that uses the same color-depth. This being said, changing the GPGPU implementation so that it supports using 16-bit color values is indeed possible.

During our experiments, we will construct Navigation Meshes of a selection of environments:

- **Military** is the 2D footprint of the McKenna MOUT training site at Fort Benning. Therefore, this is a real-life scenario, although only moderately big and complex (Figure 27a). It is 200 by 200 meters big and it includes 23 primitives.

- **City** is the 2D representation of a part of a city neighborhood. It is a fairly big, real environment with some irregular and narrow passages that lead to complex configuration of the Medial Axis (Figure 27b). It is 500 by 500 meters big and it includes 548 primitives.

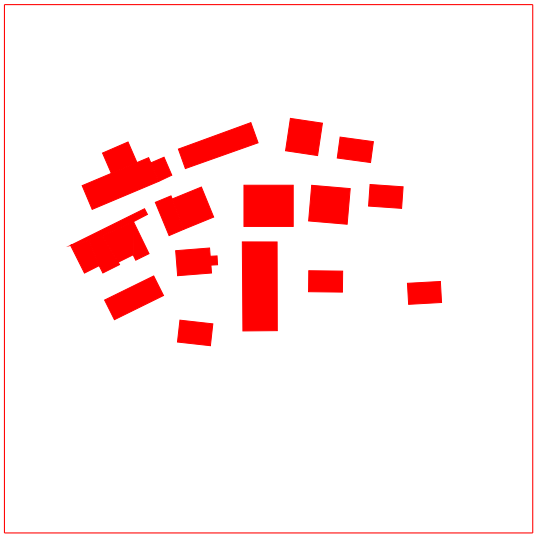
- **UUTest** is the 2D representation of a small city neighborhood, including streets and sidGewalks. It is a test environment meant to be used by the simulator of the company "Greendino". Being so big, it benefits from high resolutions (Figure 27c). It is 993 by 993 meters big and includes 1271 primitives.

- **aa64** is a fairly big environment featuring 4096 rectangles, all equally spaced, organized in 64 grids and columns. This is an example made for testing purposes only (Figure 27d). It is 160 by 160 meters big.

- **degpoints32** is a moderately big environment featuring 128 points, all equally spaced, lying on a circumference. This is an example made for testing purposes only, representing a known degenerate case for the computation of a Medial Axis: theoretically, the MA would feature 128 vertices, each one located equally distant from a pair of input points, all connected with edges to an additional vertex which would lie exactly on the center of circumference. However, approximated methods (like the one we are employing) are known to have problems locating the central vertex univocally, and thus with tracing all the other edges (Figure 27e). It is 334 by 334 meters big.

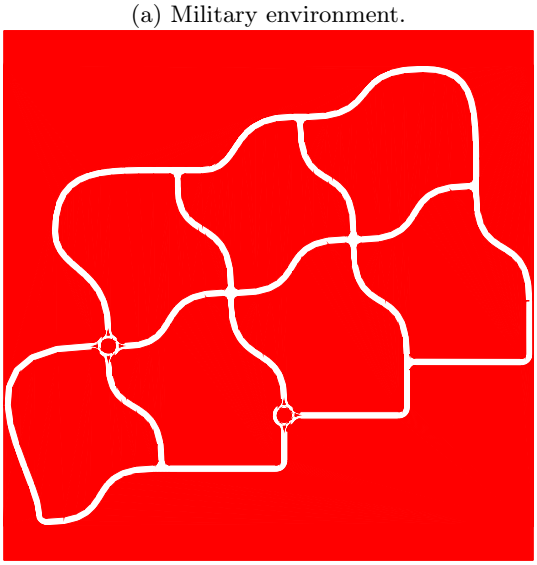
- **simplePrecision** is a big environment featuring only two seeds: a point, situated almost at the middle of the environment, and a vertical segment, situated on right of the point, very close to it. The Medial Axis of this environment mainly involves a single parabola. However, this environment can be useful to make some considerations about the precisions of the implementations (Figure 27f). It is 400 by 400 meters big and it includes only 2 primitives.

For all the tests a machine with the following specifications was used: Intel i7-3930K CPU clocked at 3.20 GHz, NVIDIA GeForce GTX 680 GPU, 16 Gigabytes of RAM and a 64-bit Windows 7 as operating system. The CPU employed features 6 cores and the Intel Hyper-threading technology, which allows each core to be treated as 2 logical computing units. Therefore, it is made to support the simultaneous execution of up to 12 threads with little effort. All the running times are averaged over a total of 40 runs. Throughout the entire chapter, we will use the phrases "significant" and "not significant" with their scientific statistical meaning. We used t-test to discriminate whether two independent samples are significantly different. Therefore, we use the term "significant" when the null-hypothesis was rejected with an $\alpha = 0.05_{2tail}$ at most.

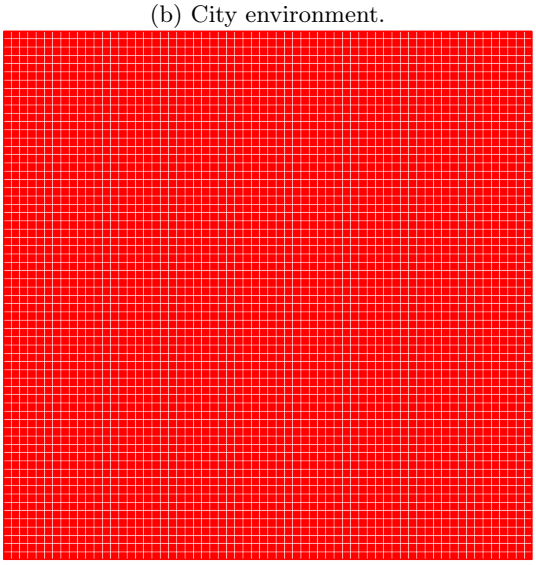


(a) Military environment.

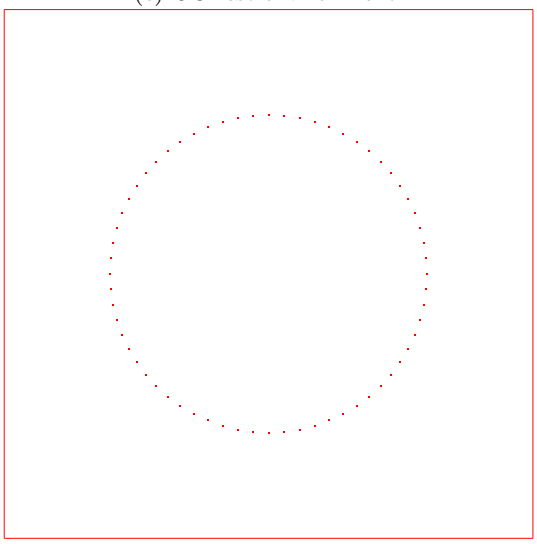
(b) City environment.



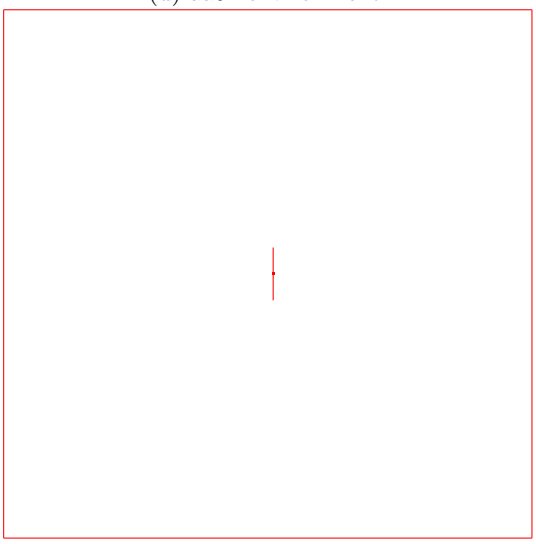
(c) UUTest environment.



(d) aa64 environment.



(e) degpoints32 environment.



(f) simplePrecision environment.

Figure 27: Environments tested.

6.2 Comparisons of Total Runtimes

Figures 28 and 29 show a plot of the construction times of ECMs of the different implementations at different total resolutions, grouped by environment. We do not list the used tile size for the GPGPU and CPU-Parallel implementation. Instead, given a total resolution, we show the run with the tile size that yields best results (we will discuss scenarios separately in the next section). We do not show a chart for the *simplePrecision* environment since it leads to a very simple and thus extremely quick to compute ECM. While picking the total resolutions, we picked multiples of 1024 since this lead to the most memory friendly and compatible scenarios for the GPGPU implementation.
What we can observe from the charts is that, in 4 environments out of 5, from a certain resolution onwards both "GPGPU" and "CPU Parallel" become significantly faster than "Original", with "GPGPU" being the fastest, leading to construction times that are, on average, 62% lower than the original ones for "GPGPU", and 28% lower than the original ones in case of "CPU-Parallel".
This being said, the environment aa64 shows a different picture: ECMs constructed using the "CPU Parallel" implementations give results that are not significantly better than the ones of "Original", while "GPGPU" achieves significantly better results only for some, high resolutions.
The previous results can be explained if we consider that what makes the "GPGPU" implementation more efficient is the GPGPU-aided filtering of vertex points and edge-points, the GPGPU computation of their associated directions, and in turn the transfer of only the data relevant to that points back to the CPU. For most environments, this leads to a noticeable speed-up, because keeping only the "useful" points is performed very quickly and subsequently a lot of "useless" data is not processed/transferred any further. This also explains the improved scalability with resolutions, since using higher resolutions increases the degree of the resulting graph by little (eventually nothing). However, in an environment with a high number and density of vertices and edge-points, like aa64, this speedup is almost nullified by the general costs of employing GPGPU. The same seems to apply to "CPU-Parallel": the big amount of workload that needs to be done anyway by all the threads, together with the cost of spawning those threads, nullifies the benefits of parallelization.
This being said, is important to notice that in the other environments, most of them representing real-life scenarios, the speed-ups are very good. Moreover, the new implementations allow us to build very high-resolution Navigation Meshes, and it's during those exact runs that we benefit more from the improved scalability.

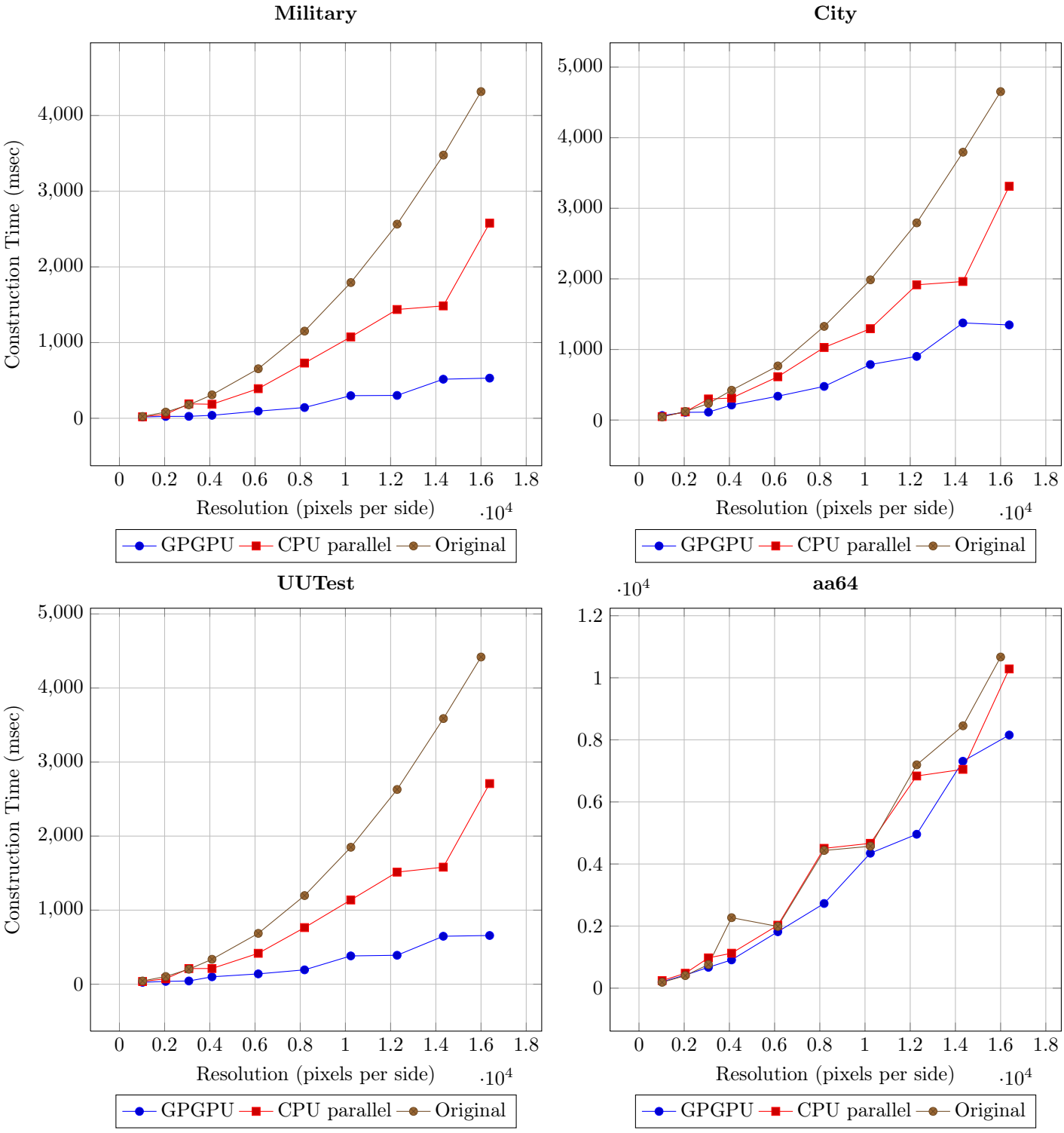


Figure 28: Comparisons between running times of the 3 implementations, environments Military, City, UUTest and aa64. In handling with the first 3 environments "GPGPU" and "CPU-Parallel" exhibit significantly lower computational times at most resolutions.

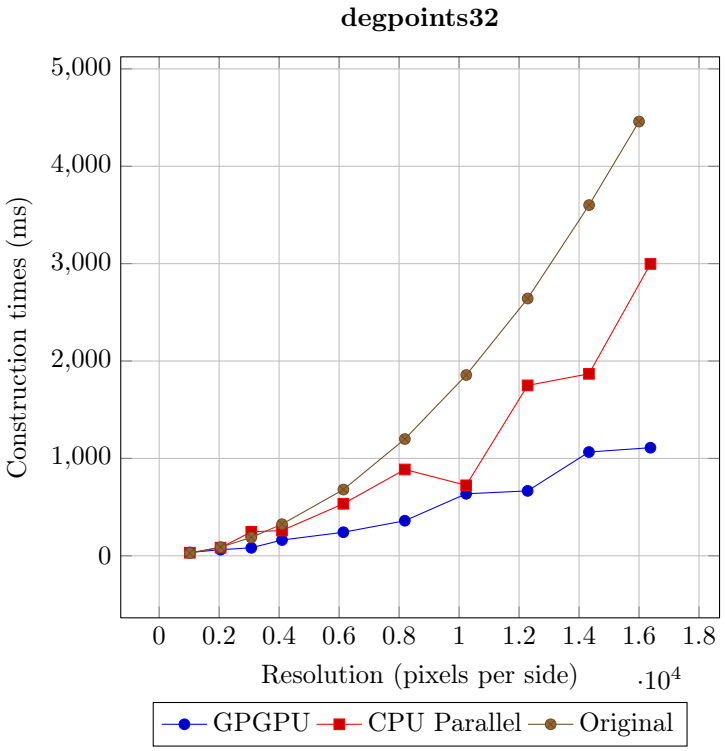


Figure 29: Comparisons between running times of the 3 implementations, environment degpoints32. Both "GPGPU" and "CPU-Parallel" exhibit significantly lower computational times at most resolutions.

6.3 Additional Tests of the GPGPU Implementation

In the previous section, we have outlined the general performance of the "GPGPU" implementation, but we omitted the incidence of the tile size t on the whole computations, showing instead the runs that exhibited the best runtimes. Figure 32 shows instead the construction times for ECMs of the environments Military, UUTest and aa64 at various resolution $r \times r$, varying the tile resolution $t \times t$. What we can see is that, simply, using a smaller number of bigger tiles leads to lower construction times. This is a fairly logical result, since tiles are processed sequentially by the "GPGPU" implementation, and using less tiles saves time during the various GPGPU launches and during merging. However, this also rules out the "myth" that operating with GPU buffers of certain sizes would lead to better results because some dimensions are more "GPU friendly" than others: such a "phenomenon" does not appear in our implementation.

We also tracked the amount of memory transferred from the GPU to the CPU by "GPGPU" and compared to the amount transferred by the "Original" implementation (Figures 30 and 31). We can notice that we definitely achieved our initial goal of decreasing the amount of memory transferred by performing additional steps on the GPU, since even at fairly high resolutions we end up transferring less than 50 MBs, versus the maximum of 976 MBs transferred by the original method. However, two additional considerations must be added:

- The maximum amount of GPU memory used by GPGPU can be rather high at high resolutions, ranging between 1500 and 1800 MBs. Luckily, most recent GPUs can deal with this requirement. Moreover, the amount of GPU memory employed is so big because we tried to minimize the memory allocations and the use of in-place operations (both computationally expensive), therefore lowering this memory requirement at the price of higher computational times could be investigated.
- "GPGPU" allocates pinned GPU memory for all the buffers that will have to be transferred to the CPU, in order to speed-up those operations. Without the use of pinned memory, the incidence of memory transfers on the total times would be higher.

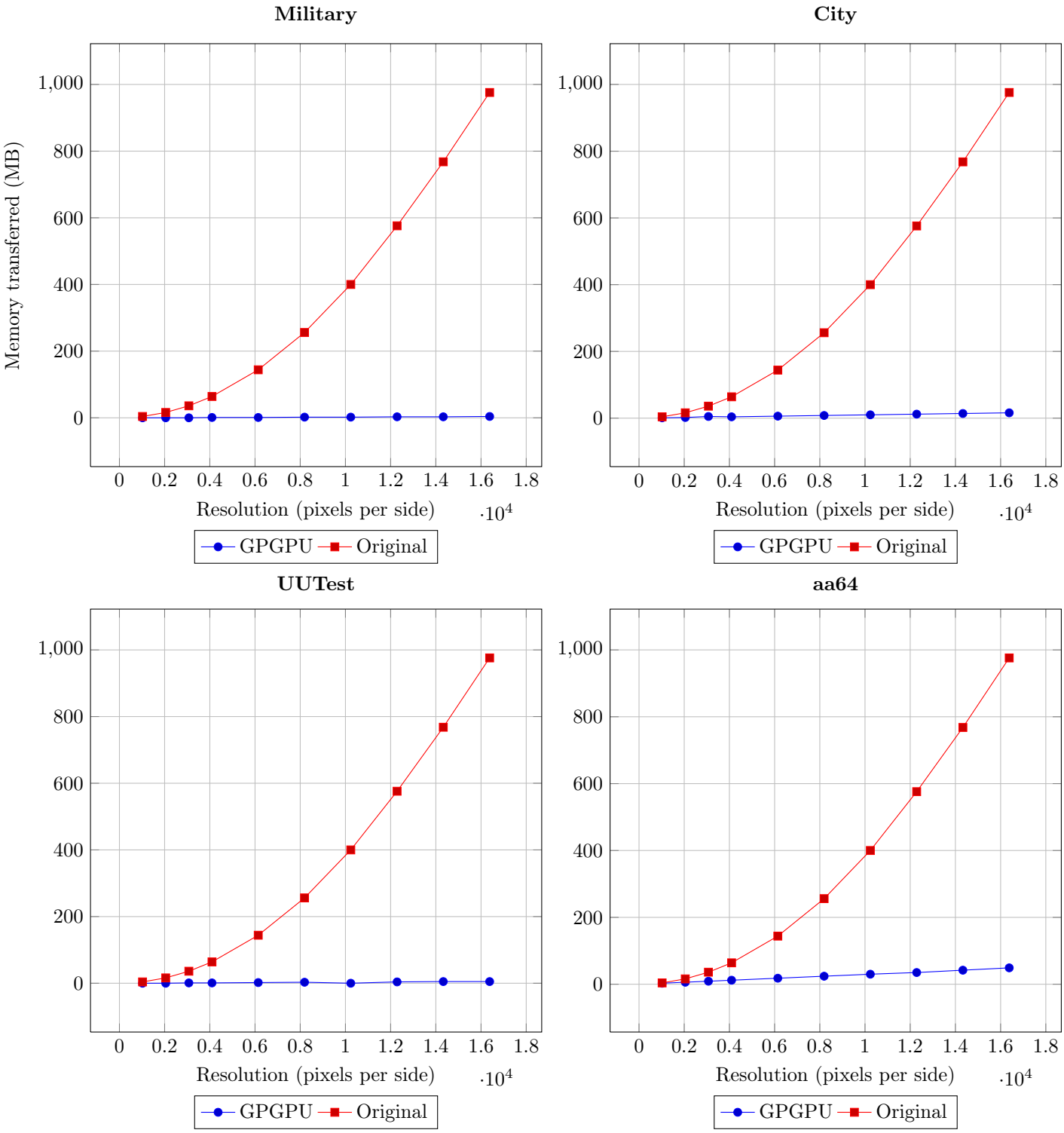


Figure 30: Amount of memory transferred from GPU to CPU (in MBs) by "GPGPU" and "Original" for Military, City, UUTest and aa64. 0 means less than 1 Megabyte. The amount of memory transferred by the new GPGPU implementation is very low: never greater than 50 MB. At the same time, we can notice how the memory transferred by "Original" does not depend on the environment tested but only on the total resolution, and how it scales quadratically with the with it.

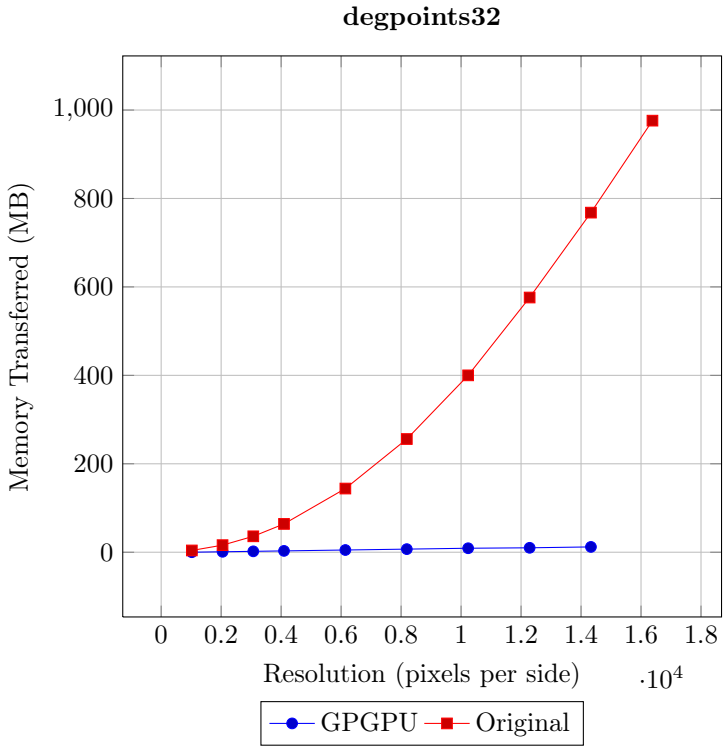


Figure 31: Amount of memory transferred from GPU to CPU (in MB) by "GPGPU" and "Original" for degpoints32. 0 means less than 1 Megabyte. The amount of memory transferred by the new GPGPU implementation is very low: being greater than 15 MB. Again, the amount of memory transferred by "Original" depends only on the total resolution and scales quadratically with it.

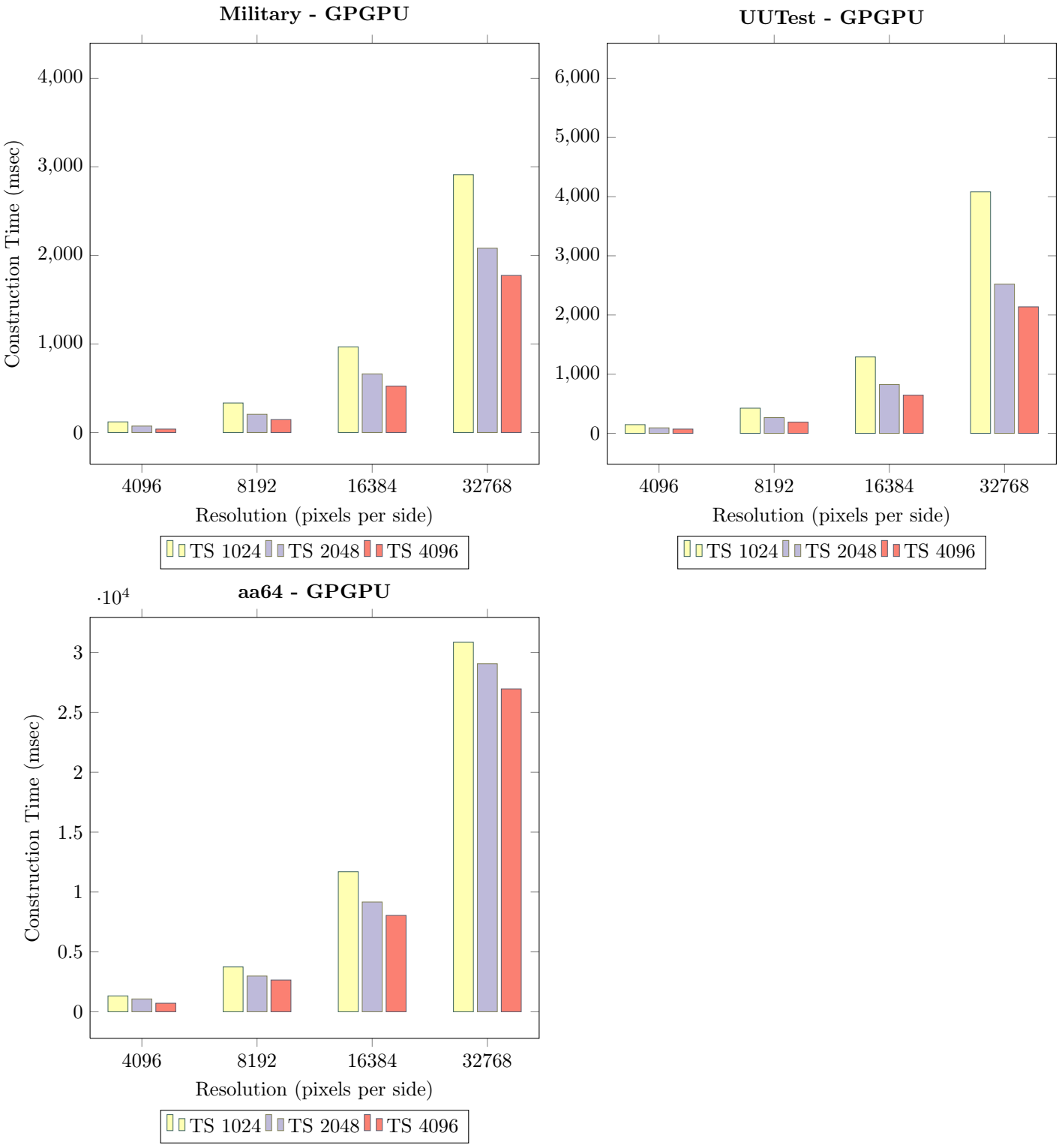


Figure 32: Incidence of the tile size in the construction times of the "GPGPU" implementation, environments Military, UUTest and aa64. We observe that using fewer but bigger tiles leads to the lowest times

6.4 Additional Tests of the CPU-Parallel Implementation

In this section we will outline the results of some additional tests performed with the "CPU-Parallel" implementation. We have already noticed how this new implementation can perform significantly better than the "Original" one, but now we will inquiry the incidence of the tile size parameter t in the performance.
Figures 33, 34 and 35 show the construction times of ECMs of 5 environments at certain total resolutions, varying the tile size. For each environment we plot both the result using the number of CPU threads employed by default by OpenMP on our test machine (usually a decent measure of the maximum load the hardware can support with no effort), 12, and the results using instead half of those threads, thus 6. The different environments show more or less the same "trends", with "aa64" slightly departing away once again. Given a total resolution $r \times r$, we tested using a tile $t \times t$ big, with t being one among $512, 1024, 2048, 4096$, or a subset of these values. A smaller tile size increases the total number of tiles the construction is split into: the constructions we are showing employed a number of tiles ranging from 4 to 4096. In the charts, lower bars represent better results (smaller times).

When 12 threads have been employed, from values of r that range from 4096 to 16384, the significantly lowest results are mostly obtained when a total number of 64 tiles is used. This translates to $t = 512$ for $r = 4096$, $t = 1024$ for $r = 8192$ and $t = 2048$ for $r = 16384$. For the environments Military and City, when $r = 16384$, the use of even more tiles, 128, yields even significantly lower results. Using 128 tiles in total instead is beneficial for ECMs produced when $r = 32768$, therefore having $t = 2048$. The environment aa64 presents similar results, but differences among values of $t = 64, 128$ are non significant when r is 4096 or 32768.

Things change when we use instead 6 threads in total. First, if we consider the best time for each total resolution, we notice a significant, but little, increase in times in all environments but aa64. Therefore, halving the number of threads negatively impacts the efficiency, although not strongly. Then, we can observe that now, for $r = 16384$, the best results are observed when only 16 tiles are used (instead of 64 or 128 as it was previously), therefore with $t = 4096$. Similarly, for $r = 32768$, the use of only 64 tiles, and therefore $t = 4096$, seems to be the most beneficial solution, as opposed to the previous 128. Once again, aa64 differs from these trends by exhibiting less significant differences.

In general, we notice that using a higher amount of smaller tiles yields better results, and this is quite logical since, unlike "GPGPU", "CPU-Parallel" processes the tiles in parallel. However, using too many small tiles then results in a "diminishing return" effect, with the amount of work being just too much for the number of CPU threads. This being said, the number of threads used during the construction is also a crucial factor: the use of less threads puts a limit on the amount of processing that can be done in parallel, therefore with only a few threads decreasing the number of total tiles employed (by increasing the tile size) proves to be beneficial. Picking the right value for t appears to be a tricky operation that would require perhaps additional investigation. In our experiments, setting t as $t = \frac{r}{16}$ or $t = \frac{r}{8}$ seemed to lead to the best results overall when 12 threads were employed, while in case of only 6 threads used, setting $t = \frac{r}{4}$ seemed to be the best option for higher resolution.

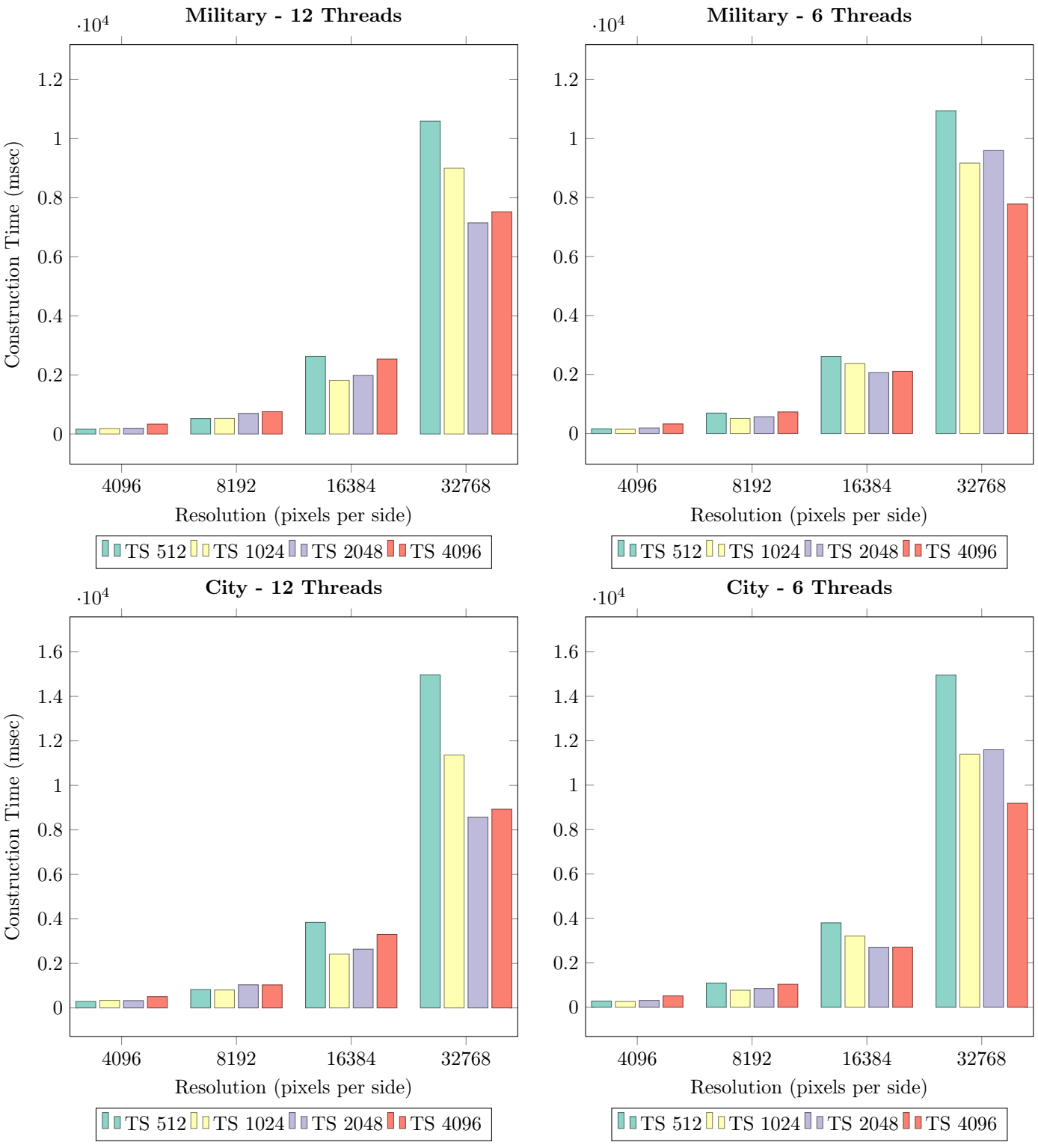


Figure 33: Incidence of the tile size in the construction times of the "CPU-Parallel" implementation, environments Military and City. Charts on the left employed 12 threads, while charts on the right only 6. Unlike "GPGPU", bigger using bigger tiles does not always lead to the lowest times. Picking the right tile size is hard.

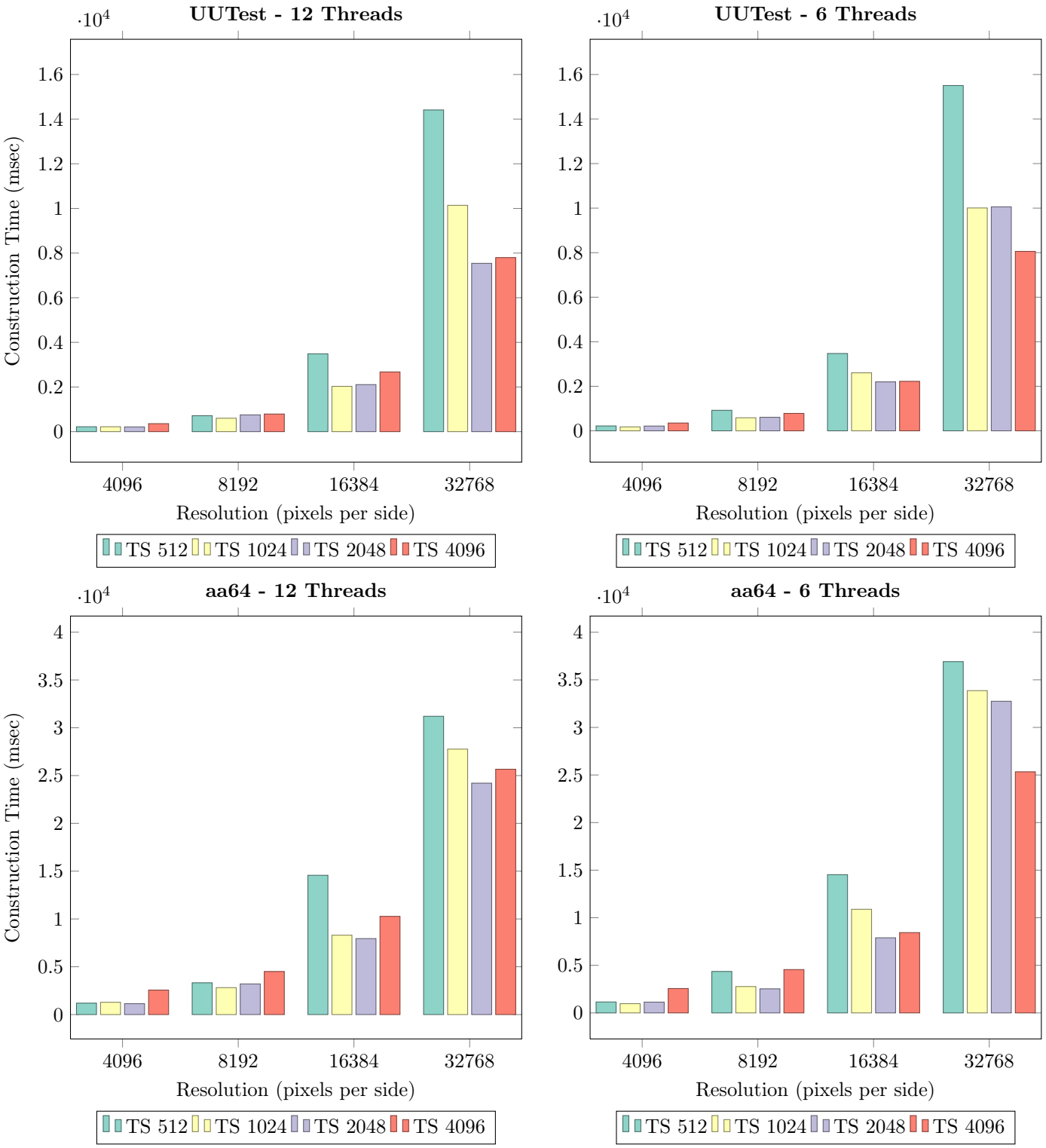


Figure 34: Incidence of the tile size in the construction times of the "CPU-Parallel" implementation, environments UUTest and aa64. Charts on the left employed 12 threads, while charts on the right only 6. Same considerations stated previously apply; however, the trend is less significant in aa64.

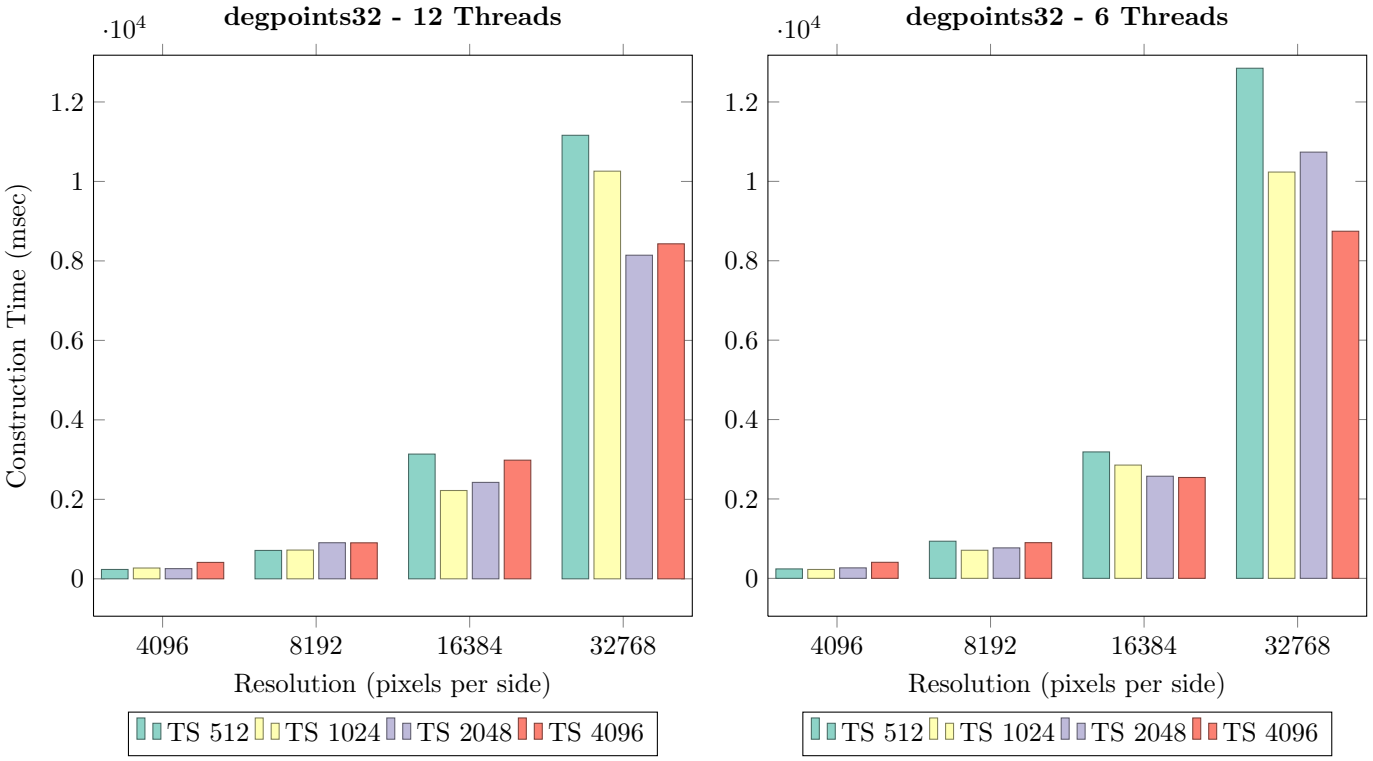


Figure 35: Incidence of the tile size in the construction times of the "CPU-Parallel" implementation, degpoints32 environment. Charts on the left employed 12 threads, while charts on the right only 6. Same considerations stated previously apply.

6.5 Considerations about High Resolutions and Precision

We have already showed how the 2 new implementations lead to considerably lower construction times for ECMs. Nonetheless, a multi-tiled approach also grants the advantage of computing Navigation Meshes at very high resolutions, basically almost removing any constraints related to the resolution of the approximated construction. If the total resolution is expressed as $r \times r$, the 2 implementations can compute ECMs up to $r = 4294967295$ (or better, the value of the MAX_UNSIGNED_INT C symbol). Previous charts show results for resolutions only up to about 32000 squared pixels for readability (at very high resolutions construction times trespass a certain order of magnitude), but we have also computed ECMs at resolutions greater than $100k$ squared pixels. In the new implementations, the main limits are represented by the maximum number of obstacles supported (which derives from the GPU color-depth used) and by a maximum number of Voronoi vertices and edge-points supported; the latter constraint is derived by the fairly high memory usage of the data-structures employed during the steps of the constructions, that can again require more memory than the available one, or single-buffer allocations that exceed again the limit of allocability.
Therefore, it becomes interesting to make some considerations on the effect that higher resolutions can have on the precisions of the Navigation Meshes. In the first 2 chapters we have explained how the approximated construction of the Medial Axis we employ can lead to some known issues, and we observed how some of them are mostly related to the resolution employed, assuming that increasing the resolution would have a positive impact on the final precision.

In order to investigate this, we included the environment "degpoints32" in our set of tested environments. As we have already noticed, an approximated construction like the method described in [HCK+99] fails at computing a correct MA of "degpoints32" because it fails at locating the single vertex in the center of the circumference, and it also fails at distinguishing distinct edges when they get too close to that central vertex. In [Cib13] these artifacts are named "weaving effect" and "flower effect", and are observed to appear because additional vertices and event-points are detected on the edges too. Therefore, an agglomeration of incorrect vertices/event-points are detected in the part close to the center of the circumference, and from the agglomeration pairs of almost overlapping edges are spawned. [Cib13] shows that the computation of an ECM of "degpoints32" based on the topology-oriented VRONI framework can instead generate a correct Navigation Mesh. Since our implementations support higher resolutions, we computed ECMs of "degpoints32" at resolutions that range from 4096 squared pixels to more than 80000 squared pixels. The ECM of "degpoints32" computed at 4096×4096 and 80000×80000 are shown in Figure 36. What we can notice is that at a high resolution the "flower effect" is slightly less accentuated, but in general, we can conclude that some tricky scenarios can not be computed correctly by an approximated method even if we massively increase the resolution. However, since an approximated method presents other advantages and could end up being used to compute ECM of error-prone scenarios anyway, being able to process them at high resolutions, having then less artifacts in the result, could still prove to be useful.

However, increasing the resolution of our approximated method does improve the precision of some of the ECMs computed, and it can potentially remove some mistakes that derive from the low resolution of the approximation taking place. We created the example "simplePrecision" to try to outline this: "simplePrecision" is a very big environment of 200 x 200 meters, bigger than any other environment tested, featuring a point and a line segment at a distance of 0.025 meters on the X axis. Hence, the point P is located at position $(2.0, 0.0)$ and the closest point to P on the segment S, Q, is located at position $(2.025, 0.0)$. Both the x values of P and Q are perfectly representable by the $float$ and *double* datatype. The point P and the segment S should generate a parabola as part of their MA, being the a parabola the locus of points exhibiting the same distance between a point and a line. Figure 37a shows an ECM of "simplePrecision" produced at resolution 8192: we can observe something similar to a parabola between the point and the segment, but in fact, geometrically, that configuration consists of 5 segments overlapping, 3 of them connecting the 2 points that,

being very close, are incorrectly visualized as just one. Figure 37b instead shows an ECM of the same environment computed at a resolution of 32768 squared pixels: we can notice how no vertices appear on the parabolic part of the Medial Axis, which is this time instead geometrically expressed as a parabola that correctly goes between the point and the line. It is worth noticing that the second ECM could not have been computed with the "Original" implementation, given its high resolution.

Understanding why this happens is, unfortunately, quite hard: what is clear is that at lower resolutions, the GPU rasterization step described in [HCK+99] fails at producing a configuration where 2 pixels signaling the presence of obstacles (black) are separated by at least 2 other pixels to which two different colors (other than black) are assigned. More precisely, considering the case of "simplePrecision" with seeds (obstacles) point P and segment S, if pixel $Pixel_1(p1_x, p1_y)$ is assigned to seed point P, then we need:

1. A pixel to its right at position $Pixel_2(p1_x+1, p1_y)$ that must feature the color assigned to the distance mesh of point P;
2. Then, a pixel $Pixel_3$ to the right of $Pixel_2$ at position $(p1_x+2, p1_y)$ that must feature the color assigned to the distance mesh of segment S, color that should be different from the color of pixel $Pixel_2$;
3. In the end, a third pixel $Pixel_4$ located at position $(p1_x+3, p1_y)$ that must be mapped as part of the obstacle segment S, featuring thus color black.

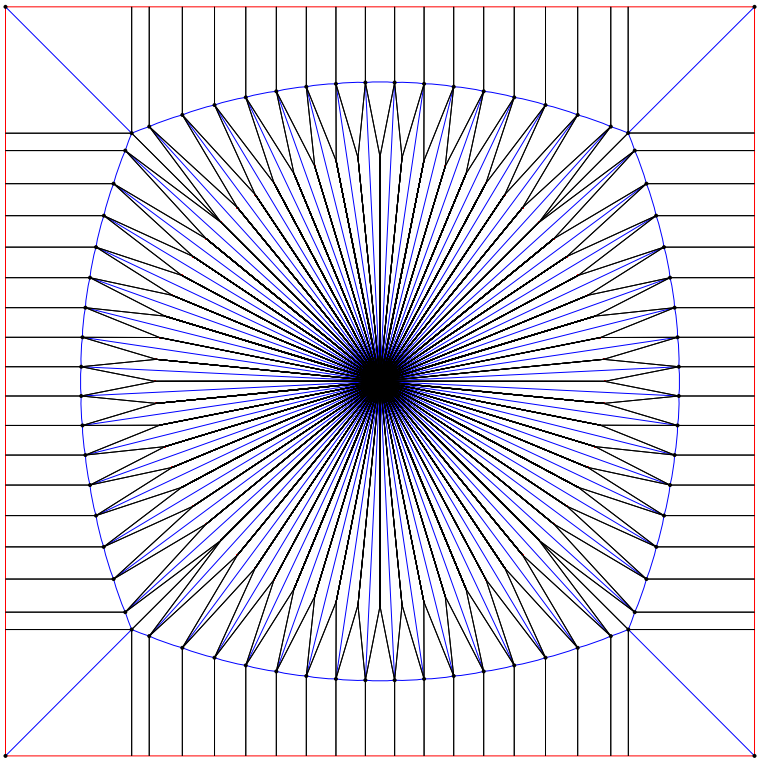
We can then notice that, given the obstacle P, we need 3 distinct pixels in order to have a correct configuration involving the closest obstacle S, 2 plus the pixel identifying the other obstacle. The "virtual" size of a pixel can be expressed as

$$pixelSize = \frac{BBoxWidth}{r}$$

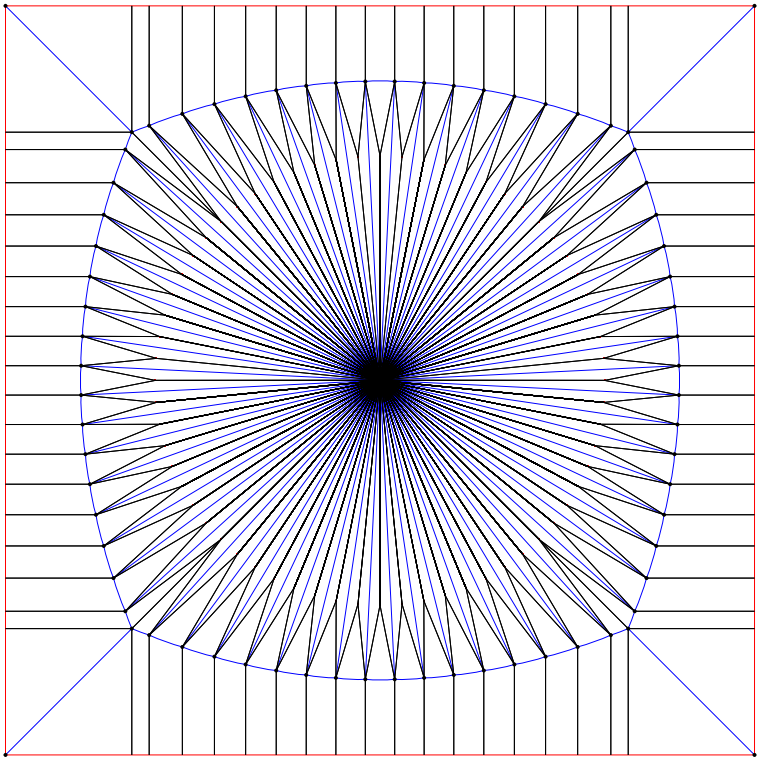
where r comes from the resolution used for the construction while $BBoxWidth$ is the width of the environment in world coordinates. Then, to produce the correct configuration of 2 obstacles at distance $dist$ in world coordinates we should make sure that

$$dist >= ceil(3 * pixelSize).$$

We can notice how this can explain the example of the previous paragraph. This being said, this is a very simplistic way of reasoning that ignores the exact way in which OpenGL translates the world coordinates into grid coordinates of rasterization, and that also does not take into account possible precision problems due to the use of floats in the process.

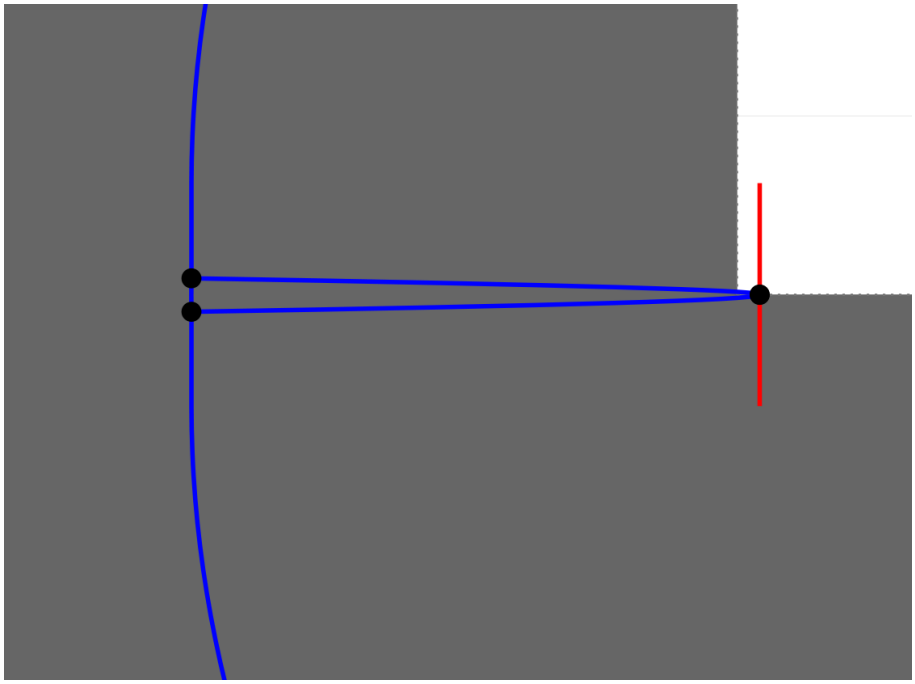


(a) Lower resolution ECM of "degpoints32" computed at 4096 squared pixels. We can notice the "black treacle" of points that are generated close to the center of the circumference (flower effect), that generate at some point pairs of partially overlapping edges (visualized as thicker) (weaving effect).

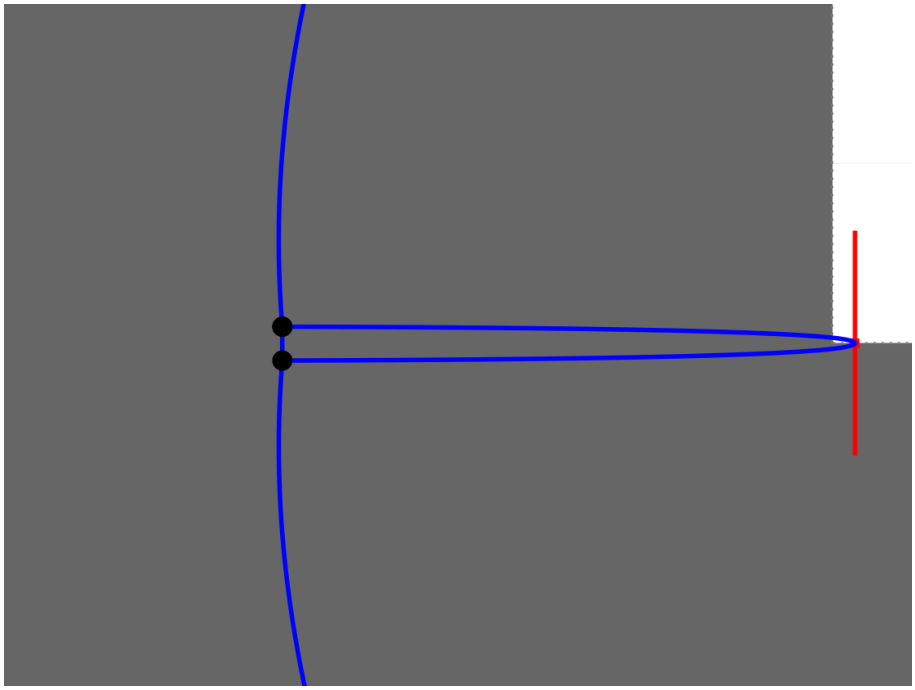


(b) Higher resolution ECM of "degpoints32" computed at 80000 squared pixels. The "black treacle" of black points around the center is slightly smaller (noticeable in the areas closer to the boundaries of what looks like a black circle). However, the ECM is still fundamentally incorrect.

Figure 36: Low resolution and high resolution ECMs of environment degpoints32



(a) At a resolution of around 16000 squared pixels, we observe an incorrect configuration. We should see a "thin" parabola as part of the Medial Axis (blue). We can notice a similar curve, although this curve is in fact represented by 4 pairs of overlapping edges connecting the 2 vertices on right (black) to the 2 other vertices on the left, and an additional invisible edge connecting the 2 black vertices on the right. The 2 black vertices, being too close, are incorrectly displayed as one.



(b) At a resolution of around 32000 squared pixels, we observe a configuration. Part of the Medial Axis (blue) now forms a parabola. We can notice it because no vertices (black points) are located on what looks like a parabola, although visualization of vertices is clearly on since the ones on the left are correctly visualized.

Figure 37: Part of the ECM of "simplePrecision" computed at two different resolutions. The only obstacles are a segment (red) and point (also red), which is hardly noticeable being too close to the segment, but they can be observed by zooming-in Figure 27f.

7 Conclusions and Future Work

In this work we showed how to introduce parallelism, both CPU-based and GPGPU, in the construction of ECM Navigation Meshes. Doing so involved shaping a new multi-tiled approach that allows first to compute and store correctly parts of a Medial Axis, and then to merge this parts into a final, high-resolution ECM. We have also shown how we can use multiple CPU threads during both the processing of the tiles and the merge step, assembling the "CPU-Parallel" implementation. Moreover, we have investigated the use of GPGPU for speeding up the construction of an ECM, describing a couple of GPGPU steps that extend the work already performed by the GPU in the current ECM framework, and then implemented them in the new implementation "GPGPU".

We have tested both implementations on a mixture of real-life scenarios and made-up examples: in most environments, "GPGPU" leads to construction times that are on average 62% smaller than the original ones, while "CPU-Parallel" similarly leads to construction times that are on average 28% smaller. In addition, the new implementations allow to compute approximated ECMs but at very high resolutions: although this is not enough to compute correct ECMs of some degenerate cases, it can still decrease the number of artifacts introduced in the Navigation Meshes. At the same time, we showed as some precision errors instead can be completely solved if we are allowed to use any kind of value (or very big ones) as a total resolution.

Still, the new described and implemented approaches have a number of limitations. Mainly, not being able to handle correctly some known degenerate cases is still a big issue, considered also that some topology-oriented approaches seem instead to be capable of solving those cases [Cib13, HH09] at "bearable" computational costs. However, our "GPGPU" implementation seems to be significantly more efficient, which poses again the familiar trade-off between precision and speed. Moreover, subdividing an ECM computation into tiles that are then merged together at the end is a very tricky operation, and because of this our implementations are still quite "unstable", especially when more complicated environment needs to be handled at high resolution or using a very small tile size. Although the main geometrical ideas behind the approach have been outlined and handled correctly, there are still probably many specific/degenerate cases that require further investigation, and that are causing the errors of current implementations.
Another thing to consider is that the "GPGPU" implementation comes with additional drawbacks: it relies heavily on GPU memory and GPGPU optimizations, and thus the code is fairly complicated, and can eventually prove to be difficult to maintain. Again, it would be more reasonable to consider the "GPGPU" implementation as a "starting point" prototype that shows how GPGPU can be used to achieve certain benefits, but different solutions and additional GPGPU-related optimizations need to be investigated and probably included before the prototype can be considered a mature implementation. Moreover, the current GPGPU implementation relies on CUDA, and, therefore, being CUDA a technology from NVIDIA, it works only on configurations featuring a fairly modern (1/2 years old maximum) NVIDIA GPU and updated NVIDIA drivers. Sure, the codebase could be adapted to make use of similar but more cross-supported technologies such as OpenCL; unfortunately, the current code is built upon the "CUDA Runtime API", a higher-level layer that abstracts many of the lower-level calls of the "CUDA Kernel API". Being the OpenCL API more similar to the CUDA Kernel API, modifying the code would require additional logical work, considered also that not all the CUDA calls have a clear, direct counterpart in the OpenCL API. In this sense, a solution could be to employ a simple library that abstracts common functionalities of both CUDA and OpenCL, allowing to write host-code that can be executed on both platforms by just changing a flag: the project "GPGPUUnit" [Bon10] aims at doing that, and it was developed in parallel to this work. This being said, CUDA still offers some features that OpenCL instead lacks, and it seems to perform better.

A more-general drawback outlined by the work is that it can be difficult to take advantage, at the same time, of CPU-based parallelism and GPGPU for the construction of ECMs, because the initial construction steps involve graphics hardware, and until they are finished we don't have any input data (Voronoi diagram description) for computations that

can use CPU threads. Therefore, we managed to increase the degree of parallelism only when not increasing the amount of steps performed on the GPU, and similarly, when we increase the number of GPU-powered steps we decrease the degree of CPU-based parallelism: the two implementation are almost mutually exclusive. It is also important to keep in mind that memory transfers between GPU and CPU can not be parallelized. This being said, the "GPGPU" approach could be extended to use multiple GPUs, which would process at the same time different tiles. This will in turn require additional work with CPU parallelism since each GPU needs to be "controlled" by a different CPU thread.

Therefore, there are many possibilities for future work. We have already discussed the ones that involve extending and improving the GPGPU/CPU-Parallel routines. Additional studies could inquire the relationship between a certain resolution/size of an environment/its complexity and the kind of mistakes that appear in the approximated ECM computed. In the same way, since the resolution parameter is almost not a constraint anymore, new investigations on which kind of configurations, and thus which kind of environments, require very-high resolutions to be handled correctly/well enough could follow, especially if paired with comparisons to solutions produced by topology-oriented or exact approaches.
Given the low computational times produced by the new implementations, it could be investigated if any of them could be adapted for computing dynamic changes of an ECM during a simulation. In the same way, either of them could become part of a new "mixed" framework that could construct Navigation Meshes by first employing an approximated but very fast approach (like "GPGPU") and by subsequently "post-process" the output structure using a more exact approach (topology-oriented) in order to try to correct possible mistakes or precision errors.

If successful, the above studies could have a strong impact on practical applications. A framework that could precisely distinguish very close objects, computing a correct graph, would increase the number of environments that could handled automatically. Currently, crowd simulation approaches are advanced enough to be able to handle a huge number of characters moving inside very big environments that are reproductions of existing ones (airports, big train stations). Those environments can feature narrow passages or oddly-shaped obstacles being very close to each-other, which are scenarios that can lead to incorrect auto generated Navigation Meshes due to precision errors, even using state of the art approaches. Those errors in the supporting data structures are generally corrected manually by users, which is a costly and inefficient solution. Hence, a gap appears to exist in the area of crowd simulation, with advanced approaches that are able to simulate convincing flows of a huge amount of characters in huge environments, but that are not capable of autonomously generate the correct support structures that those simulation would require. This research work could help filling this game.
At the same time, being able to compute dynamic changes of a Navigation Mesh would mean running convincing crowd simulations on adaptable environments, environments that could then change according to user input or as a consequence of other emerging events (i.e. fire burning tree obstacles). This is a known open problem in the field of crowd simulation, since it is hard to find/design a Navigation Mesh structure which is at the same time a compact and efficient description of the walkable space of an environment and can be dynamically changed efficiently. Since crowd simulation is often used as a validation mechanism for architectural design/planning, overcoming the issue of dynamic changes would lead to softwares in which users could draw and place on the screen objects in the most crowded areas during crowd simulations, receiving real-time feedback on the impact of the new object in the behavior of the crowd. This would drastically cut the costs of such important validations.
Dynamically changing Navigation Meshes are also a known issue of current video games, that more and more feature crowds of humans that are not only added to increase immersion, but often play an important role in terms of interaction with the players. Since crowd simulation and path-planning can strongly impact the final artificial behavior of characters, it is important that they always keep coherent with the changes in the environment that surrounds them. Recently, modern first-person shooters like "Battlefield" and "Call of Duty" strongly emphasize how all their new maps can be dynamically altered by the missiles shot or by the

airstrikes, leading to an immersive and always diverse game experience. Since first-person shooters usually feature a rather small number of total characters on the field (up to 25), the technical constraints can still be "bypassed". However, achieving the same result in a more populated "sandbox game" (like a modern Grand Theft Auto) would probably require crafting a complex and stable solution for the problem of dynamically modifiable Navigation Meshes. Therefore, the approaches and results illustrated in this work could be used also as a starting point to improve modern interactive games, eventually making them capable of supporting the high number of explosions and ignitions that players deserve to experience.

References

[ACV05] H. Alt, O. Cheong, and A. Vigneron. The voronoi diagram of curved objects. *Discrete & Computational Geometry*, 34(3):439–453, 2005.

[Aur91] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[BA92] J.W. Brandt and V.R. Algazi. Continuous skeleton computation by voronoi diagram. *CVGIP: Image Understanding*, 55(3):329–338, 1992.

[Ble90] G.E. Blelloch. Pre x sums and their applications. *Synthesis of Parallel Algorithms*, pages 35–60, 1990.

[Ble10] A. Bleiweiss. Parallel compact roadmap construction of 3d virtual environments on the gpu. In *Intelligent Robots and Systems, 2010 IEEE/RSJ International Conference on*, pages 5007–5013. IEEE, 2010.

[Bon10] R Bonfiglioli. Gpgpuunit library, 2010. https://bitbucket.org/winterismute/gpgpuunit/.

[cga] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[Cib13] A Cibotaru. Alternative algorithms for computing explicit corridor maps using exact and topology-oriented paradigms. Master's thesis, Utrecht University, 2013.

[CSW95] F. Chin, J. Snoeyink, and C. Wang. Finding the medial axis of a simple polygon in linear time. *Algorithms and Computations*, pages 382–391, 1995.

[CTMT10] T.T. Cao, K. Tang, A. Mohamed, and T.S. Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90. ACM, 2010.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M.H. Overmars. *Computational Goometry : Algorithms and Applications*. Springer, 3rd edition, 2008.

[DCSM96] A C Dusseau, David E. Culler, Klaus E. Schauser, and Richard P. Martin. Fast parallel sorting under logp: Experience with the cm-5. *Parallel and Distributed Systems, IEEE Transactions on*, 7(8):791–805, 1996.

[Den03] M. Denny. Solving geometric optimization problems using graphics hardware. In *Computer Graphics Forum*, volume 22, pages 441–451. Wiley Online Library, 2003.

[Dev92] O. Devillers. Randomization yields simple $o(nlog * n)$ algorithms for difficult omega (n) problems. *International Journal of Computational Geometry and Applications*, 2(1):97–111, 1992.

[FEC02] R. Fabbri, LF Estrozi, and L.D.F. Costa. On voronoi diagrams and medial axes. *Journal of Mathematical Imaging and Vision*, 17(1):27–40, 2002.

[Ger10] R. Geraerts. Planning short paths with clearance using Explicit Corridors. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.

[GM07] N K Govindaraju and Dinesh Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Computing*, 33(10):663–684, 2007.

[GO06] R. Geraerts and M.H. Overmars. Creating high-quality roadmaps for motion planning in virtual environments. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 4355–4361. IEEE, 2006.

[GO07] R. Geraerts and M.H. Overmars. The corridor map method: a general framework for real-time high-quality path planning. *Computer Animation and Virtual Worlds*, 18(2):107–119, 2007.

[GO08] R. Geraerts and M.H. Overmars. Enhancing corridor maps for real-time path planning in virtual environments. *Computer Animation and Social Agents*, pages 64–71, 2008.

[HB10] J Hoberock and N Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[HCK+99] K.E. Hoff, III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. *International Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[Hel01] M. Held. Vroni: An engineering approach to the reliable and efficient computation of voronoi diagrams of points and line segments. *Computational Geometry*, 18(2):95–123, 2001.

[Hel11] M. Held. Vroni and arcvroni: Software for and applications of voronoi diagrams in science and engineering. In *Voronoi Diagrams in Science and Engineering, 2011 Eighth International Symposium on*, pages 3–12. IEEE, 2011.

[HH09] M. Held and S. Huber. Topology-oriented incremental computation of voronoi diagrams of circular arcs and straight-line segments. *Computer-Aided Design*, 41(5):327–338, 2009.

[HKKOO11] S. Hong, S. Kyun Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. *SIGPLAN Notices*, 46(8):267, 2011.

[HSO07] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[KL02] S. Koenig and M. Likhachev. D^* lite. In *Proceedings of the National Conference on Artificial Intelligence*, pages 476–483. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[KWmH10] D. Kirk, W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[Lee82] D.T. Lee. Medial axis transformation of a planar shape. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (4):363–369, 1982.

[LPT98] G. Liotta, F.P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM Journal on Computing*, 28(3):864–889, 1998.

[MG11] D Merrill and A Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[MS87] S.N. Meshkat and C.M. Sakkas. Voronoi diagram for multiply-connected polygonal domains ii: Implementation and application. *IBM journal of research and development*, 31(3):373–381, 1987.

[RT06] G. Rong and T.S. Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116. ACM, 2006.

[RT07] G. Rong and T.S. Tan. Variants of jump flooding algorithm for computing discrete voronoi diagrams. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on*, pages 176–181. IEEE, 2007.

[SGM05] A. Sud, N. Govindaraju, and D. Manocha. Interactive computation of discrete generalized voronoi diagrams using range culling. In *Proc. International Symposium on Voronoi Diagrams in Science and Engineering*, 2005.

[SH75] M.I. Shamos and D. Hoey. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 151–162. IEEE, 1975.

[SIII00] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementationÑan approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.

[SN87] V. Srinivasan and L.R. Nackman. Voronoi diagram for multiply-connected polygonal domains i: Algorithm. *IBM Journal of Research and Development*, 31(3):361–372, 1987.

[SOM04] A. Sud, M.A. Otaduy, and D. Manocha. Difi: Fast 3d distance field computation using graphics hardware. In *Computer Graphics Forum*, volume 23, pages 557–566. Wiley Online Library, 2004.

[Wm11] W Hwu Wen-mei. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.