Università degli Studi di Milano

Facoltà di Scienze e Tecnologie
Corso di Laurea Magistrale in Informatica

Approximated Recentered Anisotropic Voronoi Diagram Restricted on Two-Manifolds for Topology Optimization

Relatore Prof. Marco Tarini

Autore Luigi Rapetta

Matricola 985804

a.a. 2023-24

Contents

1 Introduction **3**

2 Background **5**
2.1 Voronoi Diagram 5
2.1.1 Centroidal Voronoi Tessellation 5
2.1.2 Restricted Voronoi Diagram 6
2.1.3 Constrained and Restricted CVT 6
2.1.4 Geodesic Voronoi Diagram 6
2.2 Seeds relaxation 6
2.2.1 Lloyd's algorithm 6
2.2.2 Broyden–Fletcher–Goldfarb–Shanno algorithm 7

3 State of the art **9**
3.1 CVT-based remeshing 9
3.1.1 GVD-based 9
3.1.2 Mesh parameterization 10
3.1.3 RVD-based 10
3.2 Other applications of VDs 10

4 Our approach **11**
4.1 Working on a graph 11
4.1.1 Geodesics and graph's connectivity 12
4.1.2 Weighting by areas 13
4.1.3 Initial seed selection 13
4.1.4 Anisotropic metrics 13
4.1.5 Averaging tangent spaces 14
4.2 Voronoi partitioning 15
4.3 Lloyd's relaxation 16
4.3.1 Greedy phase 17
4.3.2 Precise phase 19
4.3.3 Conservative heuristics 21

5 Implementation **23**
5.1 Tangent field file 23
5.2 Structs and classes overview 24
5.3 Subtree costs and weights 24
5.4 Priority heap 25
5.5 Precise phase optimizations 26

6 Results **27**
6.1 Tessellation evaluation 27
6.2 Performance evaluation 30
6.2.1 Plane mesh test - 16 seeds 30
6.2.2 Plane mesh test - 1024 seeds 33
6.2.3 Hippo (hi-res) mesh test 34
6.3 Additional tests 36

7 Conclusions **39**
7.1 Considerations on the results 39
7.2 Limitations 39
7.3 Further work 41

Bibliography **43**

Chapter 1

Introduction

In the field of mesh processing, it is common to apply topology optimization algorithms to reduce the number of vertices and triangles in a given mesh. This is often necessary to deal with noisy data. Such procedures are referred to as *remeshing* algorithms.

One approach to remeshing is using *Centroidal Voronoi Tessellation* (CVT). In this method, the optimized topology is computed by first partitioning the surface of the mesh into a *Voronoi Diagram* (VD), whose seed points (also called *seeds*) are relaxed so that they coincide with the centroids of their respective regions. These seeds are treated as the vertices of the optimized mesh, and later used in the *Delaunay Triangulation*, which computes the optimized topology.

CVT requires to compute repeatedly the VD of the surface. VDs can be computed by either using the Euclidean distance or the *geodesic distance* of points from seeds. The latter, being the length of the shortest path of two points on a surface, is usually more expensive to calculate than the former.

Many approaches use the Euclidean distance to save on performance, like in [14] and [15], by computing an approximation of the VD called *Restricted Voronoi Diagram* (RVD) [6]. These approaches suffer from resulting centroids not being constrained on the mesh's surface with the need to project them.

In this thesis, we propose an algorithm for computing approximated, geodesic VDs for two-manifolds through a discrete clustering approach, in which the centroids are constrained on the mesh from the get-go without any need for projecting them onto the surface. The algorithm generates a graph of the topological elements of the mesh and assigns seeds to its nodes, where any subsequent partitioning and relaxation of seeds are simple graph operations. Our work supports both isotropic and anisotropic metrics for guiding the remeshing.

The first topic of the thesis is a brief explanation of the basic concepts found throughout this work. That is followed by a discussion of previous approaches to the problem. Then, our own algorithm is presented by introducing it at a high level of abstraction, followed by details of our implementation, optimizations, and results.

Chapter 2

Background

This chapter focuses on the basics of frequent concepts found in this thesis.

2.1 Voronoi Diagram

Given n seeds, $X = \{x_i\}_{i=1}^n \subset \mathbb{R}^N$, the *Voronoi Diagram* (VD) of X in $\mathbb{R}^N$ is defined as n Voronoi cells $C = \{\Omega_i\}_{i=1}^n$ such that

$$\Omega_i = \{x \in \mathbb{R}^N \quad | \quad \|x - x_i\| \leq \|x - x_j\|, \forall j \neq i\}.$$

2.1.1 Centroidal Voronoi Tessellation

From the definition of a VD, we can introduce Centroidal Voronoi Tessellation (CVT), a Voronoi tessellation such that each seed x_i coincides with the center of mass x_i^* (also referred to as *barycenter* or *centroid*) of its region Ω_i. The barycenter x_i^* is defined as

$$x_i^* = \frac{\int_{\Omega_i} \rho(x) x dx}{\int_{\Omega_i} \rho(x) dx}.$$

This can be interpreted as a weighted average, where $\rho(x) > 0$ is a user defined density function. A constant ρ leads to a *uniform* CVT.

It can be stated [4] that a CVT is a critical point of the following function:

$$F(X) = \sum_{i=1}^{n} \int_{\Omega_i} \rho(x) \|x - x_i\|^2 dx.$$

A function of this form is often used as an energy function to minimize in the *relaxation* process in which, given a non-centroidal Voronoi tessellation, its seeds are iteratively moved in order to make it converge towards a CVT.

2.1.2 Restricted Voronoi Diagram

Given a surface $S \subset \mathbb{R}^3$ and a set of seeds $X = \{x_i\}_{i=1}^n \subset \mathbb{R}^3$, the Restricted Voronoi Diagram (RVD) on S is defined as $\mathcal{R} = \{R_i\}_{i=1}^n$, where $R_i = \Omega_i \cap S$ (namely, the *restriction* of Ω_i onto S)[6].

2.1.3 Constrained and Restricted CVT

The *Constrained CVT* (CCVT)[5] is an extension of CVT such that $X \subset S$ (the seeds are *constrained* onto S) and $R_i = \Omega_i \cap S$. It is a critical point of the following function:

$$F(X) = \sum_{i=1}^{n} \int_{R_i} \rho(x) \|x - x_i\|^2 dx.$$

If we minimize the function above by not constraining the seeds onto S, we identify a *Restricted CVT* (RCVT).

2.1.4 Geodesic Voronoi Diagram

Given a surface $S \subset \mathbb{R}^3$ and a set of seeds $X = \{x_i\}_{i=1}^n \subset S$, the *Geodesic Voronoi Diagram* (GVD) of X in S is defined as n Voronoi cells $C = \{\Omega_i\}_{i=1}^n$ such that

$$\Omega_i = \{x \in S \quad | \quad d(x, x_i) \leq d(x, x_j), \forall j \neq i\},$$

where d is the *geodesic distance*, which is the length of the shortest path of two points on a surface.

By contrast, all the forms of VD previously introduced use the Euclidean distance.

2.2 Seeds relaxation

Generating a CVT is an iterative process. Given an initial configuration of seeds, these may not necessarily lead to a centroidal diagram. Therefore, the seeds need to be procedurally moved until a CVT is reached. This process is called *relaxation* (also known as *recentering*).

2.2.1 Lloyd's algorithm

Among several methods to do relaxation, there is *Lloyd's algorithm*. Given a VD, its seeds are moved to the barycenters of their regions. Then, a VD is computed again with the new positions of the seeds. This two-step process is repeated until convergence is reached. The VD at convergence is a CVT.

2.2.2 Broyden–Fletcher–Goldfarb–Shanno algorithm

Another known method to relax seeds is the *Broyden–Fletcher–Goldfarb–Shanno* (BFGS) algorithm. This method is used for several optimization problems, and it is not confined to tessellation alone. In our case, it explicitly minimizes the function $F(X)$ introduced in the previous sections.

The algorithm begins with an initial configuration of seeds X_0. Then, the descent direction p_k at the k-th step is computed by solving

$$B_k p_k = -\nabla F(X_k),$$

where B_k is an approximated *Hessian matrix*[1] of F at X_k, and $\nabla F(X_k)$ is the gradient of the function at X_k. The next configuration X_{k+1} is found by doing a *line search*[2] along p_k in order to minimize $F(X_k + \gamma p_k)$, where $\gamma > 0$.

A variant of this method is *Limited-memory BFGS* (L-BFGS), which approximates BFGS while using limited memory resources. This method was implemented by Liu *et al.* [11] in the context of CVT, and they demonstrate how much faster it is than Lloyd's method.

[1] A Hessian matrix is a square matrix of second order derivatives that describes the local curvature of a function with a large number of variables (the seeds, in our case).

[2] A line search is an iterative method to find a local minimum of a given function, which computes a step size that states how far the current solution should move along a descent direction.

Chapter 3

State of the art

This chapter focuses on discussing previous work related to the problem we aim to solve.

3.1 CVT-based remeshing

In this section, a series of techniques for CVT-based remeshing of a 3D mesh are discussed, briefly highlighting advantages and drawbacks.

3.1.1 GVD-based

An approach to CVT is computing a GVD of a surface. Peyré and Cohen [12] propose a discrete approximation of the geodesic distance to compute GVD. It involves a modification of the *Fast Marching* algorithm [13], the latter being a modification of Dijkstra's algorithm. Despite incorporating the relaxation process, their method provides an iterative generation of seeds that reduces the amount of relaxation to be done. Kunze *et al.* [9] compute GVD on parametric surfaces following a divide and conquer scheme, in which the boundaries of the diagram are iteratively defined by computing the *equidistant set*[1] of two seeds. Such method lacks any relaxation.

This kind of approach suffers from the need for computing geodesics, which, despite their approximations, are computationally expensive. An advantage would be that there is no need to project the centroid of the region onto the surface, since it belongs to it from the get-go. In addition, such regions would always be contiguous, no matter where the seeds are positioned.

[1] An equidistant set is a set of elements with equal distance between two other sets.

3.1.2 Mesh parameterization

Another approach to CVT is mesh parameterization [1, 2]. The mesh is unfolded onto a 2D space into a series of patches, the number of which depends on the *genus*[2] of the mesh, and several scalar maps are computed to serve as a complete substitute for the input geometry [1]. These maps are then processed and sampled. The samples are then used as an initial configuration for building a centroidal VD [2] in the 2D space. The remeshed patches are then stitched together in the 3D space.

The drawback of this approach mainly resides in how to stitch together the remeshed patches, which is not trivial. A high genus of the mesh also increases the number of patches, incommoding the reconstruction.

3.1.3 RVD-based

RVD is another technique for CVT. Valette *et al.* [14] compute an approximation of the RVD by discrete clustering of vertices and triangles, with both isotropic and anisotropic metrics for guiding the remeshing. The clustering algorithm acts iteratively on the elements of the regions' bounds by checking whether or not adding an element to another neighbouring region would benefit the minimization of the energy function. Yan *et al.* [15] propose an algorithm that computes RVD exactly. It uses a kd-tree to identify the intersections of triangles with the Voronoi cells and L-BFGS for relaxation, as proposed by Liu *et al.* [11].

These methods tend to be fast due to the use of the Euclidean distance instead of geodesics. Yet, they tend to return poor results in the presence of degenerate triangles in the input mesh, and there's the need to project seeds onto the mesh. Moreover, regions may not always be contiguous due to the use of the Euclidean distance.

3.2 Other applications of VDs

Although the main subject of the discussion is CVT, some work related to discrete clustering of graphs and anisotropic metrics is worth mentioning, especially since our method implements such concepts.

Erwig [7] proposes to partition an input graph into a VD through discrete clustering for networking-related purposes. A modified Parallel Dijkstra is used as the clustering algorithm, in which the tree of shortest paths from nodes to the seed is computed for each region. Dijkstra's frontier is implemented as a heap, minimizing the time needed to fetch the node with the shortest distance from its seed.

Brivio *et al.* [3] propose to use an anisotropic centroidal VD for browsing large datasets of images more efficiently.

[2] The genus is, namely, the number of holes of a surface. More in detail, it is the maximum number of cuts along non-intersecting closed curves without making a resulting disconnected manifold.

Chapter 4

Our approach

This chapter discusses our work at a high-level of abstraction, introducing and justifying its design choices along the way. The overall algorithm is presented in a comprehensive manner by subdividing its description into three main stages: (i) *graph generation*; (ii) *Voronoi partitioning*; (iii) *Lloyd's relaxation*.

4.1 Working on a graph

As exposed in the previous chapter, there are several ways to do CVT, with relative advantages and drawbacks. In particular, the use of Euclidean distance in RVD-based methods brings a series of problems we would like to avoid altogether with our method. Indeed, we opted for a geodesic approach for the construction of the VD. This way, centroids stay on the surface all the time, and regions are guaranteed to be contiguous.

To compute GVD, we chose a discrete clustering approach, which partitions a graph of the topological elements of the mesh. The nodes of the graph relate to such elements, either vertices or triangles, and the neighbourhood relationship of nodes, modeled by arches, reflects the one of the connected elements in the topology.

Seeds are a subset of nodes in this graph, and both partitioning and relaxation are reduced to relatively simple operations on said graph. These design choices convey simplicity to the algorithm, and guarantee seeds to always relate to an element of the mesh.

To assist the tessellation procedures further described, there are two major metrics of the graph's structure that need to be mentioned: the *cost* of arches and the *weight* of nodes. These two metrics are not interchangeable in terminology and fill distinct roles, as later explained in the next paragraphs.

4.1.1 Geodesics and graph's connectivity

Geodesics are approximated by the cost of the arches of the graph. The cost of an arch is, normally, the approximation of the geodesic distance between two neighbouring nodes. For instance, taken two nodes that refer to vertices, it is the length of the edge which connects the two vertices. In the case of two triangles, it is the geodesic distance between the barycenters of such triangles. Therefore, the approximation of the geodesic distance between two mesh elements is the sum of the costs of the arches composing the shortest path that links the nodes corresponding to such mesh elements.

It follows that the precision of this approximation depends on how the topological elements are connected with one another. Since the neighbourhood relationship of nodes normally reflects the adjacency of mesh elements, the shortest path from one node to another may not be as direct as it normally would be, but it is routed along the arches of the graph, often resulting in abrupt, jagged paths. To reduce the impact of this problem, nodes' connectivity can be enriched. Therefore, the precision of the approximation also depends on the connectivity of the graph: the more the graph is connected, the less the error introduced by discretization is.

Given an input (two-manifold) mesh, the kinds of graph that can be computed to support our algorithm are:

Vertex-based graph Nodes refer to vertices only. Neighbourhood of nodes reflects the vertex adjacency found in the mesh. The cost of an arch is the distance between vertices (therefore, the length of the edge connecting two vertices).

Dense vertex-based graph Same as the vertex-based graph, with the addition of arches connecting opposite vertices belonging to different triangles, such that the triangles share one edge and the vertices do not lie on said edge. See Fig. 4.1 for clarifications. The cost of these arches is the geodesic distance of the opposite vertices.

Triangle-based Nodes refer to triangles only. Neighbourhood of nodes reflects the per-edge triangle adjacency found in the mesh. The cost of an arch is the geodesic distance between the barycenters of the triangles.

Dense triangle-based graph Same as the triangle-based graph, with the additional per-vertex triangle adjacency. The cost of the additional arches is the sum of the distances between the barycenters of the triangles and the common vertex.

Mixed graph Nodes refer to both vertices and triangles. Neighbourhood of nodes includes, but it is not relegated to, the vertex adjacency and the per-edge triangle adjacency found in the mesh. A node referring to a triangle is also connected to the nodes referring to its vertices, and vice versa. The cost of these arches is the distance between the barycenter of the triangle and the vertex.

By offering several options for the graph type, we let the user choose the precision of the geodesics approximations. Our experiments show that more connected graphs, such as the dense variants and the mixed graph, reduce the error considerably.

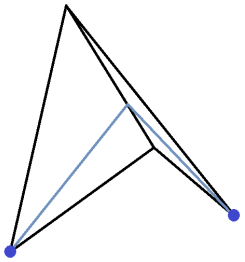

Figure 4.1: Opposite vertices, in blue, of two non coplanar triangles. The geodesic distance of the vertices over the triangles is marked by the light blue segment. This distance is then assigned to the cost of the arch connecting the opposite vertices.

4.1.2 Weighting by areas

The information of the area of surfaces is fundamental for the relaxation procedure. Indeed, a possible interpretation of the ρ density function, introduced in subsection 2.1.1, may be an estimate of the area around the point x we are evaluating.

This estimate is translated into the graph structure in the form of nodes' weight. When a node refers to a triangle, its weight is simply the area of the triangle it is referring to. Otherwise, in case the node refers to a vertex, its weight is an estimation of the vertex's surrounding area, computed as one third of the sum of the areas of the neighbouring triangles.

4.1.3 Initial seed selection

Alongside the proper graph generation, there's also the selection of nodes that operate as the seeds for computing the GVD. In our algorithm, these are randomly picked among the nodes of the whole graph.

Yet, as long as seeds remain nodes of the graph, nothing excludes to replace this method with a more advance one, like the one used by Peyre and Cohen [12], as the choice of the seed selection method does not preclude any further procedure.

4.1.4 Anisotropic metrics

As mentioned before, our work supports both isotropic and anisotropic tessellation. For the latter, the algorithm requires additional input data in the form of a *tangent field* $T = \{s_i\}_{i=1}^{n}$, where s_i is the *tangent space* local to the i-th vertex of a mesh of n vertices.

Given the normal vector $\mathbf{n}_i$ of the surface local to s_i, a tangent space s_i is defined as a couple of vectors $(\mathbf{t}_i, \mathbf{b}_i)$, a tangent and a bitangent respectively, such that $\mathbf{t}_i \perp \mathbf{b}_i \wedge \mathbf{t}_i \perp \mathbf{n}_i \wedge \mathbf{b}_i \perp \mathbf{n}_i$.

The magnitudes of $\mathbf{t}_i$ and $\mathbf{b}_i$ are intended to influence the costs of the arches. Given a field T, the cost of an arch is the average of the tangent spaces of its nodes. If a node refers to a vertex of index i, then s_i is taken as-is. Otherwise, if said node refers to a triangle defined on vertices of indices i, j and k, then its tangent space is the average of s_i, s_j and s_k. For details on how to average tangent spaces, go to subsection 4.1.5.

Given an arch of index e, that links two nodes $n_{e,0}$ and $n_{e,1}$, and its tangent space $s_e = (\mathbf{t}_e, \mathbf{b}_e)$, its cost c_e is computed as

$$c_e = \sqrt{(\mathbf{t}_e \cdot \mathbf{d}_e)^2 + (\mathbf{b}_e \cdot \mathbf{d}_e)^2},$$

in which $\mathbf{d}_e$ is defined as

$$\mathbf{d}_e = \frac{p(n_{e,1}) - p(n_{e,0})}{\| p(n_{e,1}) - p(n_{e,0}) \|} d(n_{e,0}, n_{e,1}),$$

where p returns the point in $\mathbb{R}^3$ the node relates to, and $d(n_{e,0}, n_{e,1})$ is the geodesic distance between the elements associated to $n_{e,0}$ and $n_{e,1}$.

The use of $\mathbf{d}_e$ guarantees the computation of c_e to be a simple vector calculation even in the case of the elements associated to $n_{e,0}$ and $n_{e,1}$ not being coplanar (for instance, when the two nodes refer to triangles).

4.1.5 Averaging tangent spaces

When averaging two or more tangent spaces, directly computing the arithmetic mean of the tangent and bitangent vectors is not ideal. Look at the couple of tangents, $\mathbf{t}_1$ and $\mathbf{t}_2$, pointing in opposite directions in Fig. 4.2.i. Averaging those may compromise, or nullify, the direction of the final tangent.

Moreover, averaging a tangent with another (or a bitangent with another) may not be reasonable since there is no guarantee the two vectors go along the same local axis. In cases like the one in Fig. 4.2.ii, it would be better to mix $\mathbf{b}_1$ with $\mathbf{t}_2$ as they both contribute the most along a common local axis.

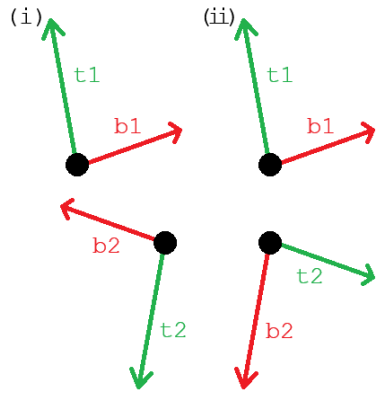

Figure 4.2: Some arrangements of tangent spaces are problematic for averaging: (i) vectors that point in opposite directions; (ii) vectors aligned to different local axes.

To overcome these issues, we address the following: (i) a tangent (or bitangent) has the same effect on costs computation as its own opposite vector; (ii) a tangent and a bitangent can be averaged, as long as they are aligned towards a common direction. These statements are verifiable by looking at the computation of c_e in subsection 4.1.4.

Assume to have two tangent spaces $s_1 = (\mathbf{t}_1, \mathbf{b}_1)$ and $s_2 = (\mathbf{t}_2, \mathbf{b}_2)$, and that both are either right-handed or left-handed. To average the two spaces, we need to find $s_2' = (\mathbf{t}_2', \mathbf{b}_2')$, an equivalent space to s_2, so that the directions of its vectors are as close as possible to the ones of s_1. s_2' is obtainable by flipping the direction of $\mathbf{t}_2$ and $\mathbf{b}_2$, and assigning these modified vectors to $\mathbf{t}_2'$ and $\mathbf{b}_2'$ while preserving the handedness of s_2.

By following this line of reasoning, we have four possible configurations of s_2': $(\mathbf{t}_2, \mathbf{b}_2)$; $(\mathbf{b}_2, -\mathbf{t}_2)$; $(-\mathbf{t}_2, -\mathbf{b}_2)$; $(-\mathbf{b}_2, \mathbf{t}_2)$. The configuration that maximizes $\mathbf{t}_1 \cdot \mathbf{t}_2' + \mathbf{b}_1 \cdot \mathbf{b}_2'$ is then picked to be averaged with s_1.

At the moment, this method is limited to averaging two spaces only. If we want to find the tangent space related to a triangle, we have to average the spaces of the first two vertices, and then to average this mean with the space of the third vertex. This may not be desirable, and it is an feature to be further improved.

4.2 Voronoi partitioning

Given a graph with seeds, partitioning into a VD is made by using a modified parallel Dijkstra's algorithm, similar to the one found in Erwig's work [7]. In this variant of Dijkstra's algorithm, there are multiple sources from which the search is made. In our case, these sources are the seeds, and there is no destination node to be found. Indeed, our goal is to define regions by setting nodes' membership to a specific seed, while exploring the graph using Dijkstra.

The algorithm is initialized by putting the seeds into the frontier. Then, the loop begins by picking from the frontier the node with the lowest distance to its seed and expanding it.

When a node is expanded, the unvisited neighbouring nodes are marked to belong to the seed the expanded node belongs to. When the neighbouring nodes of an expanded node have already been visited, then the usual Dijkstra's test takes place, with the difference that seed membership would change to the one of the expanded node if the test passed.

By exploring the graph with Dijkstra, it is ensured that nodes refer to the seed closest to them, while also making, for each region, the tree of the shortest paths from the nodes to their assigned seed.

The tree is defined with nodes referring to their *parent* node. The root of a tree is a seed, which has no parent. Parallel Dijkstra assigns the parent to the nodes it processes.

The parallel Dijkstra algorithm that runs during the relaxation can be told to enable *region lock*. When the latter is active, the trees are still computed, but without setting the membership of nodes to other seeds. Basically, when regions are locked, the membership of nodes to specific seeds is preserved from one iteration to the next. This allows to update the trees while the seeds are moved within a region without altering its bounds.

4.3 Lloyd's relaxation

Relaxation of seeds is primarily inspired by Lloyd's algorithm. We opted for a revision of Lloyd's algorithm over BFGS to adhere to our goal of keeping the algorithm conceptually simple and almost entirely characterized by graph operations.

The *relaxation loop* begins by running parallel Dijkstra on the current seeds. Then, seeds are moved into a more optimized configuration. The loop repeats until convergence. After the loop, Dijkstra may be ran a last time to partition the graph with the optimized seeds.

Moving a seed means to promote a node as seed while demoting another. This promotion is driven by a heuristic that computes whether the change is convenient for minimizing the energy function or not.

To compute said heuristics, some preliminary work is needed. After partitioning the graph, the nodes are enriched with the information of the total costs and weights of the subtrees, rooted at the nodes themselves.

We also need to attach a *score* to nodes, indicating a measurement of how well they minimize the energy function while being seeds. Generally, when a score is computed for a node, it is kept during the relaxation process. This permits us to compare nodes based on how good they were as seeds.

Scores can also be used to determine if a node was a seed in the previous iteration of the relaxation. Indeed, if scores are initialized with a negative value, we can tell if a node was a seed by looking for its non-negative score.

Our relaxation procedure comprises two consecutive phases: a *greedy phase* and a *precise phase*. The greedy phase is responsible for moving seeds by a major amount in an optimistic manner, while the precise phase takes care of the final adjustments of the seed configuration. It follows the pseudocode of the overall relaxation process:

```
Require: seeds ⊂ graph.nodes
  function RelaxSeeds(graph, seeds)
      phase ← greedy
      lockRegions ← false
      ResetScores(graph.nodes, −1)
      while phase ≠ finished do                    ▷ The relaxation loop.
          if phase = greedy then
              GreedyStep(phase, graph, seeds)
          else if phase = precise then
              PreciseStep(phase, lockRegions, graph, seeds)
      ParallelDijkstra(phase, false, graph, seeds)
```

The next paragraphs will describe the details of the phases. Any repetition found in the next pseudocode listings is intentionally there for clarity's sake.

4.3.1 Greedy phase

The greedy phase has the goal of performing the major changes in the seeds' configuration throughout the whole algorithm. While being in this phase, the relaxation loop begins with the partition of the graph. Regions are unlocked during the whole greedy phase. What follows is the greedy scoring and the greedy movement of seeds. The greedy phase converges when no seed can be moved any longer. A pseudocode for one step of the greedy phase:

```
function GreedyStep(phase, graph, seeds)
    ParallelDijkstra(phase, false, graph, seeds)
    SubtreeCostsAndWeights(graph)
    ScoreSeedsGreedy(seeds)
    MoveSeedsGreedy(seeds)
    if no seed was moved then
        phase ← precise
```

When the greedy phase is concerned, we use scores solely for determining if a node was a seed in a previous iteration of the relaxation. Therefore, the ScoreSeedsGreedy function limits itself to assigning a positive value, such as 1, to all seeds.

When moving a seed greedily, the neighbour that maximizes the heuristic value is selected as the most fitting candidate. If such candidate was a seed in a previous iteration, then it is discarded, and the seed is not moved at all. The greedy movement and the heuristic calculation are summarized as follows:

```
function ComputeHeuristic(node, seed)
    heuristic ← node.subtreeCost * node.subtreeWeight
    for each n ∈ node.neighbours do
        if n.parent = seed then
            heuristic ← heuristic + n.subtreeCost * n.subtreeWeight
    return heuristic

function MoveSeedsGreedy(seeds)
    for each seed ∈ seeds do
        maxHeuristic ← −∞
        candidate ← seed
        for each neighbour ∈ seed.neighbours do
            if neighbour ∉ seeds ∧ neighbour.regionId = seed.regionId then
                heuristic ← ComputeHeuristic(neighbour, seed)
                if heuristic > maxHeuristic then
                    maxHeuristic ← heuristic
                    candidate ← neighbour
        if candidate.score < 0 then          ▷ If candidate was not a seed...
            seed ← candidate
```

The approach of impeding seeds' movement, when the best pick was already taken before, prevents seeds from being moved indefinitely along a loop in the graph (as demonstrated by our experiments), and provides a tangible way to make the phase converge. Fig. 4.3 shows an example of how the greedy relaxation advances until convergence.

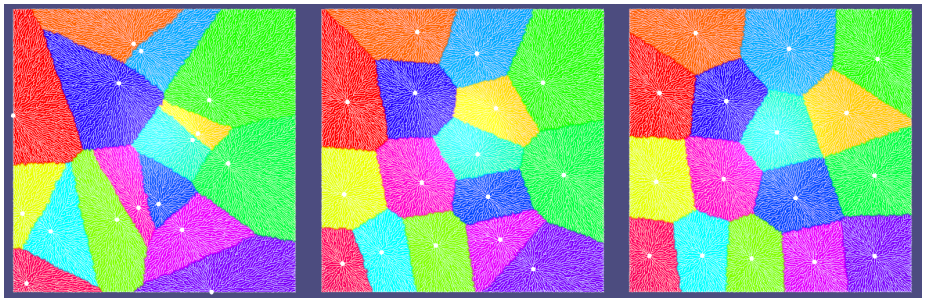

Figure 4.3: Advancement of the greedy phase running on a vertex-based graph. From left to right: initial configuration; relaxation; convergence. Seeds are the white dots, trees' branches are the white lines, while the colored areas are the various regions.

Conceptually, the heuristic states how much the node's subtree extends itself along the region. A subtree with a high (weighted) total cost is, generally, distributed over a large portion of the region. Therefore, moving the seed along this direction makes it closer to the center.

Our first approach to the heuristic was to define it as the weighted cost of the subtree of the neighbour only. This heuristic, though, would not make seeds converge into centroids most of the time. In situations such as the one displayed in Fig. 4.4, the seed would be attracted more towards the longitudinal features of the region, along which subtrees generally have higher costs, than towards the middle.

By also summing up the weighted costs of those candidate's neighbours whose parent is the current seed, we could verify by our experiments that the greedy phase converged into more desirable configurations.

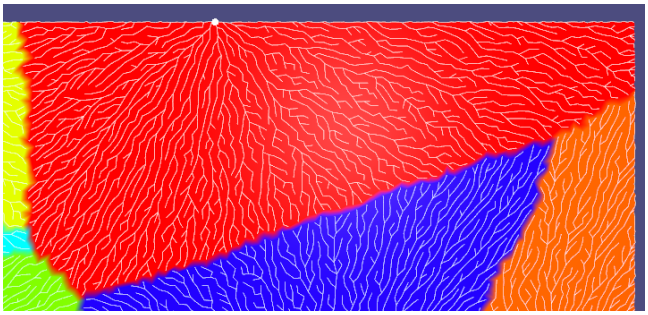

Figure 4.4: A highlight of a region (red) from a vertex-based graph, with its seed (white dot) and tree of shortest paths (white lines).

4.3.2 Precise phase

The precise phase has the goal of performing the final adjustments to the seeds' configuration. The phase can be seen as subdivided into *macrosteps*, which are comprised of *microsteps*.

At the beginning of each macrostep, at which regions are still unlocked, scores are reset to a negative value, and the partition of the graph is made. After that, the regions are locked and kept as-is for the rest of the macrostep. The first microstep can now be performed.

A single microstep is the equivalent of running a precise phase step while regions are locked. When all seeds converge into the centroids of their regions, these are unlocked, marking the beginning of another macrostep.

The whole relaxation process converges when the seeds' configurations at the beginning and end of the macrostep match. An overall step for the precise phase can be described as the following:

```
function PreciseStep(phase, lockRegions, graph, seeds)
    ParallelDijkstra(phase, lockRegions, graph, seeds)
    SubtreeCostsAndWeights(graph)
    if lockRegions = false then        ▷ If it's the beginning of a macrostep...
        ResetScores(graph.nodes, −1)
        prevSeeds ← seeds
        lockRegions ← true
    ScoreSeedsPrecise(graph, seeds)
    MoveSeedsPrecise(seeds)
    if no seed was moved then
        lockRegions ← false
        if prevSeeds = seeds then
            phase ← finished
        else
            prevSeeds ← seeds
```

Fig. 4.5 shows an example of the execution of a macrostep. It is noticeable how little the seeds' configuration changes from start to finish. Yet, a series of macrosteps can greatly improve the quality of the final tessellation.

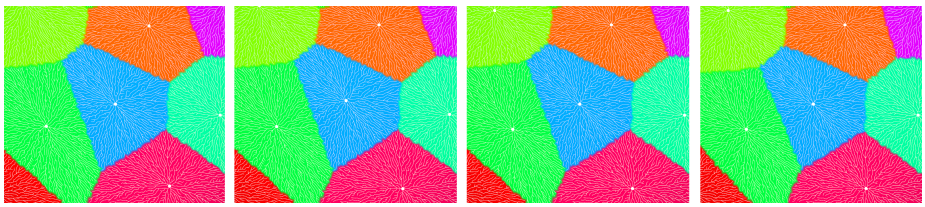

Figure 4.5: Advancement of a macrostep. From left to right: beginning of macrostep; seeds are moved within the bounds of the region; seeds converge into centroids; partitioning with unlocked regions.

Given a region, the score of its seed is the sum of the weighted, squared distances from the seed of all region's nodes. It is the contribution of a single seed of the function $F(X)$ introduced in subsection 2.1.1. Since we aim to minimize $F(X)$, finding a node that minimizes the score is a key aspect of selecting the best candidate. It follows the pseudocode of precise scoring:

```
function ScoreSeedsPrecise(graph, seeds)
    nodesCount ← new array of size seeds.size
    set nodesCount's elements to 0
    set seeds' scores to 0
    for each node ∈ graph.nodes do
        node.seed.score ← node.seed.score + node.distance² * node.weight
        region ← node.seed.regionId
        nodesCount[region] ← nodesCount[region] + 1
    for each seed ∈ seeds do
        seed.score ← seed.score/nodesCount[seed.regionId]
```

The precise movement is, essentially, an extension of the greedy one. The selection of the most fitting candidate is driven by maximizing the same heuristic. The key difference here is that if, among the neighbours of the seed, there's a node with a better score than the current seed's, then the seed is moved back to that node. This is a fallback mechanism that reverts the last movement if it ends up being inconvenient. When the current seed is surrounded by former seeds with worse scores, then the seed is left as-is and it is considered as the centroid of the locked region. The precise movement function is summarized as follows:

```
function MoveSeedsPrecise(seeds)
    for each seed ∈ seeds do
        minScore ← seed.score
        maxHeuristic ← −∞
        candidate ← seed
        for each neighbour ∈ seed.neighbours do
            if neighbour ∉ seeds ∧ neighbour.regionId = seed.regionId then
                if neighbour.score ≥ 0 ∧ neighbour.score < minScore then
                    minScore ← neighbour.score
                    candidate ← neighbour
                else if minScore = seed.score then
                    heuristic ← ComputeHeuristic(neighbour, seed)
                    if heuristic > maxHeuristic then
                        maxHeuristic ← heuristic
                        candidate ← neighbour
        if minScore ≠ seed.score ∨ maxHeuristic ≠ −∞ then
            seed ← candidate
```

4.3.3 Conservative heuristics

An important property of heuristics is that they often give conservative results. This is due to the fact that the evaluation is done by assuming the trees remain the same after changing the seed.

Clearly, this is not true. Changing the seed modifies the tree, especially in the levels close to the root. This change of trees often means that an optimization of paths has been made, the latter being a further gain of weighted costs not originally included by the heuristic.

A positive outcome from this property would be that when a candidate node is considered the most convenient among others on a local scope, it surely is. The drawback is that an estimation that would otherwise be the most attractive could be overlooked.

Chapter 5

Implementation

The application made for this thesis was realized primarily in C++ and *libigl*, a mesh processing library. The latter features a customizable OpenGL renderer, which was used to speed up the development of this project. The source code of the application is available at *https://github.com/rapfamily4/DiscretizedLloyd*.

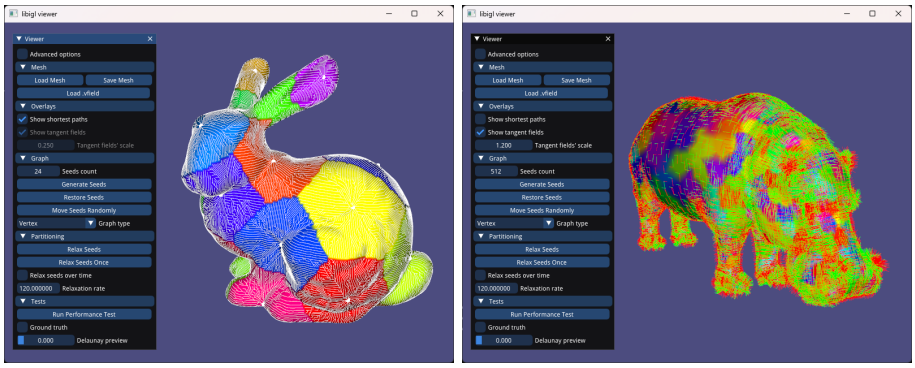

Figure 5.1: User interface of the application. On the left, trees are highlighted over the model in white. On the right, the model's tangent field is plotted on the screen.

The following sections are focused on the most remarkable aspects of the implementation of the algorithm.

5.1 Tangent field file

Tangent fields must be read from `.vfield` files. These files define the tangent space for each vertex of the mesh. The spaces' vectors are defined by providing magnitudes and directions separately. This way of storing vectors implies less normalization in the mixing of tangent spaces, which improves performance.

5.2 Structs and classes overview

The main structs useful for further discussion are:

Node It keeps track of the connected arches, as well as the index of the region it belongs to, its weight, its distance from the seed, the index to the parent, and the total cost and weight of the subtree rooted at the node itself.

Arch It holds its cost and the targeted node.

The most relevant classes found in this project are:

Model It holds the vertex and triangle matrices of the model, as well as the tangent field. It defines methods for the distances between mesh elements and for adjacency lists. These functions are fundamental for the graph generation. It also provides distances computation with *Manhattan* and *L-infinity*, alongside geodesics.

TangentSpace It models a tangent space of the tangent field. It holds the tangent and bitangent directions, alongside their magnitudes. It provides methods for averaging spaces, and for altering the cost of arches given the proper distance function.

DijkstraPartitioner It holds the graph structure, defined as a vector of nodes. It also keeps the current configuration of seeds, and defines all the methods useful for partitioning and relaxation.

Application It manages the overall advancement of the application. It generates the graph by calling the methods from `Model`, and passes it to `DijkstraPartitioner`. It also wraps the libigl viewer instance and the user interface code.

5.3 Subtree costs and weights

As discussed in section 4.3, subtree total costs and weights are fundamental for the relaxation process. The computation of those, which happens just after parallel Dijkstra, requires to sort nodes in non-descending order with respect to the distance from their seed. This is primarily due to the fact that nodes only keep the index to the parent, when the structure of the tree is concerned.

Instead of sorting the vector of node structs, which would be heavy performance-wise, we sort a separate vector, `sorted`, that contains the indices to the elements in the vector of structs.

After that, nodes' subtree total costs and weights are set to the nodes' distance and weight respectively. Then, nodes are processed from the most distant to the least. A parent node will receive the distance and weight from its child, summing those to its own subtree total costs and weights respectively. Seeds are skipped entirely, since they lack any parent (their parent index is -1).

This is the related code found in the project at the time of writing:

```
void sortNodeIdsByDistance() {
    std::sort(sorted.begin(), sorted.end(), [this] (int i, int j) {
            return nodes[i].distFromSeed < nodes[j].distFromSeed;
        });
}

void updateSubtreeInfo() {
    for (Node& n : nodes) {
        n.subtreeWeight = n.weight;
        n.subtreeCost = n.distFromSeed;
    }

    for (int i = sorted.size() - 1; i >= 0; i--) {
        int j = sorted[i];
        int parent = nodes[j].parentId;
        if (parent == -1) continue;
        nodes[parent].subtreeWeight += nodes[j].subtreeWeight;
        nodes[parent].subtreeCost += nodes[j].subtreeCost;
    }
}
```

Calling these two functions in succession is equivalent to SubtreeCostsAndWeights in the pseudocode listings in section 4.3.

5.4 Priority heap

The frontier used in parallel Dijkstra is implemented in `DijkstraPartitioner` as a priority heap. Its data is structured in a vector containing the indices to the nodes. The top of the heap holds the index to the node with the lowest distance to its seed. Functions are provided for popping the top element, and for pushing, sinking and floating indices.

The peculiarity of this heap resides in how it handles nodes changing their distance to the seed when they pass the Dijkstra's test. Each node holds an index to their entry in the frontier, called `frontierIndex`. When the Dijkstra's test passes for a node, the frontier floats the element in the heap at index `frontierIndex`. When floating and sinking occurs, the `frontierIndex` of the swapped nodes is always validated, so that it is always coherent with how elements are arranged in the frontier.

5.5 Precise phase optimizations

Some optimizations to the precise relaxation can be introduced to speed up computational times. In our work, there are mainly two optimization that improve performance considerably when used together.

The first optimization prevents running parallel Dijkstra for an already converged seed, before the whole macrostep has ended. At the beginning of a microstep, `prevSeeds` preserves the previous configuration of seeds. When initializing parallel Dijkstra, we compare `prevSeeds` with the current configuration. If a region has not changed its seed, then the related node is not put in the frontier. This stops from running parallel Dijkstra for a region whose seed cannot be moved anymore.

The second improvement prevents running precise movement routines for the seeds of those regions that have not changed between macrosteps. We need to keep track of the score of each seed at the beginning of both the current and previous macrosteps. If a seed has an identical score between macrosteps, then it is very likely its region has not changed its nodes; therefore, the seed is already a centroid and does not require to be moved.

For the latter optimization, it would be safer to check if regions have stayed identical by comparing the configuration of nodes between macrosteps. Indeed, nothing excludes that many configurations of nodes for a region may return the same score. In practice, though, this is very unlikely, and our experiments show that this optimization just improves execution times without affecting the final result.

Chapter 6

Results

The following sections explore the empirical findings of the algorithm in detail. The quality of the tessellation and the execution times are analyzed.

6.1 Tessellation evaluation

The algorithm is capable of generating a competent CVT over the mesh's surface. When enough seeds are relaxed, these furnish a solid starting point for triangulation algorithms, such as Delaunay.

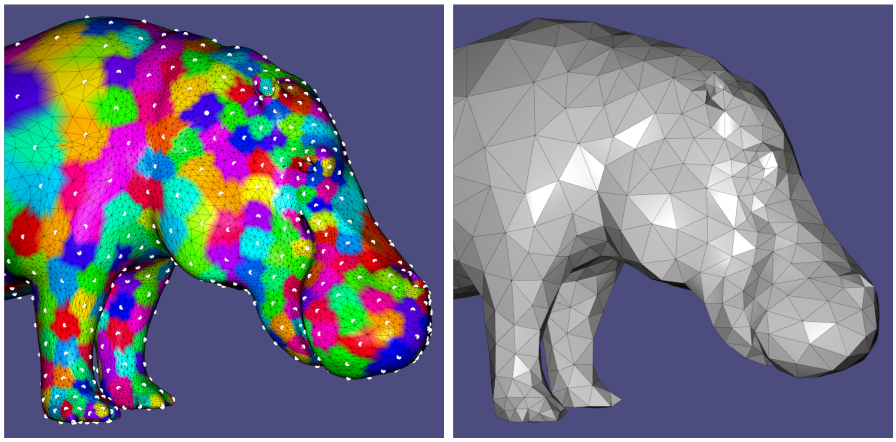

Figure 6.1: An example of isotropic tessellation using a vertex-based graph. On the left, the VD. On the right, the Delaunay triangulation preview.

The algorithm correctly handles anisotropic metrics. The tangent field influences how the CVT converges, as shown in Fig. 6.2.

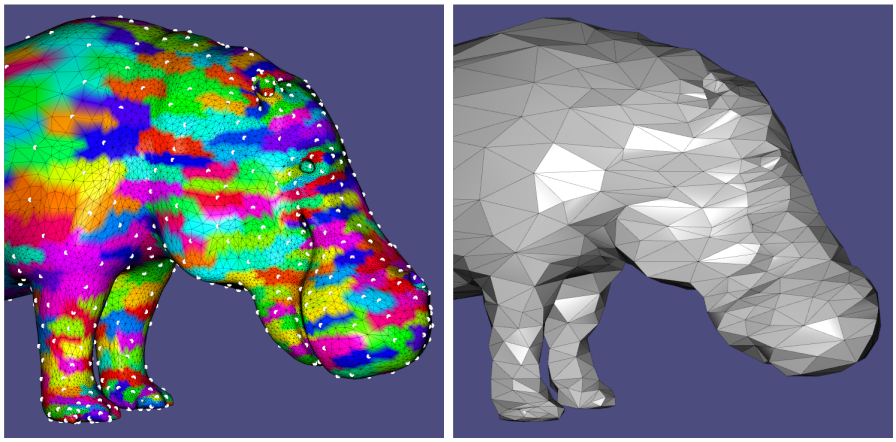

Figure 6.2: An example of anisotropic tessellation using a vertex-based graph. The tangent field taxes penalizes arches going along the Y axis. On the left, the VD. On the right, the Delaunay triangulation preview.

The weighting of the graph's arches by the local areas ensures a more regular morphology to regions even when the topology presents triangles of irregular size. In cases in which regions cover the surface extensively, like the one shown in Fig. 6.3, they are seemingly unaffected by the irregularity of the mesh, and their seeds are close enough to where the centroid would be.

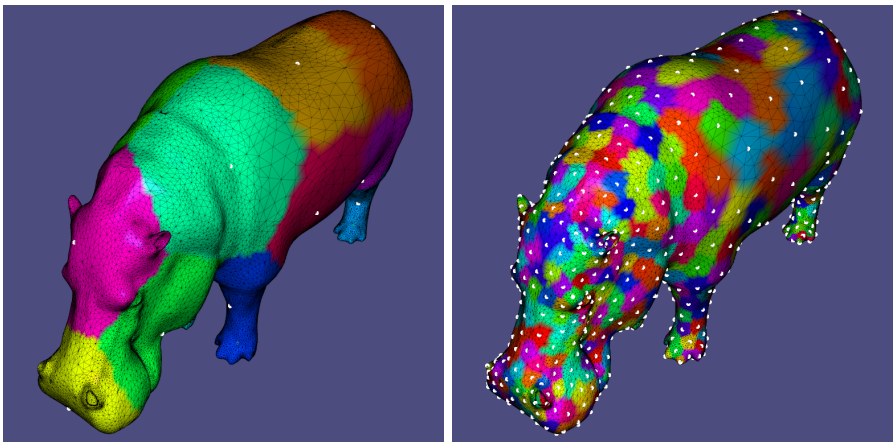

Figure 6.3: Regions are kept regularly shaped even if underlying topology is irregular. When the seed count in high, seeds tend to concentrate themselves where the topology is denser.

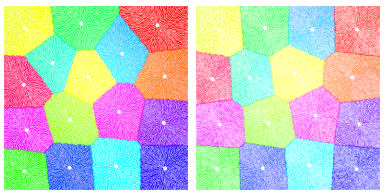

Figure 6.4: CVTs obtained with the same seeds. On the left, a vertex-based graph was used. On the right, a mixed graph was used.

Apart from any irregularity of the topology, the quality of the tessellation may suffer from how coarse the mesh can be. As suggested in subsection 4.1.1, since the algorithm basically computes discrete clusters over a graph, increasing the graph's connectivity enhances the precision of the partition and the relaxation (as seen in Fig. 6.4). More connections allow a seed to get closer to the actual centroid of the respective region, for instance.

The quality of tessellation can be verified by plotting the *ground truth* on screen, that being the discrete partition of nodes based on their Euclidean distance from seeds. The actual centroids (not the seeds) are displayed in black. When the ground truth is computed for a plane mesh, it can be a good reference for an optimal relaxation. An example is displayed next.

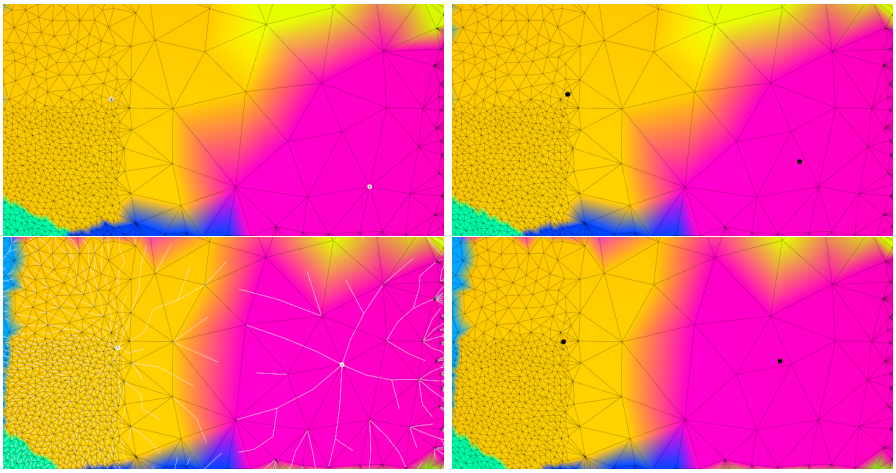

Figure 6.5: At the top, a vertex-based partition with its ground truth. At the bottom, a mixed partition, alongside the ground truth.

Comparing the two images at the top of Fig. 6.5, in which the graph used is vertex-based, the seed in the purple area is visibly far from the centroid of the ground truth. At the bottom, in which a mixed graph was partitioned, the seed of the purple area is very close to the black centroid. The latter is an indicator of a good quality tessellation. Another sign of good quality is when the regions from the CVT and the ground truth match (see Fig. 6.6).

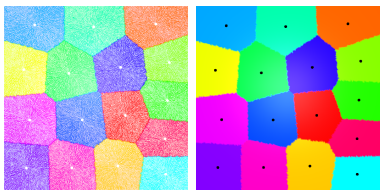

Figure 6.6: The regions of the CVT and its ground truth greatly match. The CVT was made with a mixed graph.

6.2 Performance evaluation

The tests are executed on a machine with the following specification:

CPU AMD Ryzen 7 3700X, 3600 Mhz

RAM 16 GB, DDR4, 3200 Mhz

The execution time was measured in milliseconds. We ran the algorithm in *batches*, where each batch runs the algorithm several times with a different parameter configuration. The list of parameters follows:

Graph type The type of graph used, which can be one of the following: vertex-based; dense vertex-based; triangle-based; dense triangle-based; mixed.

Greedy relaxation mode It states in which mode the greedy relaxation is executed. The available modes are: disabled; standard; extended. The last two refer to using the original heuristic and the improved one respectively, introduced in subsection 4.3.1.

Macrostep optimization It toggles the second optimization discussed in section 5.5.

Microstep optimization It toggles the first optimization discussed in section 5.5.

The seed configuration is randomly generated at the beginning of each batch, and restored to its initial state at the beginning of a new execution of the algorithm[1].

The next tests do not focus on the generation of the graph, which has demonstrated to require a negligible amount of time (relative to the whole execution) during our experiments. For clarity's sake, it usually takes up to 10ms with a plane mesh of 45 thousand triangles on the machine used for testing.

The details on the meshes used in this section:

Model	Vertices	Edges	Triangles
Plane	23,107	68,106	45,000
Hippo (hi-res)	706,244	2,118,692	1,412,448

6.2.1 Plane mesh test - 16 seeds

Our first tests consists of running 10 batches, with 16 seeds, on a plane mesh of roughly 23 thousand vertices and 45 thousand triangles. We obtained the average execution times plotted in the parallel coordinates chart in Fig. 6.7.

[1]The starting seeds may not be exactly the same from one execution to the other due to the graph type changing. When the graph type changes, seeds can be arranged so that the new configuration is spatially similar to the previous one.

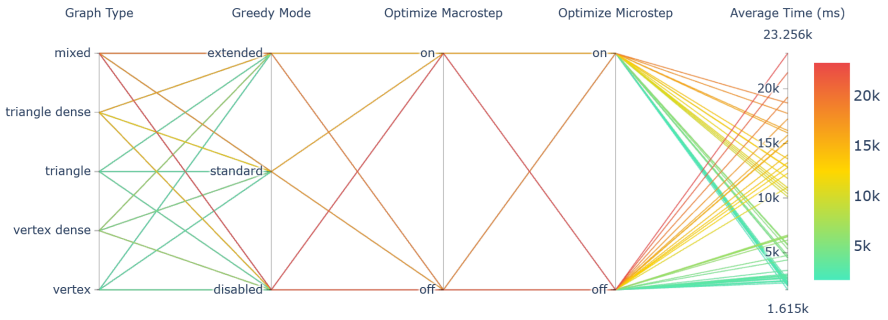

Figure 6.7: The color scale refers to the average time. Time increases with dense graphs.

We can already observe that running time drastically increases when dense graphs are being used. The best results are, in fact, returned when the algorithm is run over vertex-based and triangle-based graphs. Mixed and dense triangle-based graphs seem to have comparable performances, with an average running time between 10 and 20 seconds. The other types of graph yield times all below the 6 seconds, with the vertex-based graphs performing the best.

Fig. 6.8 shows how execution time improves when optimizations are turned on. The impact of optimizations is observable in each graph type.

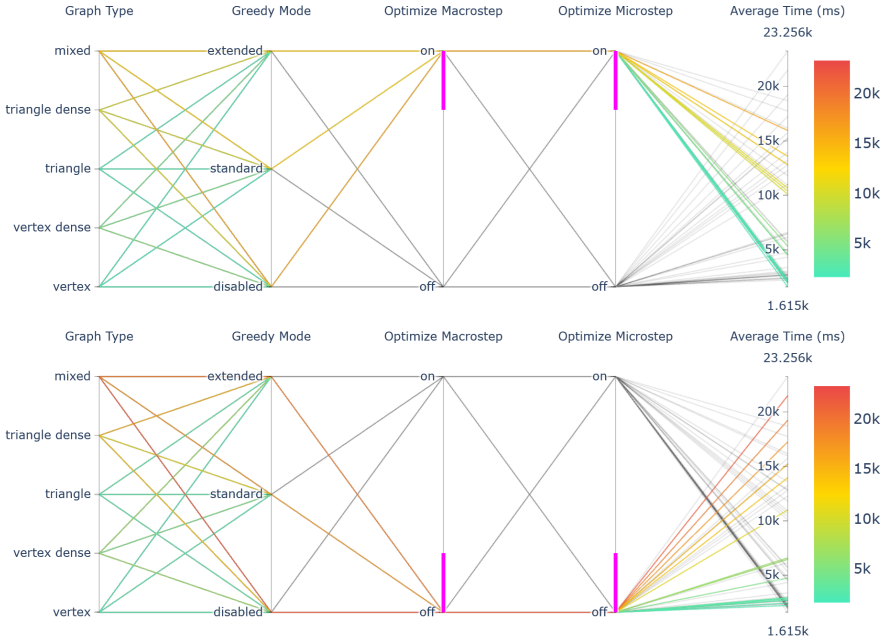

Figure 6.8: On the top, optimizations are enabled. On the bottom, they are disabled.

Another observation concerns the mode the greedy phase is set on. As shown in the charts of Fig. 6.9, disabling the greedy phase leads to worse times, as expected. Yet, the standard mode beats the extended one. Such is an unforeseen result, as the extended mode usually converges in a more ideal seed configuration compared to the standard mode.

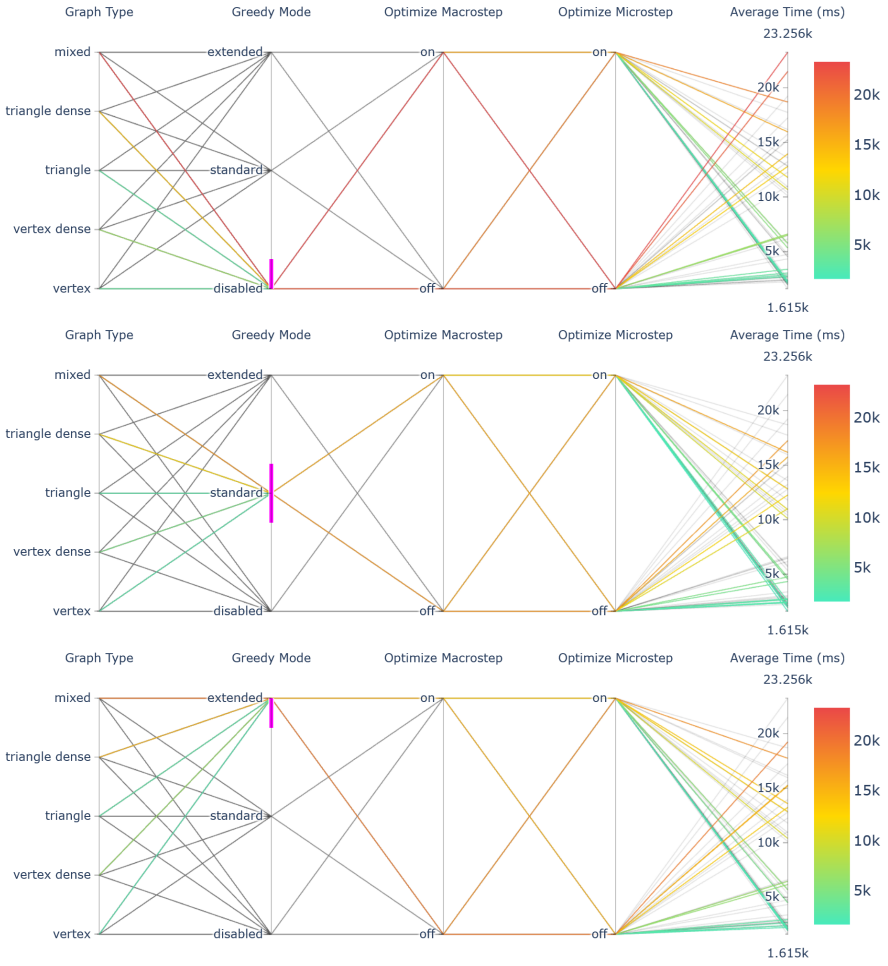

Figure 6.9: From top to bottom: disabled greedy phase; standard greedy phase; extended greedy phase.

The average number of greedy steps is similar between modes (see Fig. 6.10). When investigating this result further, it turned out that extended greedy takes slightly more time to process on average. This inevitably impacts performance.

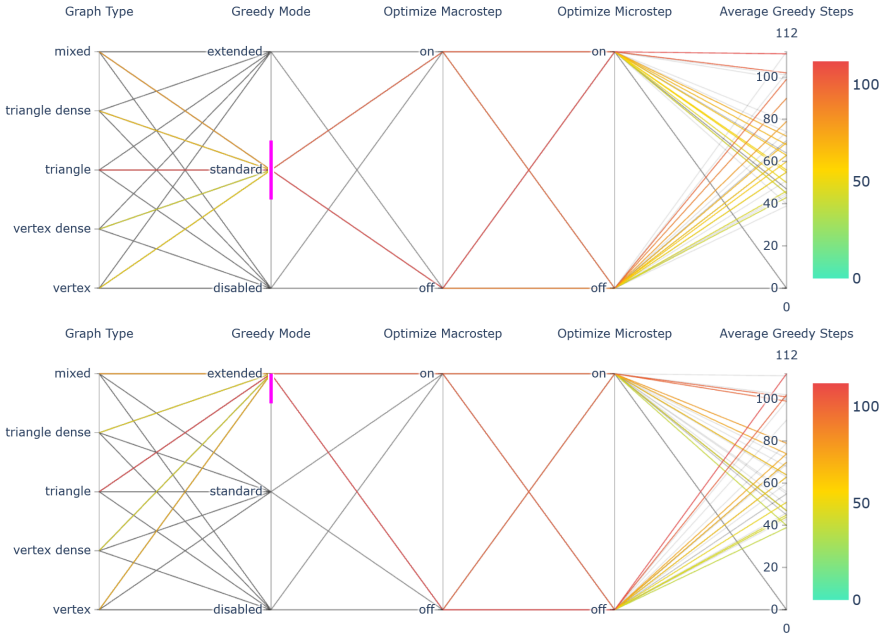

Figure 6.10: The color scale refers to the average number of greedy steps. Both standard and extended greedy have a similar count.

6.2.2 Plane mesh test - 1024 seeds

The second test was done on the same plane model, running 10 batches with 1024 seeds. The average time is plotted in Fig. 6.11. Every aspect discussed in the previous test case is also valid here, but there is an important observation to be made while comparing the results between Fig. 6.7 and 6.11.

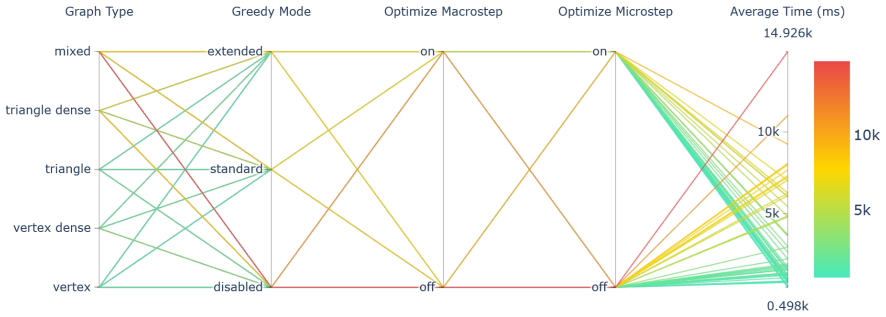

Figure 6.11: The color scale refers to the average running time.

While running the algorithm on the same model, if regions get small enough with a sufficiently high seed count, performance tends to improve.

Small regions may lead to better convergence time due to the stricter number of nodes for each region. Less nodes per region would lighten parallel Dijkstra calculations, when optimizations are enabled.

Despite such result, it is not guaranteed that increasing the seed count always improves performance, as documented in the next test report.

6.2.3 Hippo (hi-res) mesh test

The third test consists of a series of runs over a high resolution model with 706 thousand vertices and roughly 1.5 million triangles. The algorithm was run with a simple vertex-based graph, enabling the extended greedy mode and all optimizations. The seed count is doubled at each execution.

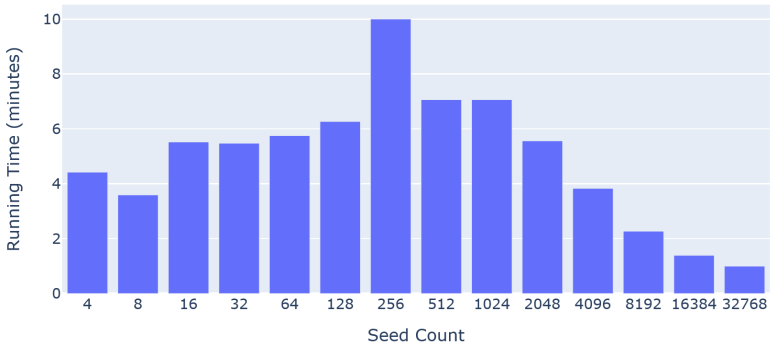

Figure 6.12: Running time as a function of the seed count.

As reported by Fig. 6.12, running time is higher than in previous tests, due to a higher polygon count. A more complex topology translates to a graph with more nodes and arches.

We can also observe that increasing the seed count does not necessarily give better performance, but when the seed count surpasses a certain threshold (i.e. when regions are sufficiently small), then running time is seen to decrease consistently. As speculated in subsection 6.2.2, small regions may lighten parallel Dijkstra calculations, when optimizations are enabled.

This phenomenon is further documented by the statistics shown in Fig. 6.13. We can see how iterations tend to diminish when the seed count is the highest. Surprisingly, the average running time does not lower consistently, with the chart showing a remarkable spike at the final run. Besides, the average time Dijkstra occupies seems relatively low, given roughly 1.5 million nodes.

In view of the data in Fig. 6.14, it appears that the majority of the execution time is taken by the precise phase. Fewer precise macrosteps are usually done, compared to greedy steps, but they take considerably more time. Combining these results with those shown in Fig. 6.14 while considering that a macrostep partitions the graph several times, it is likely that the over-reliance on parallel Dijkstra in the precise phase slows down the algorithm.

It follows that using denser graphs and meshes would make convergence times longer, since Dijkstra's complexity is a function of the number of nodes and arches.

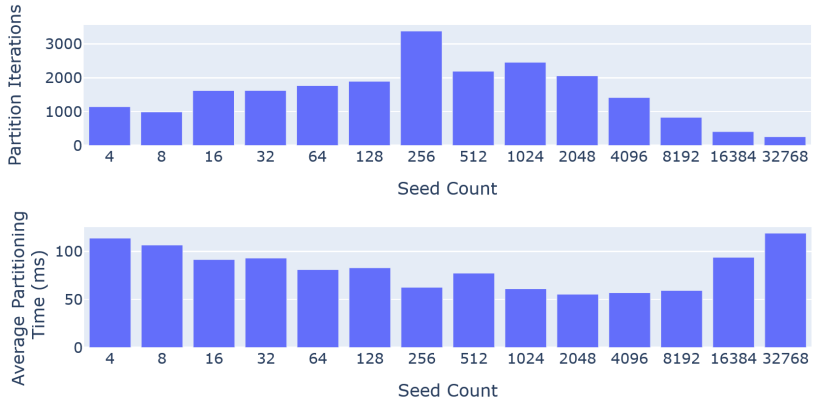

Figure 6.13: On the top, the count of each parallel Dijkstra execution during the whole relaxation. On the bottom, its average execution time.

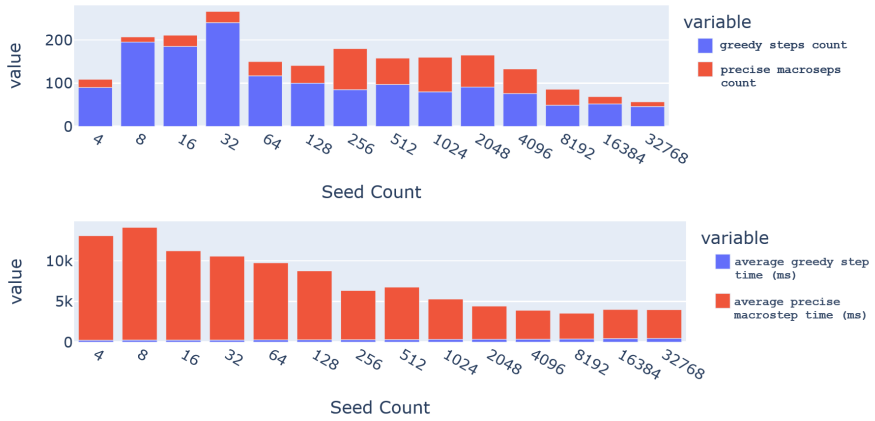

Figure 6.14: On the top, the count of greedy steps compared to the precise macrosteps. On the bottom, their average time.

6.3 Additional tests

This section includes additional results for completion's sake. The statistics of the runs and the models, as well as a short gallery of images showcasing the tessellation, are provided below.

Model	Vertices	Triangles	Graph	Seeds	Run time
Torso	44,580	29,720	Vertex dense	1024	769.775 ms
Standford bunny	34,834	69,451	Triangle	512	5,292.55 ms
Hippo (lo-res)	29,431	58,852	Vertex (aniso.)	4096	649.013 ms
Hippo (hi-res)	706,244	1,412,448	Vertex	2048	5.56 min

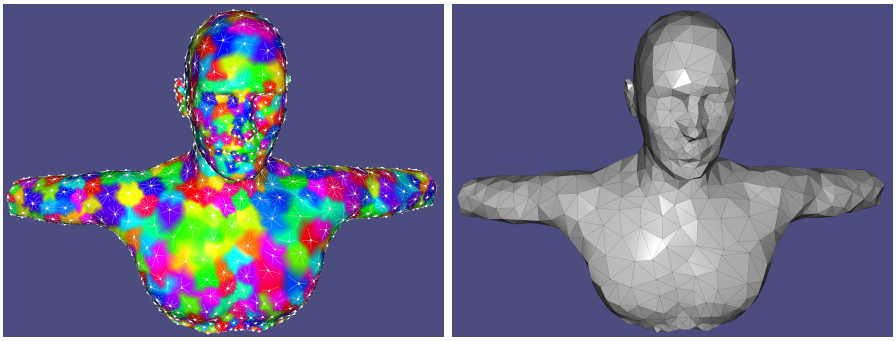

Figure 6.15: Isotropic tessellation and triangulation preview of the torso model.

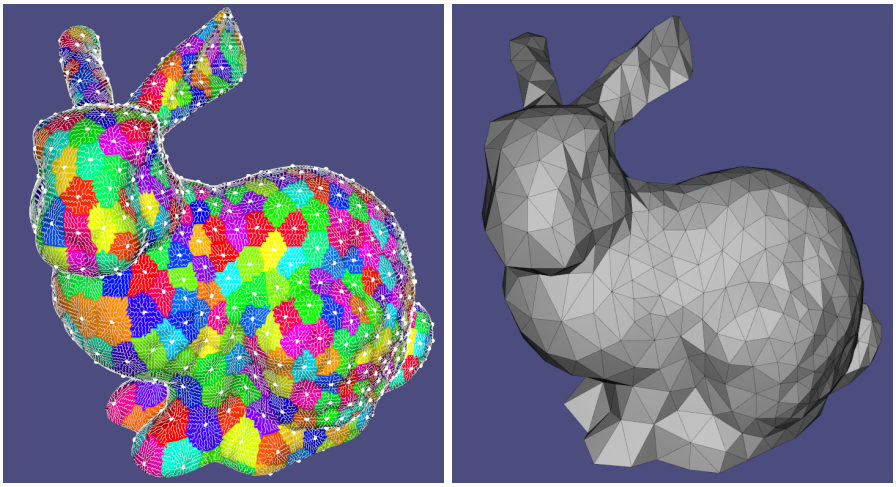

Figure 6.16: Isotropic tessellation and triangulation preview of the bunny model.

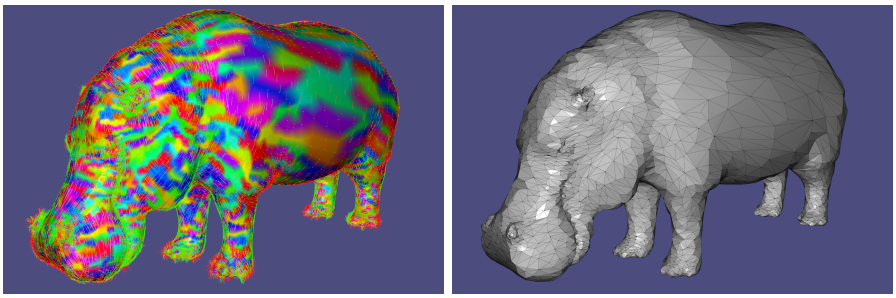

Figure 6.17: Anisotropic tessellation and triangulation preview of the hippo (lo-res) model.

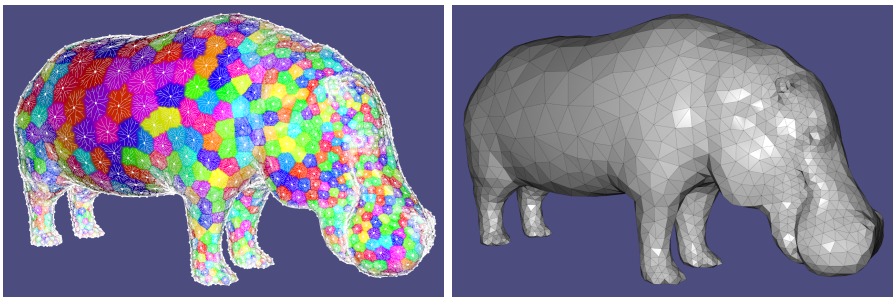

Figure 6.18: Isotropic tessellation and triangulation preview of the hippo (hi-res) model.

Chapter 7

Conclusions

This last chapter is a brief discussion on the implications of the results previously reported.

7.1 Considerations on the results

The algorithm has shown to produce competent CVTs over the input mesh's surface. It fully supports anisotropic metrics, and these influence the tessellation as expected.

When performance is concerned, our solution performs well on relatively simple meshes and non-dense graph structures, but it showed that it does not scale well with dense graphs and high-resolution meshes. The latter is underwhelming, considering the ideal use case this algorithm was designed for involves models with a vast amount of polygons.

7.2 Limitations

Both relaxation phases, greedy and precise, suffer from locking themselves into local minima. In the course of our experiments, we noticed that moving seeds randomly after convergence, even by a slight amount, and then restarting the whole relaxation loop may lead to better CVTs.

Moreover, toggling the greedy relaxation may or may not improve the final tessellation. As shown in Fig. 7.1, the relaxation loop was run on the same seeds, first enabling the greedy phase, and then disabling it. The second run gave a more regular result. Yet, this is not always the case. Not running the greedy phase at all can also yield worse results at times.

The reason why the relaxation ends up in local minima must be deeply dissected to further improve the outcoming tessellation.

Another notable limitation of this algorithm is the lack of proper support for Manhattan or L-infinity distance functions. In some contexts, building a VD with these distance functions is useful for generating quad-dominant meshes [10] and other applications [8]. As shown in Fig. 7.2, the resulting CVTs are not as expected, when the Manhattan or L-infinity distance functions are concerned.

The disappointing results obtained by using non-geodesic distances may be a side effect of our discrete approach.

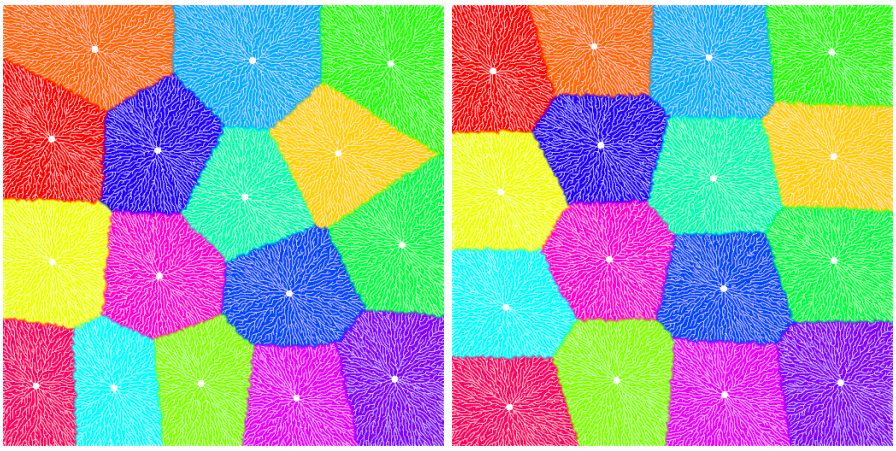

Figure 7.1: On the left, the greedy phase was kept enabled. On the right, it was disabled. In this specific case, disabling the greedy phase improved the tesselation.

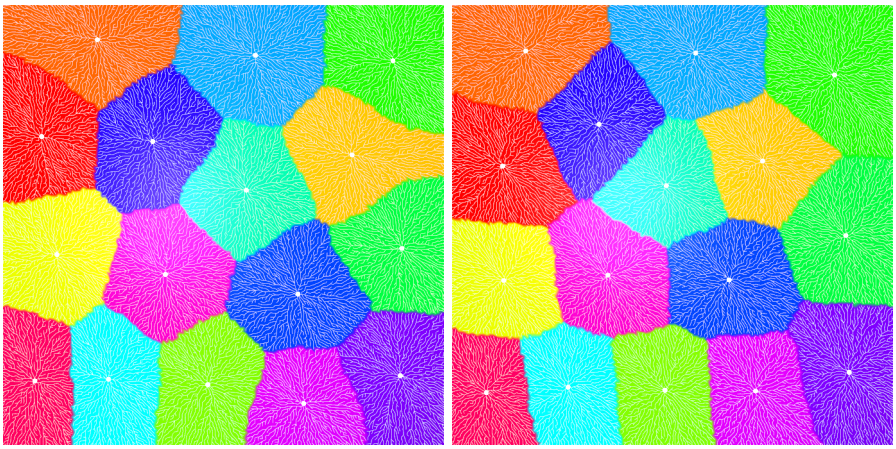

Figure 7.2: Partitioning can be done with other distance functions than geodesics. On the left, the Manhattan distance. On the right, the L-infinity distance.

7.3 Further work

The previous sections emphasize the importance of improving our algorithm. Subsection 6.2.3 shows evidence of how parallel Dijkstra heavily influences the precise phase's performance. This could be addressed by some hardware parallelization or multithreading approach, if not considering to limit the amount of partitioning within the precise movement routines.

Relaxation needs to counter the issue of local minima to improve the final tessellation. A proper support of non-geodesic distance functions can be considered.

Averaging more than two tangent spaces still requires a solid implementation, as summarized in subsection 4.1.5. This is problematic for retrieving a tangent space for a triangle.

Bibliography

[1] Pierre Alliez, Mark Meyer, and Mathieu Desbrun. "Interactive geometry remeshing". In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 347–354. ISSN: 0730-0301. DOI: 10.1145/566654.566588. URL: https://doi.org/10.1145/566654.566588.

[2] Pierre Alliez et al. "Centroidal Voronoi Diagrams for Isotropic Surface Remeshing". In: *Graphical Models* 67 (May 2005), pp. 204–231. DOI: 10.1016/j.gmod.2004.06.007.

[3] Paolo Brivio, Marco Tarini, and Paolo Cignoni. "Browsing Large Image Datasets through Voronoi Diagrams". In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (2010), pp. 1261–1270. DOI: 10.1109/TVCG.2010.136.

[4] Qiang Du, Vance Faber, and Max Gunzburger. "Centroidal Voronoi Tessellations: Applications and Algorithms". In: *SIAM Review* 41.4 (1999), pp. 637–676. DOI: 10.1137/S0036144599352836. eprint: https://doi.org/10.1137/S0036144599352836. URL: https://doi.org/10.1137/S0036144599352836.

[5] Qiang Du, Max D. Gunzburger, and Lili Ju. "Constrained Centroidal Voronoi Tessellations for Surfaces". In: *SIAM Journal on Scientific Computing* 24.5 (2003), pp. 1488–1506. DOI: 10.1137/S1064827501391576. eprint: https://doi.org/10.1137/S1064827501391576. URL: https://doi.org/10.1137/S1064827501391576.

[6] Herbert Edelsbrunner and Nimish R. Shah. "Triangulating topological spaces". In: SCG '94 (1994), pp. 285–292. DOI: 10.1145/177424.178010. URL: https://doi.org/10.1145/177424.178010.

[7] Martin Erwig. "The graph Voronoi diagram with applications". In: *Networks : an international journal* 36.3 (Oct. 2000). ISSN: 0028-3045.

[8] Alejo Hausner. "Simulating decorative mosaics". In: SIGGRAPH '01 (2001), pp. 573–580. DOI: 10.1145/383259.383327. URL: https://doi.org/10.1145/383259.383327.

[9] R. Kunze, F.-E. Wolter, and T. Rausch. "Geodesic Voronoi diagrams on parametric surfaces". In: (1997), pp. 230–237. DOI: 10.1109/CGI.1997.601311.

[10] Bruno Lévy and Yang Liu. "Lp Centroidal Voronoi Tessellation and its applications". In: *ACM Trans. Graph.* 29.4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778856. URL: https://doi.org/10.1145/1778765.1778856.

[11] Yang Liu et al. "On centroidal voronoi tessellation—energy smoothness and fast computation". In: *ACM Trans. Graph.* 28.4 (Sept. 2009). ISSN: 0730-0301. DOI: 10.1145/1559755.1559758. URL: https://doi.org/10.1145/1559755.1559758.

[12] Gabriel Peyré and Laurent Cohen. "Geodesic Remeshing Using Front Propagation". In: *International Journal of Computer Vision* 69 (Aug. 2006). DOI: 10.1007/s11263-006-6859-3.

[13] J. A. Sethian. "Fast Marching Methods". In: *SIAM Review* 41.2 (1999), pp. 199–235. DOI: 10.1137/S0036144598347059. eprint: https://doi.org/10.1137/S0036144598347059. URL: https://doi.org/10.1137/S0036144598347059.

[14] Sebastien Valette, Jean Marc Chassery, and Remy Prost. "Generic Remeshing of 3D Triangular Meshes with Metric-Dependent Discrete Voronoi Diagrams". In: *IEEE Transactions on Visualization and Computer Graphics* 14.2 (2008), pp. 369–381. DOI: 10.1109/TVCG.2007.70430.

[15] Dong-Ming Yan et al. "Isotropic Remeshing with Fast and Exact Computation of Restricted Voronoi Diagram". In: *Computer Graphics Forum* 28.5 (2009), pp. 1445–1454. DOI: https://doi.org/10.1111/j.1467-8659.2009.01521.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01521.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01521.x.