UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PARALLEL MESH SIMPLIFICATION ON A GPU

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

VALENTÍN LEONARDO MUÑOZ APABLAZA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA Z.
CRISTÓBAL A. NAVARRO GUTIERREZ
FERNANDO RANNOU FUENTES

SANTIAGO DE CHILE
2020

RESUMEN DE LA TESIS PARA OPTAR AL
GRADO DE: MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
POR: VALENTÍN LEONARDO MUÑOZ APABLAZA
FECHA: 2020
PROF. GUÍA: SRA. NANCY HITSCHFELD KAHLER

PARALLEL MESH SIMPLIFICATION ON A GPU

Las mallas geométricas son un potente medio que permite representar fenómenos del mundo real en el mundo digital, y así resolver numéricamente, problemas que no necesariamente poseen una resolución analítica. Sin embargo, el procesamiento de mallas geométricas que almacenan o representan grandes volúmenes de datos, tales como modelamiento de superficies u objetos 3D, se ha vuelto lento o inmanejable al utilizar algoritmos secuenciales para procesarlos.

Una de las operaciones realizadas sobre estas mallas es la simplificación, la cual reduce o elimina detalles geométricos y disminuye el uso de memoria de la malla, manteniendo un nivel aceptable de calidad y reteniendo propiedades importantes de ella, tales como preservar su forma original, geometría y topología. Con esto, el resultado de los algoritmos que usan como entrada esta malla simplificada, debiesen obtenerse más rápido, y seguir considerándose válidos y útiles.

Este trabajo de tesis propone un algoritmo de simplificación de mallas en paralelo basado en *edge-collapse* utilizando la GPU, tomando en cuenta varios elementos encontrados durante el proceso de investigación (tales como técnicas de simplificado, métricas de simplificación y algoritmos creados por otros autores). Esta propuesta incluye el uso de una estructura de datos utilizada en la librería Cleap, que almacena datos de la malla redundantemente y puede almacenarse en la memoria de la GPU. Esto es útil, pues los elementos de la malla (vértices, arcos y triángulos) pueden detectar inconsistencias en sus datos (principalmente referencias a elementos eliminados de la malla) luego de una iteración del algoritmo de simplificado sobre la malla, y con esto pueden reparar tales inconsistencias antes de iniciar la siguiente iteración del algoritmo.

Los resultados muestran que el algoritmo de simplificación paralelo es hasta 1.000 veces más rápido en promedio que su contraparte secuencial bajo ciertos escenarios y consideraciones, y las mallas resultantes son de calidad similar entre ambas implementaciones. También se presentaron algunos cuellos de botella en algunas etapas, que disminuían el rendimiento del algoritmo como un todo. No pudimos realizar una comparación de nuestro algoritmo con otros algoritmos encontrados durante la revisión bibliográfica, debido a que sus implementaciones no se encontraban disponibles para su uso y/o su descripción para implementarlo por nuestra cuenta era poco clara o tenía ofuscadas etapas clave del algoritmo.

Nuestro aporte es un algoritmo de simplificación de mallas en paralelo que funciona en la GPU, que está integrado en una librería que cuenta con implementaciones paralelas de otros algoritmos de refinamiento de mallas (como Triangulaciones de Delaunay y suavizado), y los detalles de su implementación se encuentran disponibles como un proyecto de código abierto. Este trabajo también aporta que se requiere saber para programar en paralelo en la GPU, así como una revisión de los algoritmos, técnicas y detalles encontrados durante la investigación.

Abstract

PARALLEL MESH SIMPLIFICATION ON A GPU

Geometric meshes are a powerful medium that allows us to represent real world phenomena in the digital world, and numerically solve problems that do not necessarily have an analytic resolution. However, the processing of meshes that handle or represent large volumes of data, such as surface modeling or 3D objects, has become slow or unmanageable when using sequential algorithms to process them.

One of the operations performed over these meshes is *simplification*, which reduces or eliminates geometric details and overall memory usage, while maintaining an acceptable level of quality and retaining some key properties, such as preserving its original shape, geometry and topology. With this, the result of algorithms that use the simplified mesh as an input should be faster while considered valid and useful.

This thesis proposes a parallel mesh simplification algorithm based on *edge-collapse* using the GPU, taking into account various elements encountered during the investigation process (such as simplification techniques, simplification metrics and algorithms created by other authors). This proposal also features the use of a data structure to store mesh data present in the Cleap library, which stores redundant data of the mesh and can be stored in GPU memory. This is helpful, because the elements of the mesh (vertices, edges and triangles) can detect their own inconsistencies (principally references to deleted elements) after a simplification step is performed over the mesh, and repair themselves after the next simplification step.

The results show that the parallel simplification algorithm is at most 1,000 times faster on average than its sequential counterpart under certain scenarios and considerations, and the resultant meshes share a similar quality between both implementations. Additionally, we observed some bottlenecks in some stages of the algorithm, that decreased the overall performance of it. We could not perform a comparison of our algorithms with other algorithms encountered during the review, because their implementations were not available and/or implementation steps to do it ourselves were not clear or some key steps were obfuscated.

Our contribution is a parallel simplification algorithm that works on the GPU, that is integrated into a library, which has parallel implementations of other mesh processing algorithms (such as Delaunay triangulation and smoothing), and its implementation details are available as an open-source project. This work also contributes an insight of what is required to start on parallel GPU programming, and also contributes with a review of the algorithms, techniques and details encountered during the investigation.

A mis gatos, doggos, familia, amiwis
y a quienes ya no están conmigo

Agradecimientos

Dicen que las segundas partes no siempre son las mejores, pero sin duda esta aventura fue super entretenida, llena de emociones, pasando por muchos sucesos inesperados (no necesariamente relacionados con el trabajo desarrollado en este documento) y dejando valiosas enseñanzas.

En primer lugar, quisiera agradecer a la Profesora Nancy Hitschfeld y al Profesor Cristóbal Navarro por todo el apoyo, paciencia y ayuda brindada durante el desarrollo de esta tesis de grado.

Por supuesto, agradecer a mi Familia, en especial a mi madre, mi padre y mis hermanas, ya que es de mala gente no agradecerles, y que sin ella y el apoyo y ayuda durante en esta aventura no sé dónde estaría ahora. Si, los gatitos y perritos son parte de la familia también, y les agradezco todas las alegrías brindadas, sus recibimientos, su cariño y su apañe en las noches en vela.

La vez pasada prometí que ahora si nombraría amiwis en especial (quizá se me pasen algunos nombres, disculpas de antemano :c). Empezaré agradeciendo a la Anto⛵ y al Vicente🦄(Ephy), por mostrarme que no hay que rendirse ante la adversidad y que siempre habrá gente dispuesta a tenderte una mano y brindar su apoyo y cariño. Al Sebi Cisneros💜, la Kari🐊, al Hiho🍺, al Pipe, a la Iva y al Witu🐧por apañar con su apoyo frente a diversos sucesos que surgieron en esta aventura. A los *frens* like la Dani🏐, Peibl, Cristobal🍍, la Mati🐶, Franqui3🦝, Checho🦑, Sebi Cisneros again, Seba3, Tait, Jordo, Barbi, Snevs y Pancho Lecaros🚮por apañarme en el trayecto final de esta aventura y su apoyo frente a uno de los temas más sensibles que me ha tocado enfrentar hasta ahora. A los *cultistas* Claudin, Pato Isbej, Gabo Dintrans🥑, Jorgi3, Varitas🍁, Anduresu, El Dani, Sebi Ferra3💙, Sebi Cisneros once more, Madurido, Hiho, Ardilom🐿️, Doge, Tomi, BastyZ, Gonzarou, Pipe, Joseto y Coloro por sacarme a conversar de la vida, despejarse un rato y salir a tomar cerveza y ramen. Agradezco también a los k-bros del *Age of Yeds* y del *Above the Clock*, en particular a Tomimi y Diggo🦊, por apañar a jugar jueguitos y ganar muchos puntos en diversión.

Agradezco también a todas las personas que alguna vez pasaron por alguna de las salas de clases o laboratorios donde me tocó oficiar de Profesor Auxiliar, ya que aprendí mucho de ellas, y que las cosas pueden salir super bien cuando existe motivación y pasión. En este apartado, quiero agradecer en especial a Cristian Parra🧞, por ser mi *partner in crime* en esta aventura de la docencia y apañar en diversas *wild ideas* que surgían en el camino, y a todos los docentes que me brindaron la confianza para trabajar bajo su alero. También agradecer a los docentes y funcionarios del DCC, en especial a Sandra y Angélica, por toda la ayuda académica y administrativa que me brindaron.

Finalmente, agradecer a toda la gente con quien haya compartido una palabra, sonrisa o alegría en este periodo, a todes les amiwis que hice en el camino y a quienes no nombré por falta de espacio. Mención especial al espacio de *la salita* y *la ofisalita* y su población recurrente, que fue donde empecé a escribir mi memoria de pregrado, y curiosamente luego de moverme por hartos lugares, terminé escribiendo allí la parte final de esta tesis, marcando simbólicamente el fin de este ciclo.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

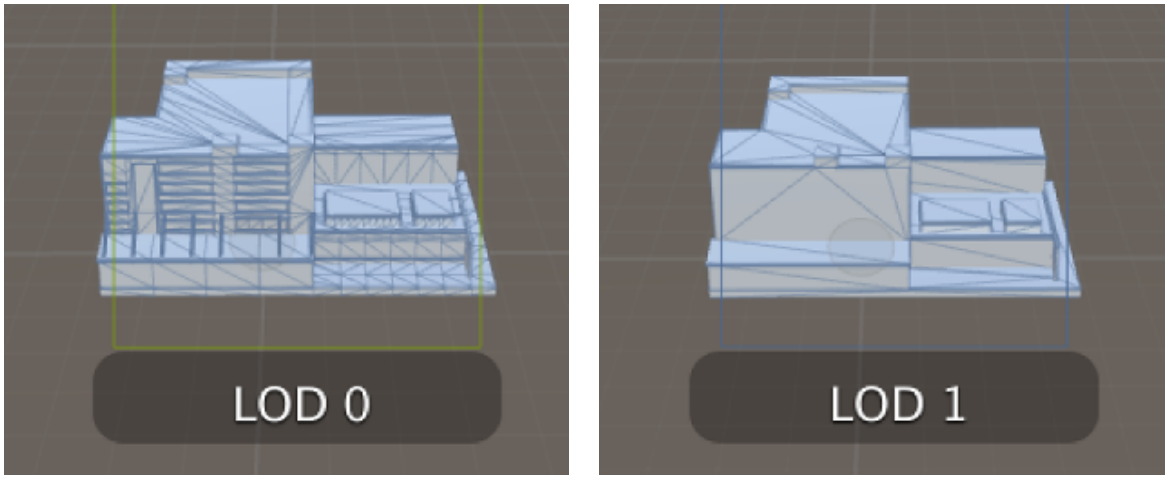
1.1 Motivation

Polygon meshes are a powerful tool that allows modeling of real world phenomena, and help solve problems that do not necessarily have an analytic resolution. However, processing meshes that handle and represent large volume of data, such as the modeling of land surfaces for Geophysics and Geology, or models representing simulations of physical phenomena, have become slow or not viable when using sequential algorithms, especially when they are real-time applications.

For example, an application takes successive photos of a 3D object in the real world (such as CT Scan or MRI), and generates a point cloud, which is converted into a 2D or 3D mesh or surface that represents this object (via Delaunay Triangulation, convex hull or alpha shapes). A problem that can arise is that the size of the generated mesh can be very large (order of 10 million vertices), which may be difficult to process and handle, even for computers with large computing capacity. This happens because the mesh generated by the conversion from the point cloud has a high level of detail (LOD) across all the object, even for zones that does not require such level of detail, like flat or plain sections of an object.

Another example are real-time applications, such as 3D games or visualization of terrain surfaces, in which meshes need to be of a certain size that allows the application that uses it to run smoothly, have good detail for objects near the point of view, and minimum detail for very far objects. In Figure 1.1 an image is shown of tthe same building with different levels of detail.

In the last couple of years, the Graphics Processing Unit (GPU), usually destined to render and process images to a display device, has been used as a data processing unit, due to its capacity to process high amounts of data in parallel in small amount of time. Therefore it is necessary to implement versions of mesh processing algorithms that allow parallel processing of the data, in order to improve the performance of algorithms or software that use polygon meshes as an input, such that its performance be proportional to the input size.



(a) Building with higher level of detail. (b) Building with lower level of detail.

Figure 1.1: Mesh of a building with different levels of detail. Picture taken from `https://docs.unity3d.com/Manual/LevelOfDetail.html`

One of the techniques used for mesh processing is mesh simplification, which reduces the elements of a mesh (edges, vertices and faces) while trying to maintain the original geometry and topology of the input mesh. An example of what we would like to accomplish is shown in Figure 1.2, in which zones with high count of slopes (like mountains) have more elements that zones that are more flat (like valleys).

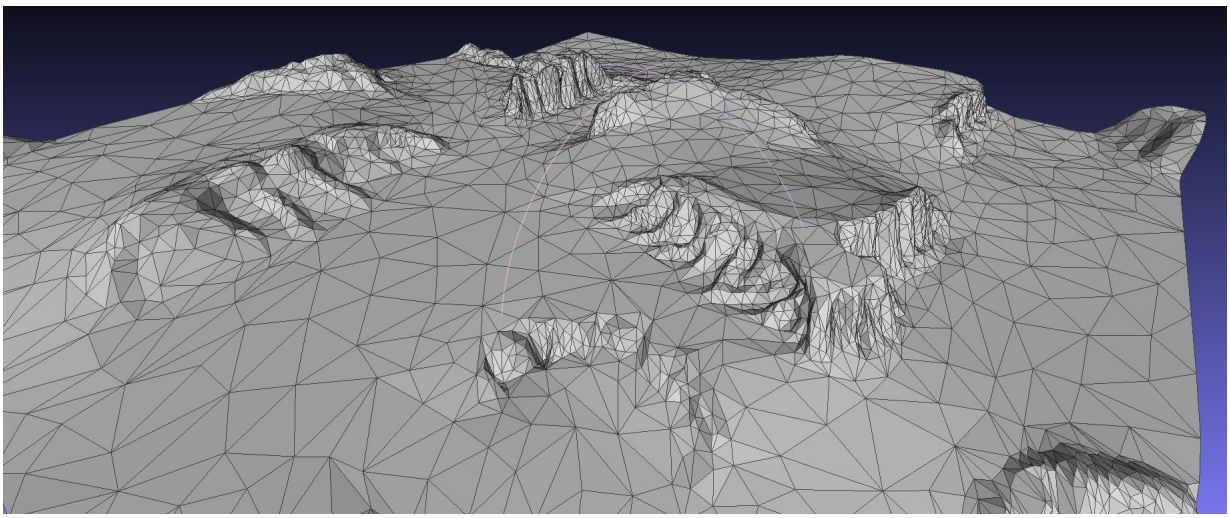


Figure 1.2: Example of a terrain mesh. Picture taken from `visualcomputing.github.io/representation/fig/terrain.jpg`

Also, on a previous work [16], we had worked in other kind of mesh processing algorithms under the Cleap Library [18], which implements a parallel algorithm to triangulate a mesh with the Delaunay criteria using *edge-flips* implemented in Cuda-C programming language. We implemented a parallel mesh smoothing algorithm (Taubin smoothing), and compared the performance of the Delaunay triangulation with the angle criteria versus the circumcircle criteria, to decide whenever an *edge-flip* has to be performed or not.

In this work, we want to implement a mesh simplification algorithm that works on high LOD terrain meshes and preserves the original topology and geometry. This algorithm uses the parallel processing power of the GPU to assure the simplification can be processed on the smallest amount of time possible, and must be integrated in the Cleap library to ensure compatibility with the other mesh processing algorithms mentioned earlier.

Our research questions are: can data structures that use redundant information allow one to design and implement more efficient parallel algorithms that the ones previously published in the literature?

1.2 Objectives

1.2.1 General Objective

The main objective of this thesis is to design and implement a novel parallel mesh simplification algorithm using the parallel processing power of the GPU, and compare its performance with an ad-hoc sequential simplification algorithm.

1.2.2 Specific Objectives

1. Provide a small survey about the "state of the art" of mesh simplification algorithms.
2. Integrate a parallel mesh simplification algorithm on the Cleap library.
3. Evaluate if the algorithm performs well under certain terrain meshes given as an input.

1.3 Methodology

To reach these objectives, we perform the following steps:

1. Research about the "state of the art" of mesh simplification algorithms.
2. Research about parallel programming on the GPU.
3. Create or adapt Cleap[1] redundant data structures to support parallel mesh simplification.
4. Develop a sequential and parallel mesh simplification algorithm in Cuda-C language.
5. Do a performance test of the algorithm and a quality test on its output.

[1]Cleap[17] is a library that implements a parallel Delaunay triangulation algorithm. We will talk more about Cleap in Section 2.3

1.4 Structure of this Work

This work is organized as follows:

1. In Chapter 2, we describe the theoretical framework of this thesis, such as GPU parallel programming, Cleap Library and mesh processing algorithms.
2. In Chapter 3, we review different mesh simplification algorithms, emphasizing the kind of simplification, resources and assumptions that each one of them uses.
3. In Chapter 4, we describe all stages of the designed algorithm and how it works in parallel.
4. In Chapter 5, we show the algorithm results, and a comparison between the sequential and parallel implementations in a qualitative and a quantitative way.
5. Finally, in Chapter 6, we summarize the conclusions of this work and discuss some of the future research lines for this work.

Chapter 2

Theoretical Framework

This research lies between Geometric Mesh Processing, Computer Graphics and Parallel Programming areas. We want to achieve a mesh simplification algorithm that can work on geometric meshes representing terrains and using the parallel processing power of the GPU.

In this chapter, we will show the background and concepts necessary to understand the subsequent investigation. This chapter is organized as follows: In Section 2.1 we talk about Polygon Meshes fundamentals and operations that can be done to them; later, in Section 2.2 we talk about GPU architecture and its programming model; In Section 2.3 we describe the Cleap Library fundamentals, including data structures and implemented algorithms; finally, in Section 2.4 we describe Geometric meshes file formats, for input and output of the algorithms.

2.1 Polygon Meshes

In computer graphics, a Polygon Mesh is a set of vertices, edges and faces that define the shape and/or surface of an object, and can be used to represent real world objects in the virtual world. With this, it is possible to simulate or visualize actions or phenomena that are costly or unfeasible to perform, like the consequences of an earthquake on a certain terrain, the deformation degree of two cars that collide with each other (as can be seen in Figure 2.1), or visualize the weight handled by the structural beams of a building, etc.

Although there are volume geometric meshes, which explicitly represent the surface and volume of an object (for example, tetrahedral or hexahedral meshes), in this research we will focus only on surface geometric meshes, which explicitly represent the surface and implicitly the volume. So, we will refer to surface geometric meshes simply as meshes or geometric meshes.

Figure 2.1: Car crash simulation using Polygon Meshes. Picture taken from `http://www.gameoverviews.com/wp-content/uploads/2012/07/BeamNG-600x300.png`

As can be seen in Figure 2.2, the components of a geometric mesh are:

- Vertex: A position or coordinate in the 2D or 3D space.
- Edge: A connection (usually a straight line) between two vertices.
- Face: A closed set of edges, typically forming a convex figure, such as a triangle or quadrilateral.

And each component may have extra properties, like colour, transparency or textures. In this research we also use the term "elements" as synonym of the components of a mesh.

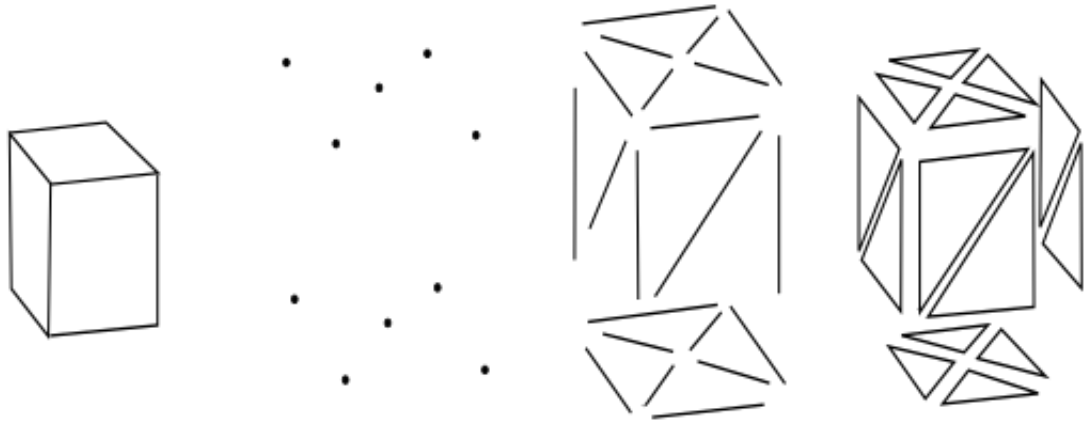


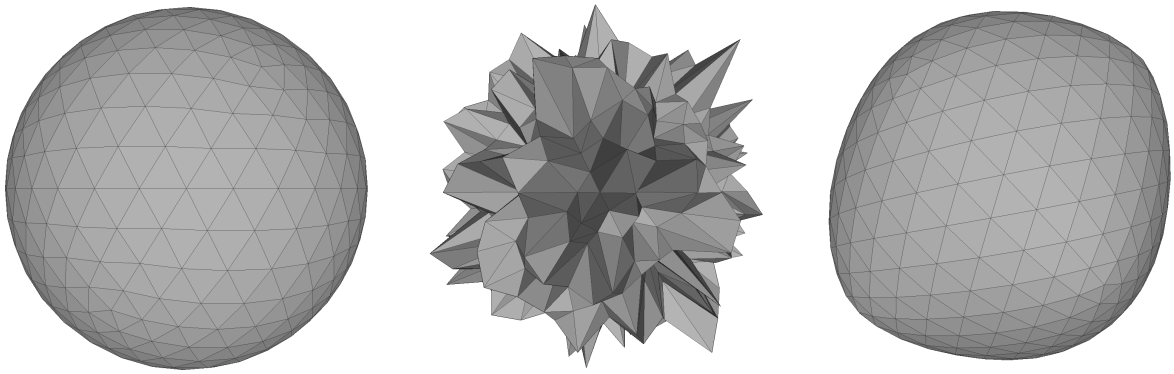
Figure 2.2: Components of a geometric mesh: original object, vertices, edges and faces. Picture taken from Wikipedia, `https://en.wikipedia.org/wiki/File:Mesh_overview.svg`

Sometimes, it is necessary to preprocess the mesh, due to factors such as: the mesh has too many elements, it does not faithfully represent the shape or details of the object, there are over-represented zones that does not need too much level of detail, etc. These issues can affect the output or the performance of algorithms and software that uses these meshes as input.

Some of those preprocessing techniques reorganize or remove components of the mesh, in order to reduce time or space used for this mesh. In the next sub-sections, we will introduce some of this techniques.

2.1.1 Smoothing

Smoothing consist of reordering the vertices in the mesh and/or its connectivity, to reduce the *noise* or imperfections on it, in order to represent the surface more accurately, capturing important patterns in the provided data. For example, different states of the same sphere after a deformation and smoothing process can be seen in Figure 2.3.



(a) Sphere in its original shape.

(b) Sphere after a deformation process.

(c) The sphere of (b) after smoothing process.

Figure 2.3: Sphere before and after a smoothing step.

One of this kind of algorithms is Taubin Smoothing algorithm[16], which uses a signal processing approach based on signal processing smoothing (which uses Fourier Analysis techniques) to change the position of the vertices according to the position of their neighboring vertices, without changing the connectivity and the number of vertices and faces of the original mesh. With this approach, the algorithm can detect which vertices produces noise in the other vertices, changing their positions to reduce the overall perturbations and distortions on the mesh, allowing the mesh to adopt a shape similar to its original form.

2.1.2 Delaunay Triangulation

A Delaunay Triangulation is a triangulation of a set of vertices, such that no vertex is inside of the circumcircle of any triangle formed by other 3 vertices (this is also known as the Delaunay condition).

With this restriction, this kind of triangulation maximizes the minimum angle on all the triangles in the triangulation, trying to minimize the number of sliver triangles (that can be seen in Figure 2.4). Slivers are undesirable in a long range of computer graphics applications, such as rendering or rasterization, because their vertices may be almost collinear and be interpreted as a simple segment instead of a triangle. Because of this, they also can produce inexact arithmetic calculations.

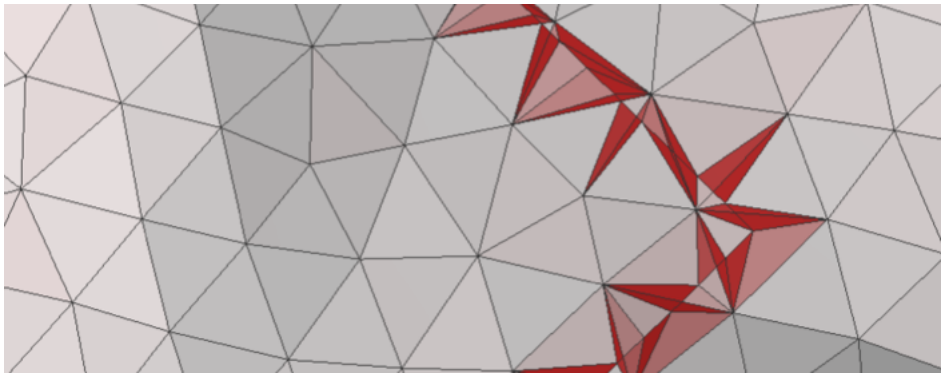
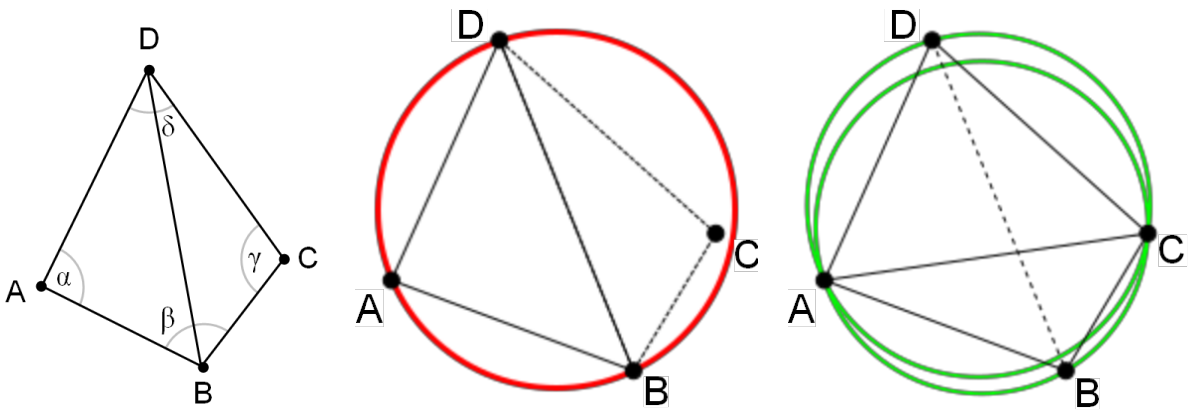


Figure 2.4: Example of a mesh with sliver triangles (in red). Picture taken from `http://www.mmmanual.com/mesh-query/`

There are many algorithms for computing a Delaunay triangulation, based on incremental constructions, divide and conquer, flipping strategies, etc. and which one to use depends on the input data and/or the application where the algorithm is used, but all rely on detecting when a vertex is inside of a triangle circumcircle, and do an operation to correct those vertex or triangles that do not satisfy the criteria. An example of one of these procedures (edge-flip) is shown in Figure 2.5.



(a) Triangulation that does not satisfies Delaunay condition ($\alpha+\gamma > 180°$).

(b) The circumcircle criterion is not satisfied, because vertex C lies in the ABD circumcircle.

(c) After flipping edge BD to AC, the circumcircle criteria is satisfied.

Figure 2.5: Example on how to satisfy the Delaunay criteria. Pictures taken from Wikipedia `https://en.wikipedia.org/wiki/Delaunay_triangulation`.

2.1.3 Simplification

Mesh simplification consists of removing components in a mesh to reduce its memory size and the overall count of elements, while trying to preserve its original topology and/or geometry, avoiding mesh degeneration such as self-intersections on faces or over-connected vertices. An example of different degrees of simplification over a hand mesh are shown in Figure 2.6.

Figure 2.6: Different degrees of simplification. Picture taken from `https://doc.cgal.org/latest/Surface_mesh_simplification/index.html`

Some of the simplification techniques include edge-collapse, vertex-contraction and triangle-decimation. which focus on deleting specific elements of the mesh, while repairing the connectivity of the remaining elements. We will review how these techniques work in more detail on Chapter 3.

2.2 GPU Architecture

In this work, we will use Nvidia CUDA[1]. CUDA is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. In addition, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, Java, Python, DirectCompute, OpenACC, etc.

First, we will describe the GPU architecture from a hardware point of view, and after that, how to use it from a programming point of view.

While a CPU is designed to process tasks and do calculations that requires a lot of flow control and synchronization, a GPU instead is designed to perform highly parallelizable arithmetically intensive calculations that do not depend directly on each other.

In a CPU, the majority of the allocated physical space is used to place memory cache, and little space to place ALUs[1]. Because of this distribution, each CPU core has a big memory cache hierarchy, which helps to greatly reduce latency in recurrent calculations. Also parallel processing is limited by the low number of cores present in a CPU, and the large amount of context switch that the CPU has to do to handle the task of running a computer.

On the other hand, in a GPU, the allocated physical space is used to place ALUs instead, at the cost of smaller memory cache per GPU core, as can be seen in Figure 2.7. With this, it is possible to do a lot of parallel calculations in comparison to a CPU, and to maintain memory latency at acceptable levels, it is recommended to launch a lot of threads at the same time (at least 32), to hide memory access and data migration between CPU memory (RAM) and GPU memory (VRAM) with calculations with high arithmetic intensity.

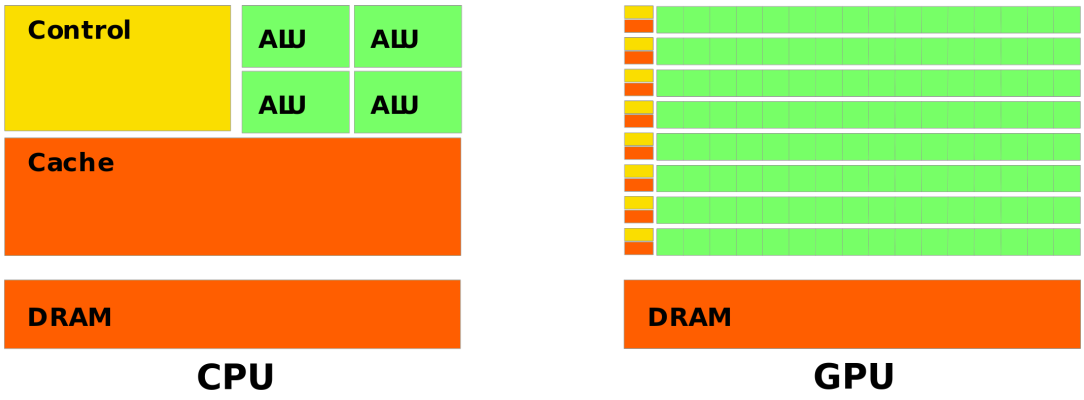


Figure 2.7: Comparison of physical space utilization in CPU and GPU architectures. Diagram obtained from CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations, i.e., the same program is executed on many data elements in parallel. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control and synchronization that a CPU has.

CUDA has three abstractions that are exposed as a minimal set of language extensions: a hierarchy of thread groups, shared memory, and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

[1] Arithmetic Logic Unit, which is responsible of performing arithmetic and bit-wise operations

The GPU architecture is built around a scalable array of multi-thread *Streaming Multi-Processors*(SMP). Each SMP contains a certain number of:

- Scalar Processor cores, which calculate simple operations between integers or floating point numbers, such as addition, multiplication, etc.
- Special function units, which calculate more complex mathematical operations, such as roots and trigonometric functions.
- Multi-threaded instruction unit, which organize the execution order of the program across all assigned threads.
- Shared Memory, which can be read and written by a certain group of threads, depending on the hierarchy level of the memory.

And some key definitions about the programming resources available to the programmer are:

- Kernel: Program or subroutine that is executed N times in parallel by C different CUDA threads (N, C positive integers).
- Thread: An independent execution of a program, with it own set of values and registers.
- Block: An array of threads.
- Grid: An array of blocks.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, depending on how many hardware resources an specific CUDA-capable card has. For example, in Figure 2.8, it is shown how a CUDA program that fits in 8 blocks, can adapt in run-time to fit on the SMPs available in each case.

2.2.1 Programming model

As said before, CUDA is a language extension of C, allowing to define a special kind of functions, called kernels.

Each thread that executes a kernel is given a unique *thread ID*, which is a 3-component vector, that enables the use of a 1D, 2D or 3D thread blocks among a grid. With this, it is possible to realize computations across the elements in domains represented as a vector/array, a matrix/grid or a volume/tensor respectively, matching the thread indices with the indices of the array, matrix or tensor representing the problem data.

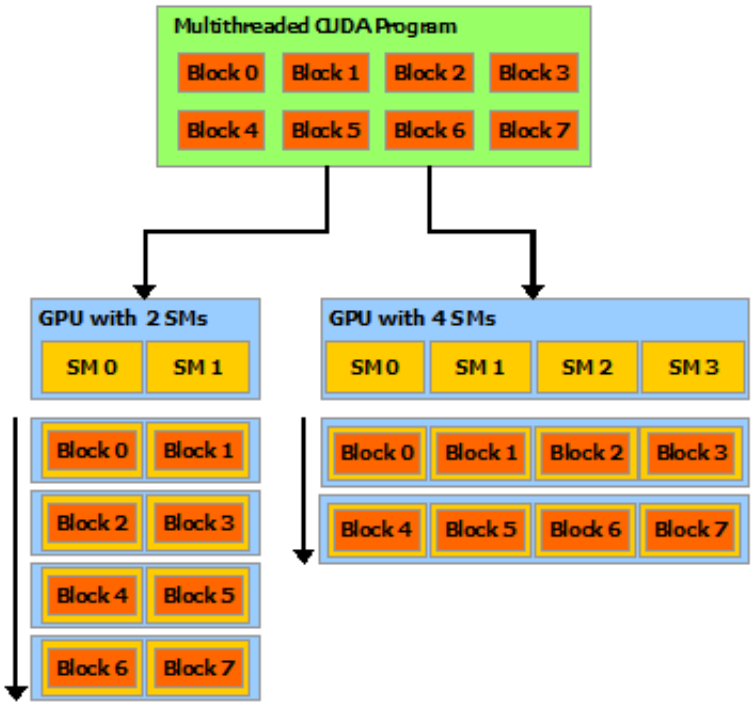


Figure 2.8: Automatic Scalability of a CUDA program, who adapts to different availability of SMPs on a GPU. Diagram obtained from CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same SMP and must share the limited memory resources of that SMP. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. The number of threads per block and the number of blocks per grid are specified when a kernel is executed, according to its needs of computation capacity. In Figure 2.9, it is shown an example of a Grid with 2×3 blocks, and each block has a configuration of 3×4 threads, representing an application in a 2D space.

Each thread block is executed independently of each other, which allows blocks to be scheduled in any order across the available multiprocessors. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the *syncthreads()* intrinsic function, that acts as a barrier at which all threads in the block must wait before any is allowed to proceed with the execution of the program.

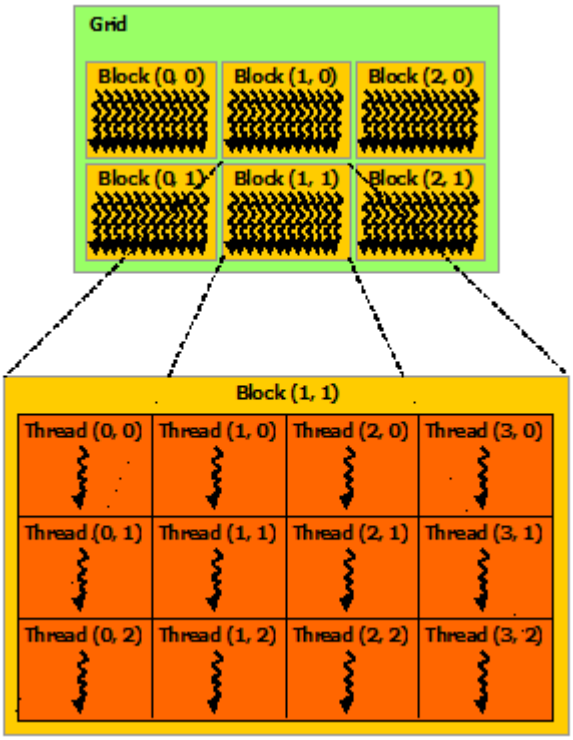


Figure 2.9: Example of a grid organization of 2D thread blocks. Diagram obtained from CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

CUDA threads may access data from multiple memory spaces during their execution. In Figure 2.10 it is shown that each thread has private local memory, each thread block has shared memory visible to all threads of the block, and all block grids have access to a common global memory. This enables a lot of flexibility to implement algorithms that requires some kind of data sharing or soft synchronization across threads, without incurring on big critical sections or memory locks.

2.2.2 Heterogeneous Computing

The CUDA programming model assumes that CUDA threads are executed on a physically separate device that operates as a coprocessor to the host running the C program. This is the case when the kernels execute on a GPU and the rest of the C program executes on a CPU. CUDA also assumes that both the host and the device maintain their own separate memory spaces in RAM and VRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

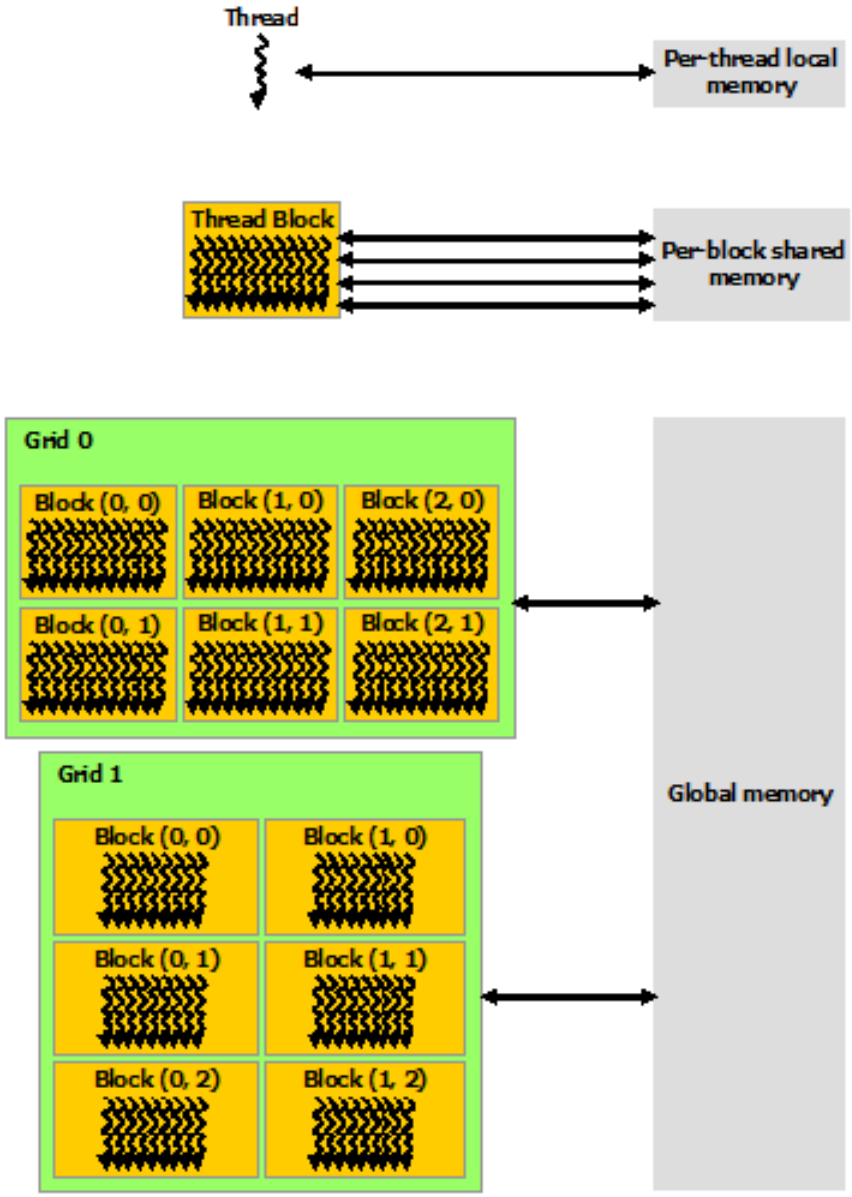


Figure 2.10: Different kinds of memory that a thread can access. Diagram obtained from CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

This implicitly states that in most cases, serial code executes on the host (the CPU), while parallel code executes on the device (the GPU), as can be seen in Figure 2.11. The main idea then, is to use the GPU to do all the parallelizable sections of the program, while using the CPU only to launch kernels, synchronizing its inputs and outputs and execute the portions of the algorithm that can not be parallelized. With this approach, we can ensure gains in the performance of the algorithms, especially if they handle large amounts of data that can be processed in parallel.

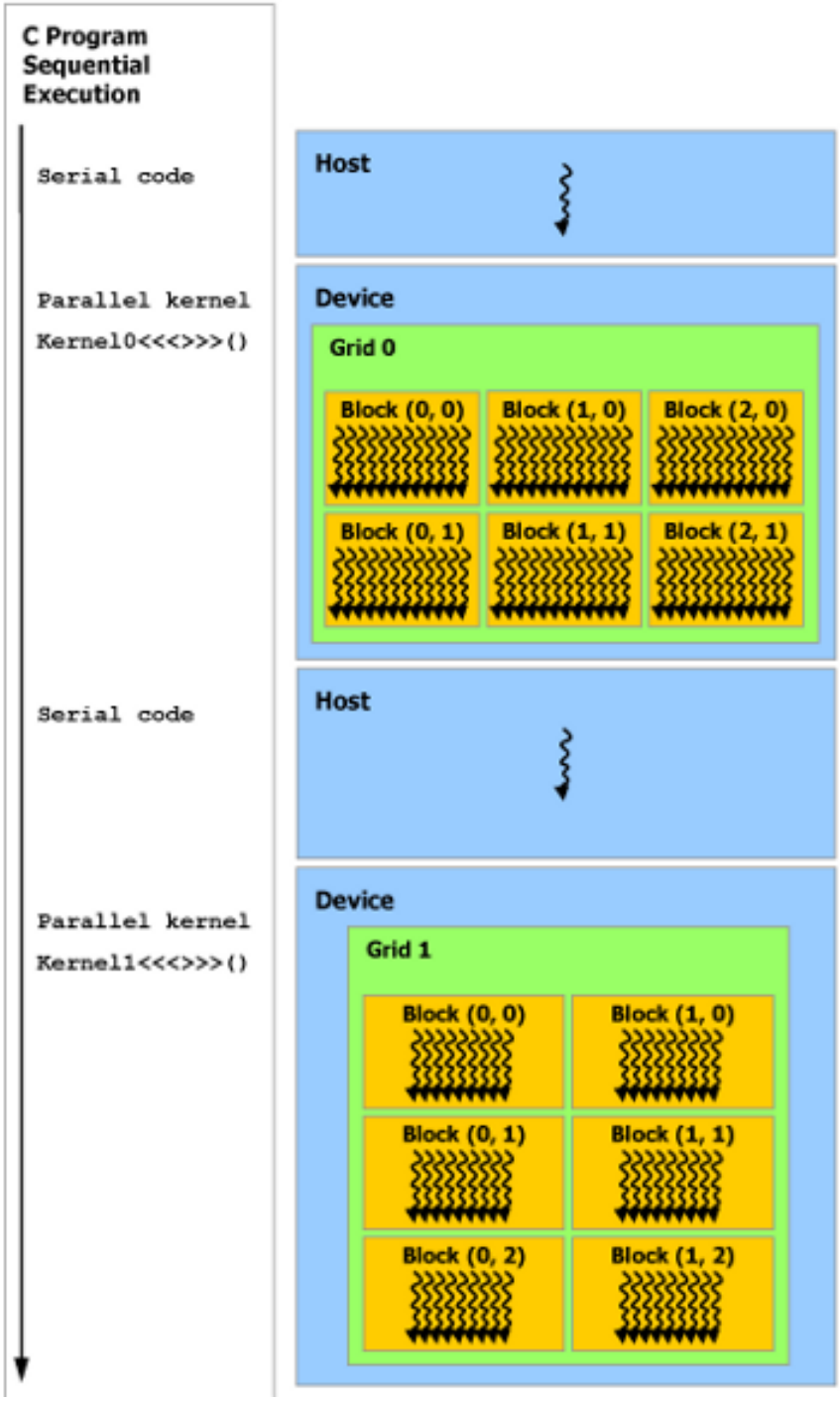


Figure 2.11: Example of a Heterogeneous program flow execution. Diagram obtained from CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

2.3 Cleap Library

Cleap[17] is a library, developed by Cristobal Navarro, which contains algorithms to process geometric meshes in parallel on the GPU, in particular, parallel Delaunay triangulation based on edge-flips[18] and parallel Taubin smoothing[16].

This library has a very interesting Data Structure, based on the Dynamic Render Mesh[24]. The Cleap data structure consists of an explicit representation of vertices, triangles and edges of a geometric mesh, as can be seen on Figure 2.12. The vertices are represented as a 1D array in the same way as the OpenGL VBO (Vertex Buffer Object), in which each position stores a vector with the vertex coordinates (x, y, z) in the Euclidean space. Triangles are represented as a 1D array containing indexes to the array of vertices, where 3 consecutive vertex references define a triangle. Finally, the edges are represented as a 1D arrangement of structures, each structures contains a pair of indices to the vertices array (the ones that are its endpoints) and another pair of indices to the triangles array (the ones that uses this edge as a border). In this way, the same data structure can be used for visualization and rendering in OpenGL, and calculations in CUDA.

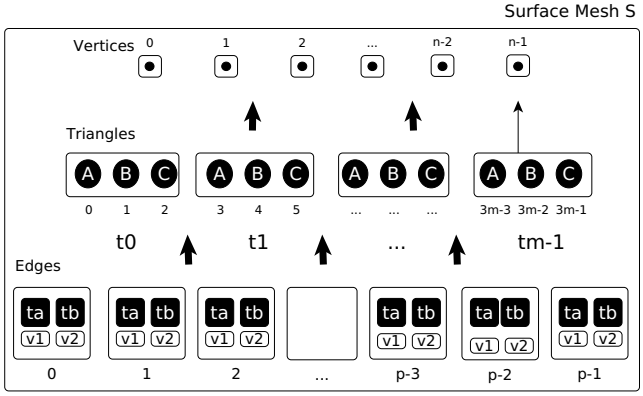


Figure 2.12: Cleap Data Structure Diagram. Diagram by Cristobal Navarro

With this data structure, an edge can know its endpoints directly, checking the references to its vertices, or indirectly, by checking its adjacent triangles, and then the vertices (an example can be seen on Figures 2.13 and 2.14. This redundant information allows to check the integrity and consistency of the mesh after an operation that modifies its connectivity (either by the addition or removal of elements), because it can check that its references are consistent or not, and if they differ, then something on the data associated to the edge is not right and must be fixed. This can be checked in parallel, assigning a thread to each edge, vertex or triangle that we want to check, and report if something is wrong, writing a flag on the global GPU memory.

Because one the goals is to extend Cleap, to add a parallel mesh simplification algorithm, this data structure may be useful, as has been proven in the past, when Cleap was extended to support a parallel smoothing algorithm[16]. So we expect to use this structure with almost no modifications, only adding support data structures if it is necessary.

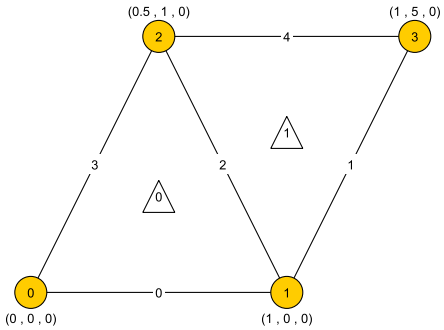
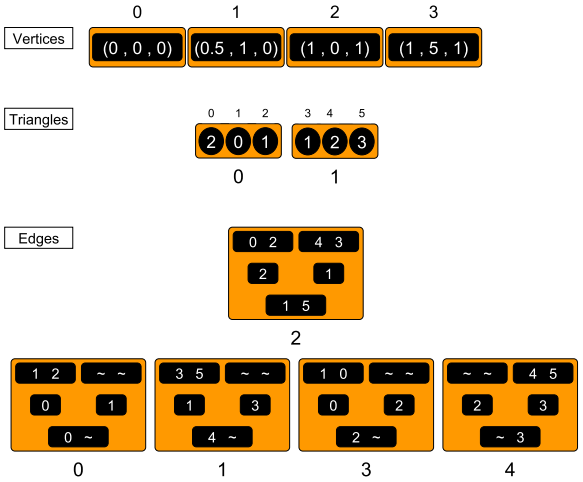
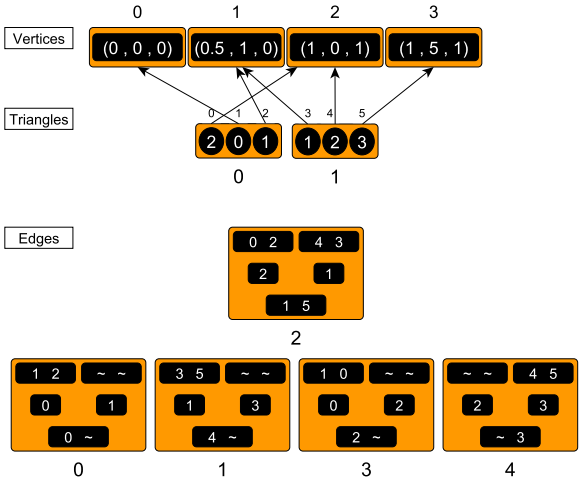


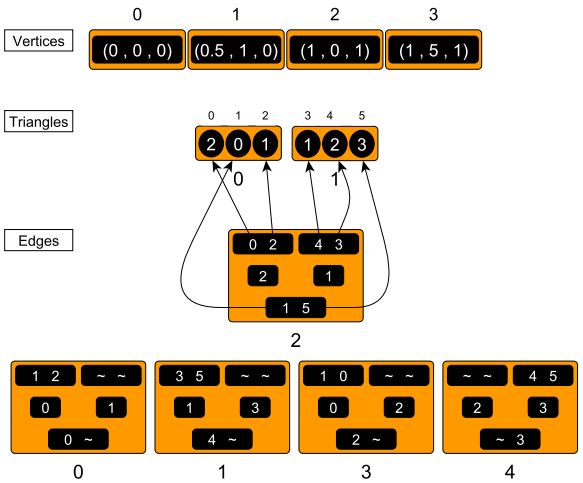
Figure 2.13: A mesh with 4 vertices, 2 triangles and 5 edges.



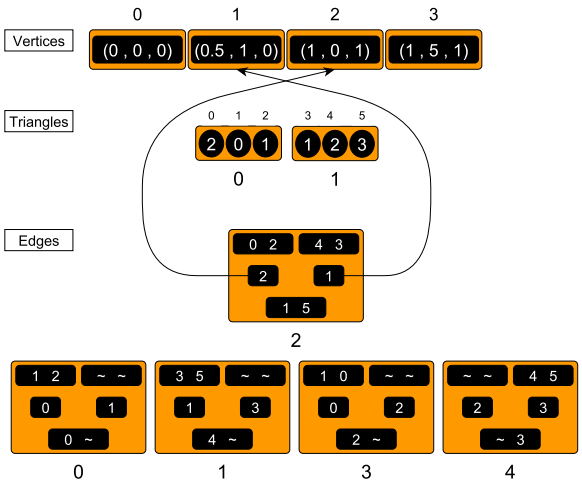
(a) Data structure that holds the information of the mesh in Figure 2.13



(b) References from the triangles to its vertices.



(c) References from the edge to its adjacent triangles.



(d) References from the edge to its vertices.

Figure 2.14: Examples of the references that the Cleap data structure holds.

2.4 Mesh File Format

Finally, it is necessary to choose a file format, to load meshes into Cleap, an save the output of its algorithms into a file. Actually, Cleap supports the .OFF file format for input and output, but we have decided to add support for an additional format, .PLY.

The principal reason is that the .OFF format only saves the information about the edges implicitly (via the faces information), but some simplification algorithms works by modifying the edges of the mesh, so it would be useful to have edges and their information explicitly represented on a file, to visualize partial results of the algorithm, and use that as a help to make decisions about the design and implementation of our algorithm (for example, to know which edges may be deleted under an edge-collapse scheme).

In the next subsections, we will explain in more detail the .OFF and .PLY file formats.

2.4.1 .OFF Format

Object File Format (OFF) is a geometry definition file format containing the description of the composing polygons of the 3D object. This format consist of:

- The first line contains the number of vertices (N), the number of faces (F) and the number of edges (E, that can be ignored)
- Each of the next N lines contains the x, y and z coordinates of each vertex.
- Each of the next F lines contains the number of vertices of that face, followed by a list of vertices indexes that compose this face (the count of the vertex index stars from zero).

An example file of a mesh in this format can be seen in Listing 2.1. In this format, the vertices are explicitly represented via its coordinates in the Euclidean space, and the faces are represented as a list of vertices. On the other hand, the edges are not explicitly represented, and must be deducted from the connections between vertices on the faces if we want to explicitly represent them in a data structure.

```
OFF                 # OFF header
5 4 8               # Number of vertices, faces and edges
0.0   0.0   0.0     # vertex 0
1.0   0.0   0.0     # vertex 1
1.0   1.0   0.0     # vertex 2
0.0   1.0   0.0     # vertex 3
0.5   0.5   0.0     # vertex 4
3   0   1   4       # face 0
3   1   2   4       # face 1
3   2   3   4       # face 2
3   3   0   4       # face 3
```

Listing 2.1: .OFF file that represents Figure 2.15

Additionally, the normal or orientation of the faces can be determined using the *right-hand rule* on the sequence of vertices of each face.

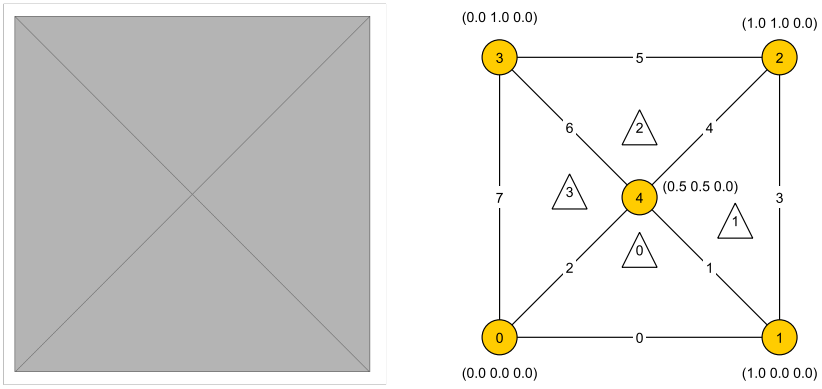
2.4.2 .PLY Format

Polygon File Format (also known as Stanford Triangle Format) is a file format designed to store 3D data from 3D scanners. This format supports a relatively simple description of an object as a list of vertices, faces and edges (similar to the .OFF format). Additionally, a variety of properties can be explicitly stored, including color, transparency, face normals, textures, etc. This format consists of:

- The first line contains the format data type (ASCII or binary).
- The next line, with the *element* keyword, introduces a description (meta-data) of how an element is stored and how many properties each one has. For example, a vertex element is represented by 3 real coordinates.
- When all the elements and its respective properties are listed, the keyword *end_header* is placed.
- Each of the next N lines contains a list of vertices with its (x, y, z) coordinates and properties.
- Each of the next F lines contains the number of vertices of that face, followed by a sequence of the indexes to the vertices (starting from zero) that represent that face, and its properties.
- Each of the next E lines contains two vertex indices (starting from zero) representing the endpoints of the edge, and its properties.

An example file of the same mesh on Listing 2.1, but in .PLY format, can be seen in Listing 2.2. In this format, the vertices are explicitly represented via its coordinates in the Euclidean space, the faces are represented as a list of vertices, and the edges are represented as a pair of vertex indices. Also additional properties (like the color of the edges for this example) can be explicitly declared for each element in particular.

Note that both files in Listings 2.1 and 2.2 leads to the same representation, that can be seen in Figure 2.15, both in rendering and how the information of connectivity may be stored in the data structure of the application.



(a) Render of the object. (b) Logical detail.

Figure 2.15: A simple geometric object render and representation.

```
PLY                                     # PLY Header
format ascii 1.0                        # format specification
element vertex 5                        # meta-data of vertices
property float x                          # x-coordinate
property float y                          # y-coordinate
property float z                          # z-coordinate
element face 4                          # meta-data of faces
property list uchar int vertex_index      # list of vertices
element edge 8                          # meta-data of edges
property int vertex1                      # vertex 1
property int vertex2                      # vertex 2
property uchar red_color                  # edge color
property uchar green_color                # in RGB format
property uchar blue_color
end_header                              # end of meta-data
0.0   0.0   0.0                         # vertex 0
1.0   0.0   0.0                         # vertex 1
1.0   1.0   0.0                         # vertex 2
0.0   1.0   0.0                         # vertex 3
0.5   0.5   0.0                         # vertex 4
3   0   1   4                           # face 0
3   1   2   4                           # face 1
3   2   3   4                           # face 2
3   3   0   4                           # face 3
0   1   0   0   255                     # edge 0
1   4   0   0   255                     # edge 1
4   0   0   0   255                     # edge 2
1   2   0   0   255                     # edge 3
2   4   0   0   255                     # edge 4
2   3   0   0   255                     # edge 5
3   4   0   0   255                     # edge 6
3   0   0   0   255                     # edge 7
```

Listing 2.2: .PLY file that represents Figure 2.15

Chapter 3

Literature Review

In this chapter, we will show a review over various research publications related to mesh simplification, with emphasis on:

- The mesh simplification technique used.
- The criterion or metric used to determine which elements will be collapsed or simplified.
- If the algorithm is sequential or parallel, and if it is parallel, review the mutual-exclusion condition used.

In Section 3.1, we describe the most important simplification techniques found in the review. Then, in Section 3.2, we describe the most relevant simplification metrics or criterion. Finally, in Section 3.3, we describe each of the algorithms, including their assumptions, steps, what kind of data structures they use, and any information that can be useful for our purposes.

3.1 Simplification Techniques

Mesh simplification techniques consist of removing mesh elements. The different techniques varies on which elements they focus on the removal, or if they try or not to preserve some constrains, such as the original geometry or topology of the mesh. In the next subsections we describe the most relevant simplification techniques.

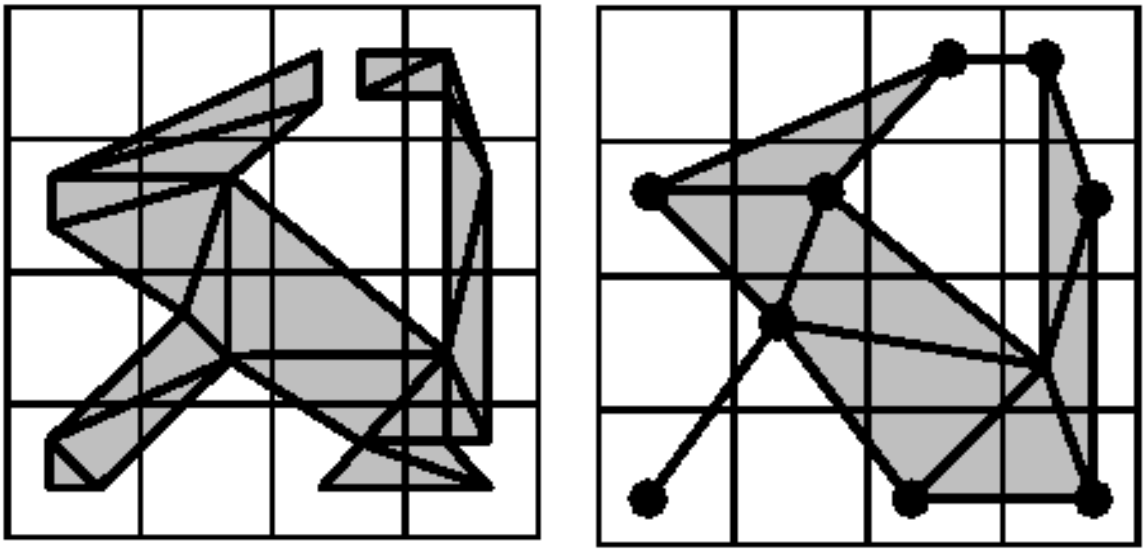
3.1.1 Vertex-Clustering

The Vertex-Clustering[20][21] technique consists of massively deleting a group of mesh elements, leaving only a vertex representing the deleted elements of that group. The steps of the technique are as follows:

First, if the faces are not exclusively triangles, then the non triangular ones are decomposed into triangles. Then, using a criterion, like geometric proximity or if a vertex falls on a certain bound or region, all the vertices are clustered and grouped in *cells*. Then, for each *cell*, a representative vertex is computed. This representative vertex may be defined as the center of mass of all the vertices in the *cell*.

Then, for each triangle in the original mesh, whose three vertices map to the same representative vertex, the triangle degenerates into a single vertex (the representative one). If two of the vertices maps to the same representative vertex, then the triangle degenerates into an edge (effectively joining two *cells* via its representative vertex).

Finally, a search is performed to delete all possible duplicated elements of the mesh (usually elements from distinct cells that degenerate to the same simplified element). Figure 3.1, shows an example of how a mesh is simplified under this scheme. Note how the vertices under the same *cell*/square, converge to the same representative vertex after clustering, who are connected to representative vertices of other areas if an edge previously existed between the two areas.



(a) Before Clustering. (b) After Clustering.

Figure 3.1: Mesh before and after a vertex decimation is performed. Pictures taken from StackExchange `https://computergraphics.stackexchange.com/questions/3723/exact-down-sampling-and-re-triangulation-of-mesh-datanon-convex`.

3.1.2 Vertex-Decimation

The Vertex-Decimation[22][5][4] technique consists of selecting a vertex in the mesh, and then removing it, along with all triangles that are incident to the deleted vertex. Then, the resulting hole is patched by forming a local triangulation. This process repeats until some termination condition is met, usually, a percent reduction of the original mesh or a maximum number of vertex or triangles decimated is reached.

First, each vertex is characterized and weighted in function of a certain criterion (typically, how much its decimation affects the local topology and geometry). This produces an ordered list of vertices able to be deleted (sorted by their influence over the mesh quality). In the next step, a vertex is selected and deleted, as explained before. Then the hole is re-triangulated using the vertices that were connected to the deleted vertex (using a recursive loop triangulation scheme). This step is repeated until the termination criteria is reached.

An example of how this scheme works can be seen in Figure 3.2, where the central vertex of the mesh is deleted, with all the edges connected to it. Then, the hole is re-triangulated, re-connecting the vertices on that zone (the choice of which vertices are re-connected is decided by the particular algorithm or implementation).

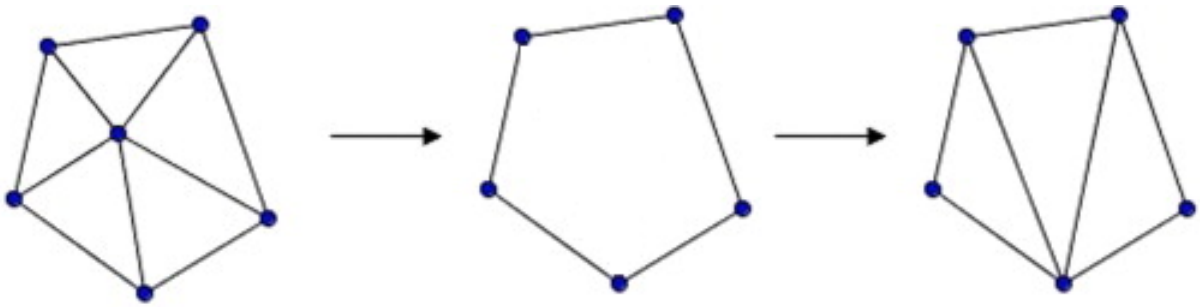


Figure 3.2: Vertex Decimation steps. Image taken from `https://www.sciencedirect.com/science/article/pii/S009830041100313X`

3.1.3 Vertex-Contraction

The Vertex-Contraction[7] technique consists of contracting two vertices of the mesh, that are no necessarily directly connected by an edge. First, all possible vertex-pair candidates are evaluated by a metric or criteria, which usually takes into account if the distance between the vertex that conforms the pair does not exceed a predefined distance threshold and/or if that pair was actually connected by an edge instead.

Then, a pair is selected (usually the one, according to the metric evaluation, that does not affect too much the geometry and topology of the mesh when contracted). Let us call this pair (v_1, v_2). Both v_1 and v_2 are moved to a new position v'. Then all edges connected to v_2 are connected to v_1, and finally, v_2 and all possible edges or faces which may become degenerate or inconsistent are deleted. This process is repeated until the termination criteria is reached.

Figure 3.3 shows the key step of the Vertex-Contraction procedure, that can join two sectors of the mesh that were previously separated.

One of the benefits of this technique is the ability to join previously unconnected regions of the mesh and reduces the sensitivity of the original mesh connectivity. This can be useful in some applications, such as rendering, where topology may be less important than the overall shape of the mesh.

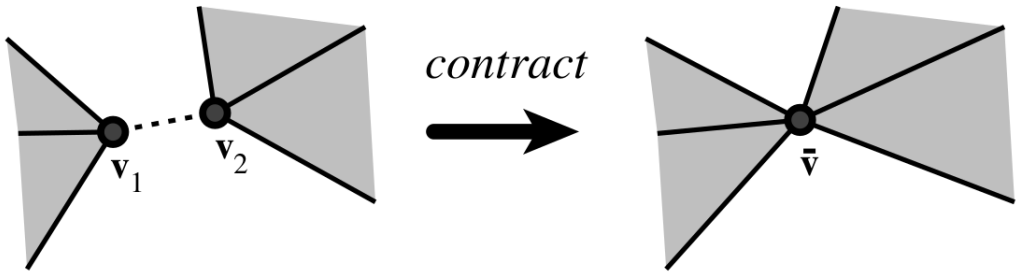


Figure 3.3: Vertex Contraction step. Diagram taken from *Surface Simplification Using Quadric Error Metrics*[8], Figure 2.

On the other hand, the threshold selection is based on the assumption that, in a good approximation, vertices do not move far from their original positions. Also notice if the threshold is high, widely separated portions of the mesh could be connected by pair selection, which could be undesirable (because it may change the overall shape of the mesh) and could produce $O(N^2)$ vertex-pairs that need to be evaluated and impact on run time of the algorithms.

3.1.4 Edge-Collapse

The Edge-Collapse[13][9][11][19] technique consists of removing an edge of the mesh, while simultaneously merging the vertices that were joined by this edge into a single vertex, who inherits the connectivity of both vertices (its incident triangles and connected edges). This also removes the triangles that use the removed edge as one of its sides (typically one or two).

First, all edges are evaluated by a metric or criterion. Then, an edge is selected (usually the one, according to the metric evaluation, that does not affect too much the geometry and topology of the mesh when deleted), and let us call it e $= (v_1, v_2)$. Both v_1 and v_2 are moved to a new position v'. This new position is computed trying to find the best position in the surrounding area of v_1 and v_2 that minimizes some kind of error function or maximizes a quality criterion over the mesh. Usually this position lies on a point on the edge that connected $v1$ and $v2$.

Finally, all edges connected to v_1 and v_2 are connected to v', and the two triangles that uses e as one of their sides is removed as well, along with v_1 and v_2.

This technique can also be viewed as a particular case of Vertex-Contraction, in which only vertex-pairs connected by an edge are allowed to be contracted.

One particular case of this technique is known as *half edge-collapse*[14], in which the edge, can only collapse onto one of its vertices (the only allowed positions for v' are v_1 or v_2). This has the advantage that there are only two possible outcomes for the edge-collapse to evaluate, which can save processing time, but has the disadvantage that the two outcomes may be sub-optimal, leading to a simplification that may not be as good as the one generated by selecting the best position available.

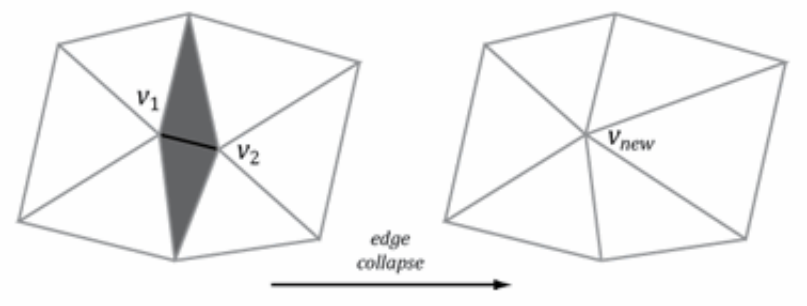


Figure 3.4: Edge-collapse step, Image obtained from Intel `https://software.intel.com/en-us/articles/3d-modeling-and-parallel-mesh-simplification`

3.2 Simplification Metrics

The other important aspect of a mesh simplification scheme is how the elements are selected or prioritized among the others to be deleted. In the next sub-sections we review the most important simplification metrics encountered.

3.2.1 Edge length/Angle

One of the most simple measures to decide which edge will be collapsed first, is its length[15]. Edges with the shortest length are selected to be collapsed first, under the assumption that they do not have a lot of information about the mesh (its geometry and topology), or that shorter edges tend to be associated with triangles that have a small angle (opposite to the edge), which produces two edges that are almost co-linear and do not contribute additional information, and its deletion does not change the overall shape of the mesh.

A simple upgrade of this metric is to weigh the length of the edge by a curvature term. This curvature term may be a simple calculation, as the dot product between the two triangle faces that shares the edge, or more complicated expressions, that considerate the vicinity of faces around the vertices of the edge. The idea behind this curvature term is to weigh the local geometry, trying to not delete key edges that may have implicit information about the shape of the mesh (like mountains or borders or another important shapes).

This criteria is similar to the edge-angle criterion, in which the edges opposite to a very acute angle in sliver triangles, like the ones we saw in Figure 2.4, are candidates to be deleted.

3.2.2 *Quadric* error metric

This criterion is an attempt to estimate or have a notion of the *cost* of an edge contraction (and more generally, an arbitrary pair of vertex contraction)[6][8][7]. To define this cost, this criterion characterizes the geometric error at each vertex. To do this, each vertex is characterized as the solution of the intersection of a set of planes, where each plane on this set is the generalization of the triangles that meet on that vertex. In Equation 3.1, the error $\Delta(v)$ is defined as the sum of squared distances between the vertex and each of its planes.

$$\Delta(v) = \Delta([v_x\, v_y\, v_z\, 1]^T) = \sum_{p \in planes(v)} (p^T v)^2 \tag{3.1}$$

Where $p = [a\, b\, c\, \mathrm{d}]^T$ represents the plane defined by the equation $ax + by + cz + \mathrm{d} = 0$ with $a^2 + b^2 + c^2 = 1$, and v is a vector with the coordinates of the vertex in homogeneous coordinates. This equation can be rewritten as:

$$\begin{aligned} \Delta(v) &= \sum_{p \in planes(v)} (p^T v)(p^T v) \\ &= \sum_{p \in planes(v)} v^T (pp^T) v \\ &= v^T \left(\sum_{p \in planes(v)} K_p \right) v \end{aligned} \tag{3.2}$$

Where K_p is the matrix:

$$K_p = pp^T = \begin{bmatrix} a^2 & ab & ac & a\mathrm{d} \\ ab & b^2 & bc & b\mathrm{d} \\ ac & bc & c^2 & c\mathrm{d} \\ a\mathrm{d} & b\mathrm{d} & c\mathrm{d} & \mathrm{d}^2 \end{bmatrix} \tag{3.3}$$

The matrix on Equation 3.3 is called the *fundamental error quadric*, and can be used to find the squared distance of any point in space to the plane p. With this, an entire set of planes can be represented by a single matrix Q, which is the sum of all the *fundamental quadrics* of the planes incident to the vertex v, and the error at vertex $v = [v_x\, v_y\, v_z\, 1]^T$ can be calculated as:

$$\Delta(v) = v^T Q v \tag{3.4}$$

In Figure 3.5, we can see the model of a bunny mesh, where each vertex has an ellipsoid (in green), representing the measure of the respective quadric. The error associated to a vertex is proportional to the size of the area of the respective quadric ellipsoid. Note that for vertices placed in sharp areas, such as the ears, the ellipsoids are more thin shaped, while in plane areas, such as the forehead, the ellipsoids are more circular shaped.

This means that the first ones are associated with high curvature zones of the mesh and its deletion or merging with other vertex may result in a big error and quadric size, which is not desirable for preserving the original shape of the mesh. On the other hand, the second ones are associated with low curvature zones, tend to be more smaller and are more desirable for the purpose of deletion or merging.

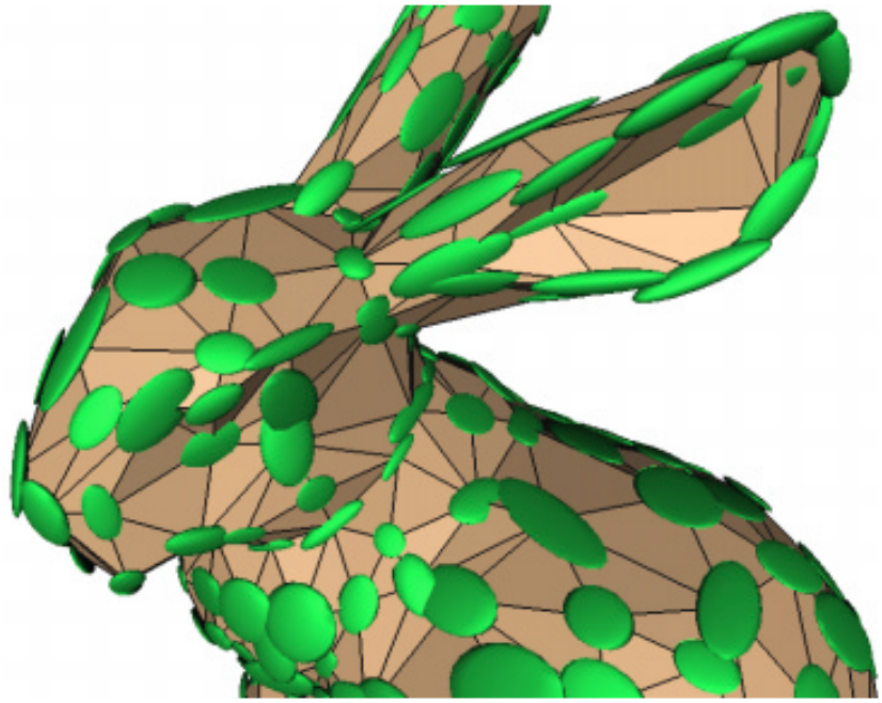


Figure 3.5: A mesh with quadrics represented as ellipsoids (in green). Image taken from *Simplifying Surfaces with Color and Texture using Quadric Error Metrics*[7], Figure 2.

For a given edge collapse or vertex contraction $(v_1, v_2) \rightarrow v'$, the new matrix Q' which approximates the error at v', is the sum of the Q matrix of each of this vertices: $Q' = Q_1 + Q_2$.

This criterion is useful to minimize the overall error over the mesh after successive simplifications, with a fairly simple calculation, giving an equilibrium between computation complexity and quality of the error estimation.

3.2.3 Volume Error

This criterion[10] corresponds to assigning an error zone volume (typically, a sphere) to each original vertex of the mesh. When a vertex is removed, the error zone is activated. Then, the criterion is that all active error zones have to be intersected by at least one face of the simplified mesh at all times, so if an element that is a candidate to be removed produces that the resulting mesh does not intersect an active error zone, then this element cannot be removed and the simplification is discarded. This criteria is useful when the mesh simplification needs to take into account mechanical properties of the mesh (for example, the joints of a robotic arm), in which the error zones of this key parts of the mesh can be adjusted to tolerate less distortion in these parts.

3.2.4 Energy Function

Another criterion is to view the simplification problem as an energy optimization problem[11]. In this approach, we want to minimize an energy function that represents the mesh, trying to obtain a mesh that is similar to the original mesh but with a small number of vertices. The energy function for a given mesh is:

$$E(K,V) = E_{\mathrm{dis}t}(K,V) + E_{rep}(K) + E_{spring}(K,V) \tag{3.5}$$

Where V is the set of vertices in the mesh, K is a set of subsets of vertices that represents the connectivity between vertices, edges and faces. The first term $E_{\mathrm{dis}t}$, is the distance energy, calculated as the sum of squared distances between each vertex of the mesh, and the mesh (See Equation 3.6). This term helps to measure if the actual mesh is a good fit to the original mesh. $\phi_v(x_{\mathrm{i}})$ is the minimum distance between the mesh and a particular vertex v_{i} on it.

$$E_{\mathrm{dis}t}(K,V) = \sum_{\mathrm{i}=1}^{n} ||x_{\mathrm{i}} - \phi_v(x_{\mathrm{i}})||^2 \tag{3.6}$$

The second term E_{rep} is the representation energy, calculated as a fraction of the number of vertices in the mesh. This term penalizes meshes with a large number of vertices.

$$E_{rep}(K) = C_{rep}|V| \qquad C_{rep} \in \left[10^{-4}, 10^{-5}\right] \tag{3.7}$$

The third term E_{spring} is the spring energy, which adds to each edge on the mesh, an elastic tolerance constant k. This term acts as a regularizing term that guides the optimization to a desirable local minimum, and whose magnitude decreases as the mesh converges to a solution.

$$E_{spring}(K,V) = \sum_{\{j,k\}\in K} \kappa||v_j - v_k||^2 \qquad \kappa \in \left[10^{-5}, 10^{-8}\right] \tag{3.8}$$

This metric works by minimizing the result of the Equation 3.5, by using operations that modifies the elements of the mesh (and thus, affecting the value of each term of the Equation). Unlike the other ones, this metric may lead to simplified meshes that are not exclusively generated by simplification operations, which can lead to better results in some cases (we will see this in more detail in the review of the algorithm that uses this metric, on Section 3.3.7).

3.3 Algorithms Review

In this section we summarize the most relevant algorithms we have encountered over the course of this research work, focused on:

- If it is a sequential or a parallel algorithm.
- The criterion used to choose which part of the mesh will be simplified first.
- The data structures used to keep track of the mesh.
- Mutual exclusion mechanism for parallel solutions.
- How they measure the quality of their algorithm.

3.3.1 Shontz-Nistor

The Shontz-Nistor algorithm[23] proposes 3 sub-routines (1 CPU-sequential based and 2 GPU-parallel based), which can work in tandem to simplify a mesh. Combining the CPU routine with each GPU routine, they produce two different mesh simplification algorithms. All routines rely on the edge-collapse simplification technique and both algorithms work with the principle of allocating a portion of the mesh to the CPU and the rest to the GPU.

To assure that the simplification process is performed uniformly on the mesh, each element who has participated in an edge-collapse is marked, either because it has been deleted, or is a neighbour to a deleted element (we will define later the concept of neighborhood). A marked element cannot take part in subsequent edge-collapses until all elements of the mesh have been marked or no more elements can be marked, which signals the end of the overall simplification iteration. This ensures, according to Shontz, that the edge-collapse operation is performed uniformly across all the mesh, and no area is over-simplified or under-simplified. Also, this rule acts as a quality metric for the mesh.

As can be seen in Figure 3.6, the elements that need to be marked when an edge collapse over edge (v_1, v_2) (in red) is performed, are all the faces which contains either one of the vertices of the candidate edge v_1 or v_2, or one of the vertices opposite to the edge, v_3 or v_4. All the elements marked as affected are shaded in gray, and we call this group of marked elements, the Shontz-Nistor vicinity.

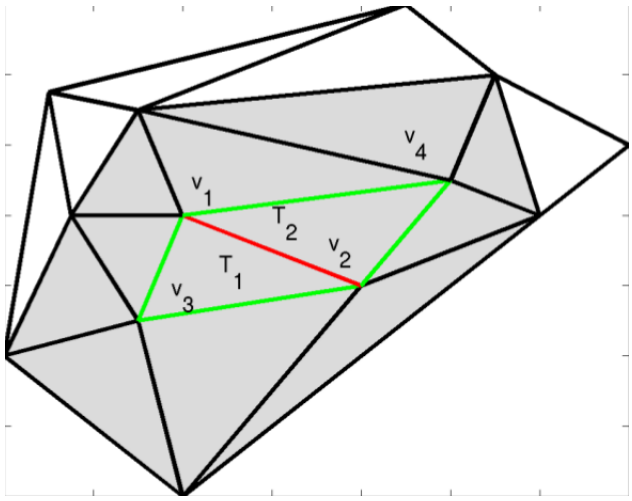


Figure 3.6: Graphical description of the Shontz-Nistor Vicinity, Image taken from *CPU-GPU Algorithms for Triangular Surface Mesh Simplification*[23], Figure 2.

Figure 3.7 shows the description of the CPU routine, which is a simple edge-collapse sequential algorithm implementation. Each face is examined to see if it is affected, and if an element that is not affected is found, then an edge-collapse is performed if all elements in its vicinity are not marked as affected.

For the GPU algorithms, a mechanism of locks over the vertex of the mesh is used, to ensure that updates over the mesh on zones that are close to each other, can be performed simultaneously without producing inconsistencies for concurrent updates. This lock mechanism is implemented as an array of bits/flags, on global GPU memory, so that every parallel thread can read and write on it.

The first GPU algorithm described in Figure 3.8 is the GPU marking routine, which is a naive adaptation of the CPU sequential algorithm. The search for the elements assigned to the GPU is performed one at a time, and when a candidate is found, the elements affected by this eventual edge-collapse are read and marked concurrently with the GPU threads.

The second GPU algorithm described in Figure 3.9 is the GPU inverse reduction routine, which uses the capabilities of the GPU on a higher degree. It starts by concurrently examining all elements assigned to the GPU with a fixed number of threads. When a GPU-thread finds a non-marked element the thread attempts to lock each vertex that belongs to its vicinity, and if it finds a vertex has already been locked, it means that another thread locked it and this thread cannot continue on this vicinity, so it releases all previously locked-vertex and starts the search of a new non-marked element. If a thread successfully locks all vertices, then this means that no other thread has locked the vicinity of this element and it can be simplified without concurrency updating issues.

Algorithm 1 The CPU Edge-Collapse Simplification Algorithm

```
function MARK-AS-AFFECTED(element)
    for all v ∈ element do
        affected[v] ← true
    end for
end function

function MARK-AS-COLLAPSED((v1, v2))
    for all T ∈ elements ⊃ {v1, v2} do
        collapsed[T] ← true
    end for
end function

function CPU-SIMPLIFY(elements, vertices)
    for all T = (v1, v2, v3) ∈ elements do
        if T ∈N(affected elements) then
            mark-as-affected(T)
        else
            collapse((v1, v2))
            mark-as-collapsed((v1, v2))
        end if
    end for
end function
```

Figure 3.7: Shontz-Nistor CPU algorithm. Scheme taken from *CPU-GPU Algorithms for Triangular Surface Mesh Simplification*[23], Algorithm 1.

Algorithm 2 The GPU Marking Simplification Algorithm

```
function GPU-MARK(mark, elements, vertices)
    for all T ∈ N(mark) do GPU thread T : mark-as-affected(T)
    end for
end function

function GPU-MARK-SIMPLIFY(elements, vertices)
    for all T = (v1, v2, v3) ∈ elements do
        if not marked(T) then
            collapse((v1, v2))
            mark-as-collapsed((v1, v2))
            GPU-Mark(T)
        end if
    end for
end function
```

Figure 3.8: Shontz-Nistor GPU marking algorithm. Scheme taken from *CPU-GPU Algorithms for Triangular Surface Mesh Simplification*[23], Algorithm 2.

Algorithm 3 The GPU Inverse Reduction Simplification Algorithm

```
function GPU-Simp-Try(target = (v0, v1, v2), elements, vertices)
    if affected(target) then
        return
    end if
    if test-and-set(collapsed[v0]) = 0 then
        if test-and-set(collapsed[v1]) = 0 then
            if test-and-set(collapsed[v2]) = 0 then
                collapse((v0, v1))
                GPU-Mark(target)
            end if
          collapsed[v1] ← 0
          collapsed[v0] ← 0
        end if
        collapsed[v0] ← 0
    end if
end function

function GPU-IR-Simplify(elements, vertices)
     i = |elements|
    while i ≥ 1 do
        if threadid mod i = 0 then
          GPU-Simp-Try(elements[threadid])
        end if
        i = i div 2
    end while
end function
```

Figure 3.9: Shontz-Nistor GPU inverse reduction algorithm. Scheme taken from *CPU-GPU Algorithms for Triangular Surface Mesh Simplification*[23], Algorithm 3.

3.3.2 Garland-Heckbert

The Garland-Heckbert algorithm[7][8] is a sequential algorithm based on iterative contraction of vertex pairs (a generalization of edge-collapse). A set of valid pairs are chosen at initialization time, under the assumption that vertices do not move far from their original positions in a good approximation. A pair of vertex (v_1, v_2) is a valid pair for contraction if either (v_1, v_2) is and edge, or $||v_1 - v_2|| \leq t$, where t is a threshold parameter, where $t \in [0, 1.5]$ ($t = 0$ means an edge-collapse algorithm).

The cost of the contraction is measured using Quadric Error Metrics, where the error associated to a given vertex is $\Delta(v) = v^T Q_v v$. For a given contraction $(v_1, v_2) \rightarrow v'$, it is necessary to derive a new error matrix $Q_{v'}$ and a new position for v'. For the new error matrix, they have chosen the additive rule $Q_{v'} = Q_1 + Q_2$, where Q_1 and Q_2 are the error matrix associated to v_1 and v_2 respectively. For the new position, they try to find a position that minimizes $\Delta(v)$, and because $\Delta(v)$ is quadratic, finding its minimum is equivalent to solve the matrix system on Equation 3.9:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} v' = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.9}$$

Assuming that this matrix is invertible, then the optimal new position is:

$$v' = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.10}$$

If the matrix on Equation 3.10 is not invertible, then they try to find the best possible position for the vertex along the edge (v_1, v_2), its endpoints v_1, v_2, or using the middle point $(v_1 + v_2)/2$.

The authors represent the mesh using and adjacency graph structure, in which vertices, edges and faces are all explicitly represented and linked together. Additionally, each vertex has associated a list of the valid contraction pairs where it belongs. Also, each edge has special flags to mark them as a discontinuity edge (usually boundary edges for open surfaces, or edges that join two faces with high curvature rate), which are penalized with a high simplification cost, because the elimination of this kind of edges may arise deformations and inconsistencies across the mesh. Finally, a simple face inversion test is performed before doing the contraction (compare the normal vector of all faces involved in the contraction, to see if they perform a flip or not), and discard it if it not pass the test.

With this, the Garland-Heckbert algorithm summarizes as follows:

1. Compute the Q matrices for all the initial vertices.
2. Create a set of valid pairs candidates to be contracted.
3. Compute the optimal contraction target v' for each valid pair.
4. Place all the pairs in a heap, where the pair with the minimum cost is placed at the top.
5. Remove the pair (v_1, v_2) with the minimum cost from the heap, contract this pair and update the cost of all pairs involving v_1 and v_2 (rearranging its order on the heap if necessary)
6. Repeat the last step until target simplification is reached.

An example of the results of this algorithm can be seen in Figure 3.10, which starts with a cow model of ≈ 6000 faces, and the subsequent ones results in models with ≈ 1000, ≈ 500, ≈ 250 and ≈ 60 faces respectively. Note that some features of the original model, such as horns and tail, disappear in the most simplified model.

Figure 3.10: Sequence of simplifications made by the Garland-Heckbert algorithm. Image taken from *Surface Simplification Using Quadric Error Metrics*[8], Figure 4

3.3.3 Papageorgiou-Platis

The Parageorgiou-Platis algorithm[19] is a parallel GPU algorithm based on the edge-collapse operation. The cost of an edge-collapse is measured using Quadric error metrics over the vertices (and the cost of the edge collapse is the sum of quadric error at its vertices) and this cost is ordered using a Bitonic sorting network.

To do edge-collapses in parallel without having concurrency problems, the algorithm finds areas on the mesh that are independent. An area on the mesh is defined by a vertex (who they call the central vertex), and its surrounding vertices (for example: the ones connected by an edge). For two or more areas to be independent of each other, it is required that their central vertices have to be super-independent. Two or more areas are super-independent of each other if the minimal path between their central vertices has at least 3 edges (We are going to describe the concept of super-independent areas in more detail on Section 3.3.5, from which they inspired). To find a group of super-independent areas, they use an adaptation of the greedy serial algorithm that finds a maximal independent set over a graph.

When the super-independent areas are determined, an edge will be collapsed in each of these areas. This edge will be one of the ones who shares the central vertex of its respective area. From all the candidate edges, one among them is chosen, determined by which of them incurs in the smallest simplification cost. In Figure 3.11 we can see how this process works. The super-independent area is marked in red, and all candidates edges are the ones that have the central vertex of this area as one of its endpoints. When one of this edges is selected as the optimal one (the one marked with a red dot), then the edge collapse is performed in this area.

Overall, the steps performed for each edge-collapse in an independent area are:

1. Find the best edge to be collapsed in each independent area.
2. Find the triangles to be deleted in the data structure.
3. Modify the connectivity of the triangles.
4. Remove the triangles and the vertex, updating the data structures accordingly.
5. Update the error metrics.

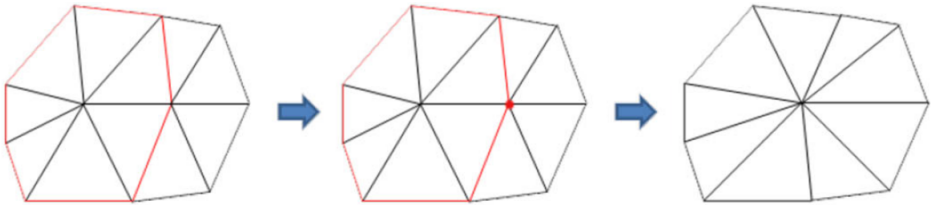


Figure 3.11: Papageorgiou-Platis edge selection process. Image based on a diagram taken from *Triangular mesh simplification on the GPU*[19], Figure 3.

As for the data structures, the mesh is stored as two arrays, one with the vertices (represented as 3 floating point coordinates each), and one with the triangles (three indices to the vertex array for each triangle). Also, for each vertex, they track the triangles who are incident on it, with an extra data structure, which is like a series of unrolled linked list implemented as two arrays (*headers* array and *data* array), as can be seen in Figure 3.12. The *header* array tracks for a given vertex, where starts its incident triangle data on the *data* array, how many incident triangles it has, and if the data continues on another sector of the *data* array. The *data* array has a fixed maximum space to store the incident triangles of a vertex. The *cont.* field on the *header* array, allows a vertex to register more incident triangles that the fixed space, without the need to allocate more space in memory, because it reuses the space from deleted vertices (as can be seen for the case of vertex number 3)

To generate the table on Figure 3.12 and initiate these arrays, they work in parallel on the triangles. For every triangle, a counter associated to each of its vertices is increased atomically in the *header* array, and this triangle is registered as incident to that vertex in the respective zone of the *data* array.

With this, the overall pipeline of the Papageorgiou-Platis algorithm is summarized in Figure 3.13. All steps on the diagram are executed on the GPU, except for those in dashed rectangles, which are the initialization and ending steps.

vert. no.	pos	size	cont	data									
0 :	0	7	/	40346	40365	40383	40477	40525	40760	40762	/	/	/
1 :	10	6	/	33202	35503	35640	35918	34032	34408	/	/	/	/
2 :	20	7	/	31021	31139	31583	30207	30292	30492	30664	/	/	/
3 :	30	10	5	39902	37706	35503	35918	36514	36722	38296	38556	38840	38859
4 :	40	7	/	39294	39333	39429	39593	39759	39920	39998	/	/	/
- :	50	2	/	37018	39998	/	/	/	/	/	/	/	/
6 :	60	5	/	29866	29937	29994	30207	30292	/	/	/	/	/

Figure 3.12: Data Structure used to hold the indices of triangles incident on each vertex. Diagram taken from *Triangular mesh simplification on the GPU* [19], Figure 2.

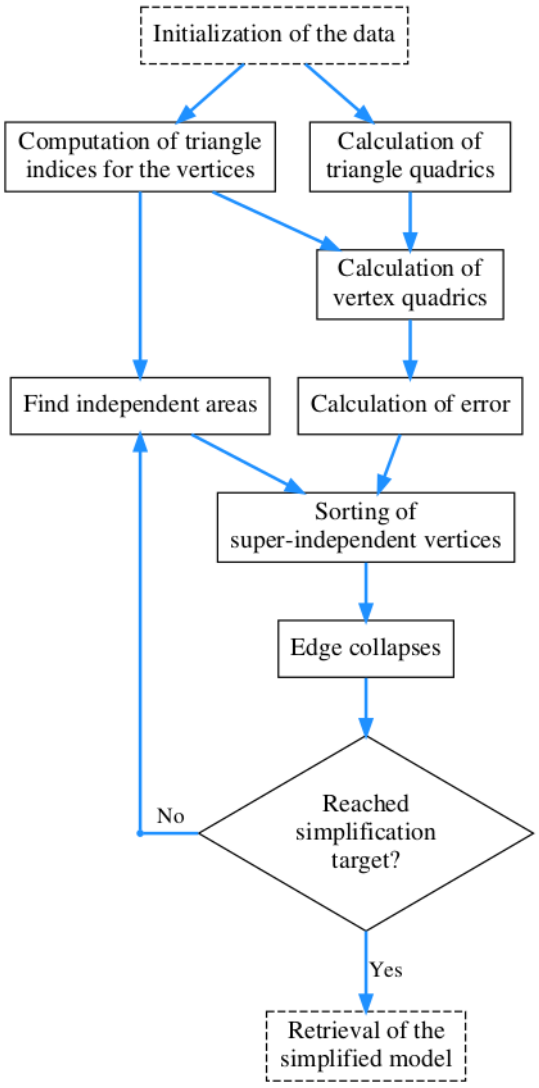


Figure 3.13: Structure of the Papageorgiou-Platis algorithm. Diagram taken from *Triangular mesh simplification on the GPU* [19], Figure 1.

3.3.4 Melax

The Melax algorithm[15], is a CPU sequential algorithm based on applying repeatedly the edge-collapse operation to a mesh, until a polygon reduction percent target is reached. Because at that time (1998), the known metrics were very elaborated, time consuming or difficult to implement, he developed a metric which helps to generate good low-polygon models for game-application in little time relatively to that metrics, which is fairly simple to calculate.

The cost of collapsing an edge is measured with the Edge length metric, which is the length of the edge, multiplied by a curvature term. The curvature term associated to an edge (v_1, v_2) is calculated by comparing the dot products of triangle normal in order to find the triangle adjacent to v_1 that is furthest away from the other triangles that are along (v_1, v_2) , which formally is:

$$cost(v_1, v_2) = ||v_1 - v_2|| \max_{f \in T_{v_1}} \min_{n \in T_{v_1 v_2}} (1 - f_{normal} \cdot n_{normal})/2 \tag{3.11}$$

In Figure 3.14 we can see the result of this algorithm. From a model of a female torso (left), the algorithm generates a simplified model with 20% (center) and 5% (right) of the original faces of the model.

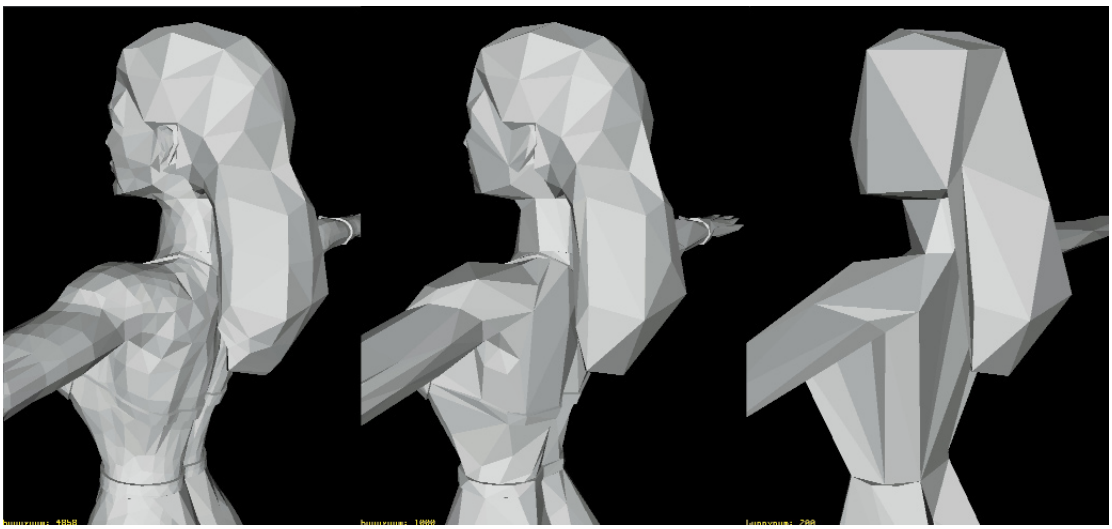


Figure 3.14: Sequence of simplifications of the Melax algorithm. Image taken from *A Simple, Fast, and Effective Polygon Reduction Algorithm*[15], Figure 7.

3.3.5 Franc-Skala

The Franc-Skala algorithm[5] is a CPU parallel algorithm that relies on the vertex-decimation operation. Although the cost of a vertex-decimation is measured, they do not specify how or what kind of metric they use.

To do the vertex-decimation in parallel, they use a mechanism that does not involve defining critical sections and synchronization primitives, but instead it consists of generating an independent set of vertices, that can be decimated at the same time, without producing data races or inconsistencies for concurrent updates on the mesh. This mechanisms is called super-independent set of vertices. Before we define this set, we need to define other concepts.

The area of a vertex is defined as the sum of the surrounding triangle areas. A set of vertices is independent if their areas do not overlap with each other, or in other words, if all vertices in the set do not share an edge with another vertex in the set. A set of vertices is super-independent if they do not intersect or share any vertex, edge or triangle. In more simple words, there has to be a distance of at least 3 edges between vertices on the super-independent set. In Figure 3.15 we can see the areas of v_1, v_2 and v_3 shaded in gray. The three vertices are independent to each other, but only v_1 and v_3 are super-independent of each other.

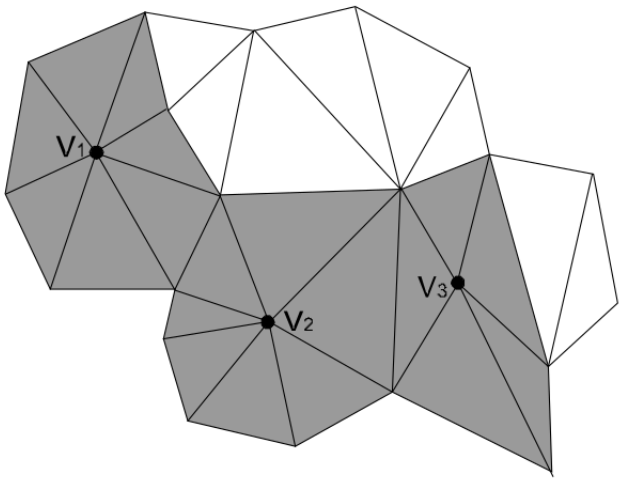


Figure 3.15: Super independent areas concept. Image taken from *Parallel Triangular Mesh Decimation Without Sorting*[5], Figure 1.

To generate this super-independent set, a greedy serial algorithm is used. For each vertex on the mesh:

- All vertices start with an unused flag.
- If a vertex flagged as used is found, then it is skipped.
- If a vertex flagged as unused is found, it is added to the super-independent set if all its neighbouring vertices are flagged as unused. If this happens, then the vertex is added to the super-independent and this vertex and its neighbors are flagged as used.

Then, they state that each vertex in the super-independent set can be decimated independently of the other vertices in the set, without inconsistencies or data races. With this, the Franc-Skala algorithm summarized as follows:

1. Divide the vertices of the mesh into N parts, where N is the number of free processors.
2. Run N threads to evaluate the cost of each vertex. Each thread makes this computation for its own set of vertices.
3. Sort the vertices according to its cost in increasing order.
4. Find a super-independent set of vertices with the greedy algorithm.
5. Divide the independent set into N parts.
6. Run N threads for decimation. Each thread does the decimation in its own group of super-independent vertices.
7. Repeat steps 1-6 until the simplification target is reached.

A variant of this algorithm from the same authors[4], is one that does not do the step 3, which trades quality for execution time.

3.3.6 Lee-Kyung

The Lee-Kyung algorithm[14] is a GPU parallel algorithm that relies on the half-edge-collapse operation and uses Quadric error metrics to measure the cost of each edge.

The authors use a simplification scheme called tree collapsing, which generates a group of half-edge collapses that can be performed in parallel and are dependant of each other (which is known as an edge-tree). In Figure 3.16, we can see a group of edges e_0, e_1, e_2 and e_3 (in red), which collapse into the root vertex r. Because of the inter-dependent collapsing operations, they cannot adjust the optimal vertex position for each performed collapse (and that is why the half-edge-collapse operation is used instead of the edge-collapse). To minimize the quality loss, they add a final step to optimize the vertex positions after all edge-collapses are performed.

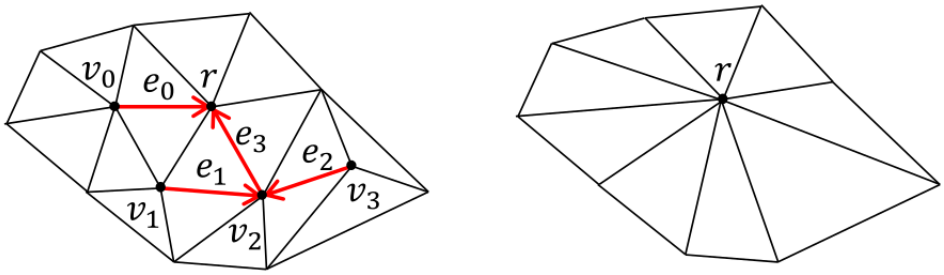


Figure 3.16: Tree collapsing example. Image taken from *Parallel mesh simplification using embedded tree collapsing*[14], Figure 1.

As for the data structures, they use a *doubly connected edge list* (DCEL), also known as half-edge data structure. This structure maintains a record for each edge, vertex and face of the mesh. In particular, as can be seen in Figure 3.17, each edge is handled as a two half-edges, which has a pointer to the face that it is beside it, pointers to the previous (*prev(e)*) and next (*next(e)*) half-edge that belongs to the same face, and a pointer to the twin half-edge (*twin(e)*) that connects to the other face that is adjacent to the edge. This structure is useful to traverse the mesh via *prev(e)*, *next(e)* and *twin(e)* operations.

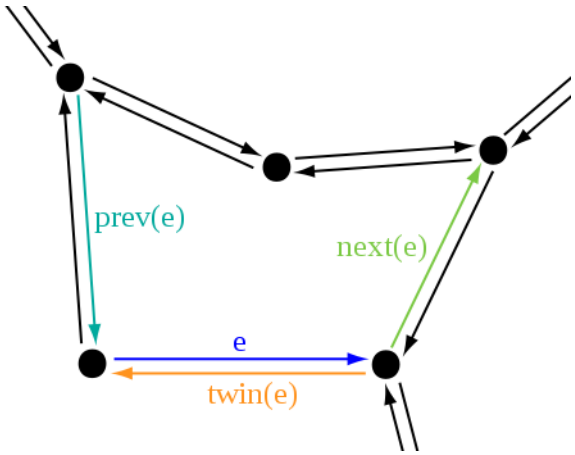


Figure 3.17: DCEL data structure scheme. Image taken from Wikipedia `https://en.wikipedia.org/wiki/File:Dcel-halfedge-connectivity.svg`.

The edge-trees are independently selected over the vertices in parallel, in which the half-edge with the smallest error linked is selected for each vertex in parallel (which are the roots of each tree). Then another half-edge is selected and added to each tree, until a threshold is reached. The quantity of elements in each tree is controlled by an user-specified parameter τ, which bounds the accumulated quadric error of the root vertex of the tree (the sum of the error associated to each half-edge that belong to the root vertex). With this, the idea is to add half-edges to the tree, until the threshold τ is reached, while discarding half-edges whose error is greater than τ.

Because the accumulated error for each tree can be larger than τ in some cases, a tree-split phase is needed, in which the accumulated error is counted in a bottom-up way (from the leafs edges to the root vertex). If at some point, the accumulated error at some vertex is greater than τ, then its children are pruned from the root vertex and a new tree is created from the pruned elements.

Instead of edge-locking to prevent race conditions, they decompose the edge-collapsing into two steps:

1. Record merged edges resulting from edge-collapse in a table.
2. Update the references in the data structure using the information of the table.

Both task are performed on the GPU, in which each thread process a half-edge update. The update to be performed is to reconnect the half-edges to their new *twins*.

Triangle flipping is prevented by doing an inversion test to the triangles that will survive after simplification, with a simulation over them, and the simplification over that tree is discarded if an inversion is detected. On the other hand, topological changes are prevented by testing a link condition over the edges, which determines if an edge can be safely collapsed without changing the overall topology over the mesh (the proof can be seen in the Lee-Kyung article [14]).

With all this, the algorithm summarizes as follows:

1. Compute the quadric error for each vertex.
2. Construct a set of trees with collapsible edges.
3. Split the trees to meet τ.
4. Discard tree edges that incur in illegal mesh changes (topology and inversion).
5. Collapse trees to their root vertices.
6. Refine root vertex positions.

3.3.7 Hoppe et. al.

The Hoppe et. al. algorithm [11] is a CPU sequential algorithm that relies on the edge-collapse, edge-split and edge-swap operations, and uses the energy function metric to measure what operation will be made on an edge at a certain time.

The authors try to solve a problem that they call the mesh optimization problem, which can be stated as: given a collection of data points in R^3 and an initial triangular mesh M_0 near the data, find a mesh M of the same topological type as M_0 that fits the data and has fewer number of vertices.

The edge-collapse, edge-split and edge-swap are defined as the three elementary transformations that can be performed on the mesh. A legal move over a mesh K is defined as the application of one of these elementary transformations to an edge of K that leaves the topology of K unchanged. In Figure 3.18 we can see how these transformations acts over an edge.

To reach M from M_0, the algorithm tries to find a sequence of legal moves: randomly selects a legal move over a candidate set, to change the mesh K into K'. If $E(K') < E(K)$ (where E is the measure of the energy function over the mesh), then the legal move is accepted, otherwise, another legal move is tried. If a legal move is not accepted over a certain number of trials, then the algorithm terminates. The candidate set consist of all edges that may lead to a beneficial move, and initially contains all edges. To generate a legal move, randomly an edge from this set is selected. First an edge-collapse over this edge is evaluated, and accepted if the move is legal and reduces the energy. If it is not accepted, then an edge swap and edge split are evaluated. If one of the transformations is accepted, then the candidate set is updated, adding all neighbors of this edge to the set (because they could have been eliminated from the set in a previous iteration).

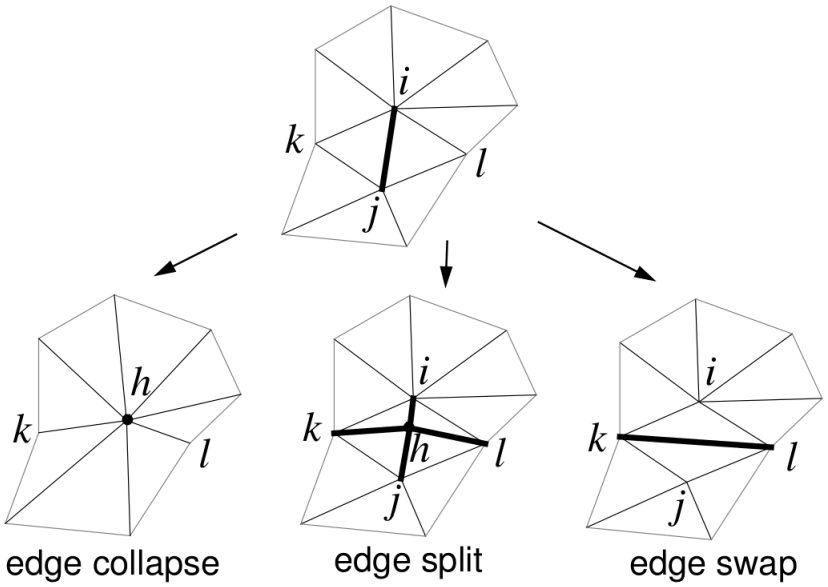


Figure 3.18: Example of the three legal moves. Image taken from *Mesh Optimization*[11], Figure 4.

This legal moves allow vertices to be added or removed from the mesh. When a vertex is added, the distance energy E_{dist} term from Equation 3.5 is likely to be reduced, while the E_{rep} term is increased, incurring in a penalty, so that vertices are not added indefinitely.

With this, the algorithm summarizes as follows:

1. Put all edges of the initial mesh in a set.
2. Select a random edge from the set, and perform a legal move if and only if, the Energy from the new mesh is smaller than the Energy of the previous one.
3. Repeat step 2 until no legal moves that minimizes the energy function can be performed.

Also, this algorithm can be used to solve the surface reconstruction problem, in addition to mesh simplification.

3.3.8 Hjelmervik-Leon

The Hjelmervik-Leon algorithm [10] is a hybrid CPU-GPU algorithm based on the half-edge collapse operation, and uses volume error metrics to measure the cost of a contraction. This algorithm considers mechanical based criteria, with the objective of removing shape details of 3D models, to use them in Finite Element Analysis, and the error zones of the metric can be adjusted individually to take additional mechanical criteria into account for the simplification operation. The hybrid behavior of the algorithm is that the mesh data structure is maintained in main memory (CPU), while the calculations are performed on the GPU. This enables the use of data structures for meshes on main memory that are hard to manage on GPU memory, while doing the intensive calculation tasks (like validation of the decimation criteria) on the GPU.

The edges that may be collapsed are selected from a sub-set of vertices. This sub-set is generated by sorting the vertices based on discrete curvature values, and then an independent set is constructed from the sorted vertices. In the next step, the candidate vertices are split into batches that can be allocated on GPU memory with all the information needed for each vertex (the geometrical information of a vertex and information about its adjacent vertices).

Because sharp edges are desired for mechanical applications, from a vertex, only at most the four edges with the smallest value (where this value is the dot product between the faces of the triangles that share that edge) are considered for the half-edge collapse step. To detect if an edge-collapse will incur in illegal configurations, a simple test is performed: if the angle between the new and the old triangle connected to each boundary edge is larger that a certain threshold (approx 80 degrees), then the new configuration may be illegal and it is excluded. This does not replace a more complete test that is performed later on the CPU (because it needs information outside the vicinity of the vertex), but helps to discard some edge-collapses and performs fewer calculations.

Two tables are maintained in GPU memory. From an adjacency list (in which, for each vertex, the info about its curvature value, its neighbor vertices, and its valence number (used to track the number of iterations in which this vertex has participated. This valence number is used to prioritize edge-collapse of elements that has not participated before), they construct a table of candidate vertices, in which candidate vertex with the same valence are in the same row of the table, and another table, in which for each row the indices to the neighbors vertices are stored, correlative to their position in the first table.

Finally, a test is performed on the resulting mesh, in which each error zone sphere (from the volume error metric), must intersect at least one triangle of the resulting mesh of an edge-collapse simulation performed on GPU. The vertices that pass all the tests are registered on memory along with its destination vertex (from the half-edge collapse).

Back in the CPU, the final consistency test is performed, and then the edge-collapse updates are performed on the mesh data structure. This process is repeated until no more vertex batches remain.

With this, the overall pipeline of the Hjelmervik-Leon algorithm is summarized in Figure 3.19. All steps on white rectangles on the diagram are executed on the CPU, While the gray ones are performed in the GPU.

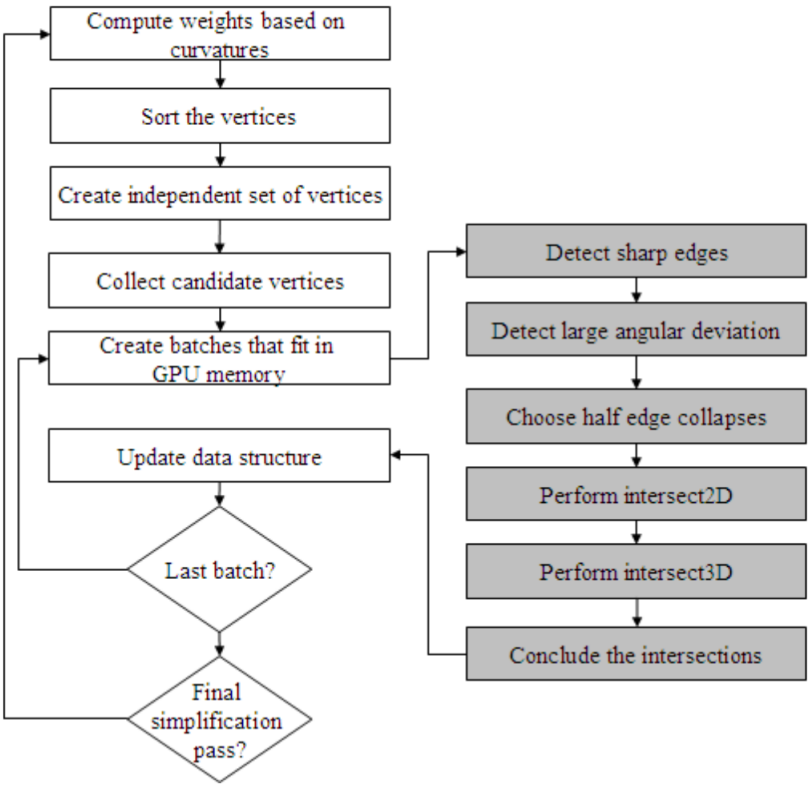


Figure 3.19: Structure of the Hjelmervik-Leon Algorithm. Image taken from *GPU-Accelerated Shape Simplification for Mechanical-Based Applications*[10], Figure 1.

3.3.9 Grund et al

The Grund et. al. algorithm [9] is a GPU parallel algorithm that relies on the edge-collapse operation, and uses Quadric error metric to measure the cost of an edge contraction. The principal objective of this algorithm is to generate a sequence of simplified meshes with smaller level of detail in each iteration for real time applications.

The first step is to compute the quadrics for each vertex, then the optimal collapse position and cost are computed for each edge. Then the local minimal collapse cost collapse associated for each vertex is computed and simplifications are performed, if they not exceed a certain error threshold (the threshold increases as a model with fewer details need to be generated). After updating the connectivity of faces and edges, the collapsed edges are removed from the data structure. If no further collapses are possible without exceeding the threshold value, the current mesh level can be copied to an output buffer. Then, the simplification loop starts again with an increased threshold, until all required LODs are generated or no more models can be generated.

The data structure the algorithm uses, holds the following data:

- For each vertex, store in an array its position, its quadric and the ID of the edge associated with this vertex that has the minimal collapse cost.
- For each face/triangle, store the indices to its vertices.
- For each edge, store its vertex indices, its collapse cost and the optimal position of the resultant vertex after collapsing this edge.

The selection of edge to be collapsed look as follows: first, for each vertex, the edge ID with the minimal collapse cost is calculated and stored. Then, an edge is a candidate to be collapsed only if its vertices have the same ID stored as associated edge with minimal cost. The collapse operation moves the vertices to its new pre-calculated optimal position.

With this, the overall pipeline of the Grund et. al algorithm is summarized in Figure 3.20. All steps on red rectangles on the diagram are executed on the CPU.

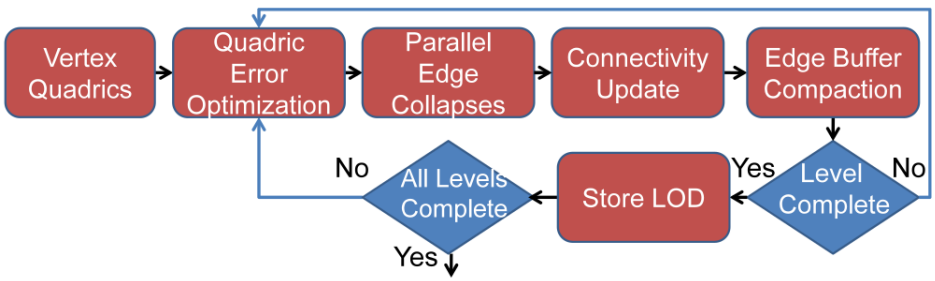


Figure 3.20: Structure of the Grund et. al. Algorithm. Image taken from *Instant Level-of-Detail*[9], Figure 4.

3.3.10 Cellier et al

The Cellier et. al. algorithm [2] is a parallel GPU algorithm based on edge-collapse, and uses the Quadric error metric to measure the cost of contracting an edge. The main objective of this algorithm is to generate Geographic Information Systems (GIS) representations that can be used in mobile devices and streaming context, offering user experiences similar to desktop applications on devices with limited resources.

The algorithm uses a data structure similar of the one used by the Grund algorithm, in which each vertex has a reference to the incident edge that can be candidate to a collapse, and are used to determine which edges can be collapsed in parallel. The parallel mechanism to prevent data races, is stated as:

- In parallel, for each vertex, store the ID of the associated edge that would generate the minimal error according to the metric.
- In parallel, search for all the pair of vertices that has the same edge ID stored (called reciprocal candidates), and change its state to *fusion requested*.
- In parallel, for each vertex with a *fusion requested* state, if no other vertex in the neighborhood has a state of *fusion requested* with a smaller error, then this candidate is accepted.

Because of the main objective, this article also talks about compression and decompression schemes to stream the mesh data over the web, but this is beyond the scope of what we want to investigate. The simplification algorithm steps are:

1. For each vertex, compute the quadric matrix.
2. Fix the maximum error threshold that generates an edge-collapse.
3. For each vertex, calculate and store which of its incident edges yields the minimal error.
4. For each vertex, if one of its neighbor vertices have the same stored edge, change its state to *fusion requested*.
5. For each vertex that has a state of *fusion requested* and if no other neighbour has a *fusion requested* state, then the vertex is updated with the new coordinates (resulting of the vertex generated by the edge-collapse), the quadric matrices are updated, and the information to reverse this collapse is stored. If several neighbors have a state of *fusion requested*, only the collapse with higher priority is performed between the conflicting candidates.
6. For each triangle in the neighborhood of the collapse: if the triangle is incident to only one merged vertex, its information is updated. If the triangle is incident to two merged vertices, then is deleted.
7. The edges and vertex array are updated to reflect the changes and information to undo this change (for the decompression phase).

This algorithm works based on two observations: the order in which the candidates are merged does not influence the error metric, and the influence of an edge-collapse does not exceed the zone delimited by its immediate neighbors.

3.3.11 Vad'ura

The Vad'ura algorithm [25], is a parallel GPU algorithm based on edge-collapse, that uses volume error metrics to measure the cost of contracting an edge.

This algorithm works by splitting the input mesh into clusters that can be processed independently of each other. This has a similar fashion of a vertex clustering algorithm, but the difference is that instead of clustering all vertices into a single point, the edge-collapse operation is performed on each cluster exhaustively. Also, when splitting the mesh, border triangles on each cluster are marked, so that their edges cannot be removed (because its removal may involve data races among the threads who handles contiguous clusters). Each cluster should fit in the shared memory of a GPU multiprocessor.

With this, the algorithm summarizes as follows:

1. First, each edge determines if it will collapse into one of its endpoints or its center, depending on which one yields the lowest value of the error metric.
2. In each cluster, a thread searches the edge with the lowest cost. Because the clusters tends to be small in size, this should not be so inefficient as it seems.
3. Finally, the selected edge in each segment is collapsed, and the surrounding geometry is altered to reflect this change.
4. After a successful simplification of all clusters, all simplified parts are merged back into a single mesh.

This process can be repeated until a specific target simplification is reached (such as the number of triangles that we want), but in every new pass, a new segmentation of the mesh has to be calculated.

In Figure 3.21, we can see the result of this algorithm. The borders of the segments are clearly visible, and show that this algorithm can lead to simplified meshes that are not homogeneous. Finally, the document does not explain how to split the mesh so that the elements are distributed uniformly in a grid.

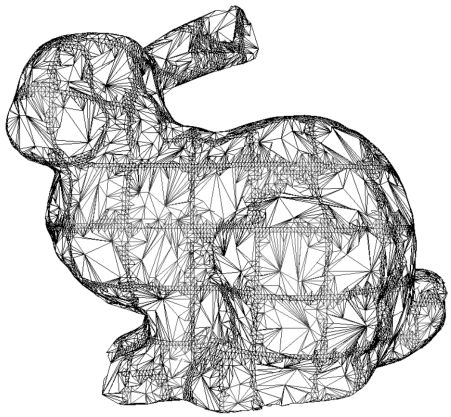


Figure 3.21: Example of a result of the Vad'ura algorithm. Image taken from *Parallel mesh decimation with GPU* [25], Figure 1.

Chapter 4

Algorithm Proposal

In this chapter, we propose our own version of a mesh simplification algorithm, taking some steps or advice from the algorithms reviewed in the previous chapter. First, we include the data structures used, and then describe how to select the candidate edges and the mechanism used to prevent data races from parallel operations and concurrent updates. Next, we discuss how to preserve the integrity of the mesh during the operations. Finally, we describe in detail the proposed algorithm and the phases to carry out the simplification technique.

4.1 Simplification Scheme

We propose a novel algorithm that re-use data structures, error metrics and strategies described in the literature review, for a simplification algorithm based on edge-collapse, adding new steps and data structures that use redundant information to solve this process in a most robust and efficient way.

The main reason we selected edge-collapse over the others techniques reviewed in Section 3.1 is that the majority of the algorithms that we saw in Section 3.3 use this strategy as a main simplification technique (especially the parallel ones), and led us to believe that it will be easier to implement. Also, the Cleap data structure handles explicit and redundant information about the mesh edges, and may be useful for consistent deletions and updates on it.

Our algorithm proposal consists of the steps described in Figure 4.1. In the first two steps we select which edges will be candidates to be evaluated for edge-collapse, and is explained in more detail in Section 4.3. In the next steps, candidates are assigned to a thread and then are tested if they are suitable for edge-collapse, and in that case are simplified. This steps are explained in more detail in Section 4.5 and Section 4.6. The final step is the generation of the output mesh file, and it is explained in Section 4.7.

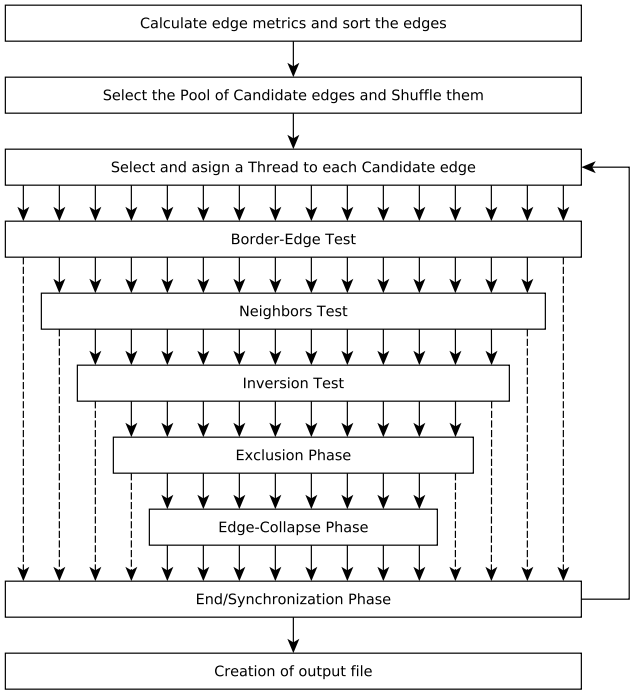


Figure 4.1: Overall Steps of the Simplification algorithm.

4.2 Data structures

The base data structure we use is the Cleap data structure described in Section 2.3 and shown in Figure 4.2. The principal reason is that we want to integrate the algorithm into the Cleap library, and so we would not want to implement a whole new side-structure just to handle the edge-collapse simplification case. This is not bad as it seems, because the Cleap data structure offers explicit references between vertices, edges and faces. It also stores redundant information about the connectivity between elements of the mesh, which helps check for inconsistencies in parallel much easier, as it has been proven in previous works, such as parallel Delaunay triangulation[18] and parallel mesh smoothing[16].

As for complementary data structures, to handle the simplification more efficiently, we include an adjacency table for vertices, similar to the one used by the Papageorgiou-Platis algorithm we saw in Section 3.3.3, Figure 3.12, to know which vertices are in the immediate vicinity of a given vertex, which helps calculate the cost of a given edge-collapse much faster (because we would not want to seek over all the list of vertices to know which vertices are neighbors of other vertices).

This table can be seen graphically in Figure 4.3, where each vertex has an associated number of triangles that are incident to it and a field that indicates if there are more incident triangles in the memory space of another vertex (the ones that have been merged with it because of the edge-collapse operation). The spaces marked as a dot have no data stored. In the case of vertex $v2$, which has a *continues* field $n-1$, indicates that in the memory space associated to the vertex $n-1$, there are more incident triangles associated to the vertex $v2$.

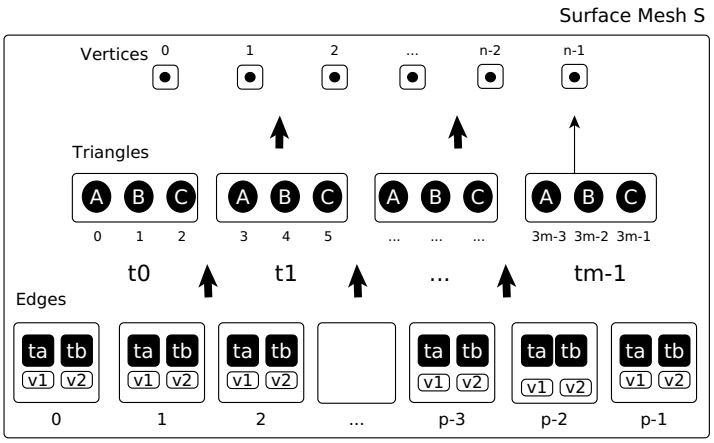


Figure 4.2: Cleap Data Structure Diagram. Diagram by Cristobal Navarro

Appendix B displays a graphical example of the updates performed over the Adjacency table in the main part of the simplification algorithm described in Section 4.6, when a merge of the adjacency information of the vertices is performed.

For memory-efficiency reasons, this table is implemented as a big one-dimensional array, where the space associated to each vertex is handled via modular operations. For example, the first 10 elements are associated to the first vertex, the next 10 elements are associated to the second vertex, etc. In each group of 10 elements, the first is the triangle count, the second is the continue field, and the remaining ones are the incident triangles IDs.

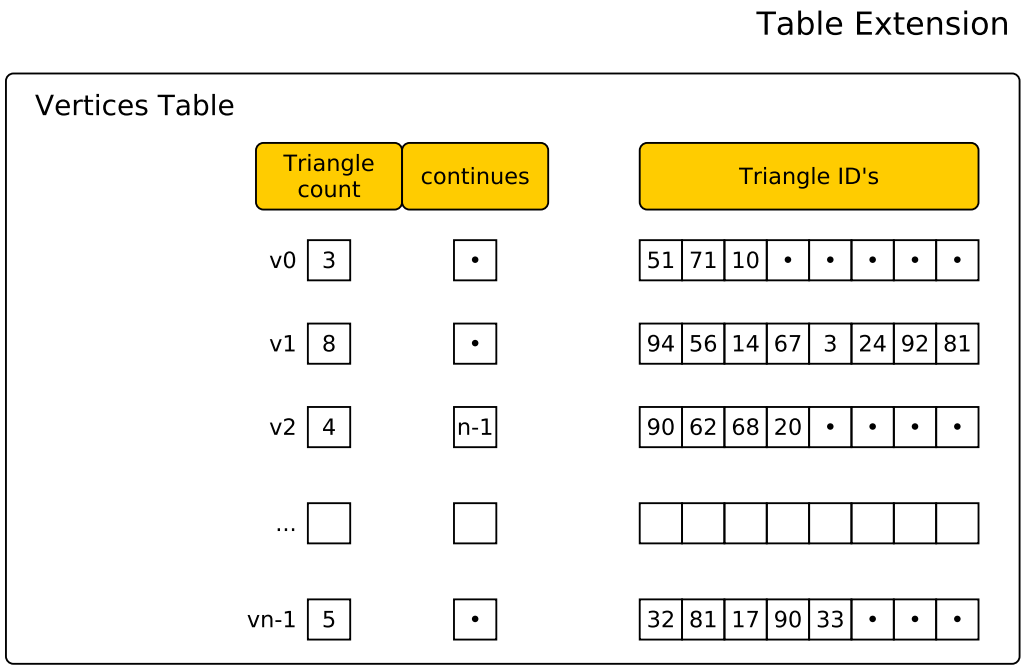


Figure 4.3: Vertices Table extension of the main data structure.

We need also an array to associate each edge with its simplification cost, according to the metric, for a fast calculation of the edge-collapse cost. This is equivalent to adding a new field to the edges in the main data structure, but we have chosen not to do this, to not give an extra overhead to that data structure, because that field would not be used by all the algorithms in the library, and may lead to memory inefficiency in meshes with a high number of vertices (approx. over 10 millions). We will call this array, the *cost array*.

Finally, we need a structure that will handle locks for the mutual exclusion mechanism. Similar to the one used by the Shontz-Nistor Algorithm we saw in Section 3.3.1, we need an array that helps us register which triangle has been taken by a certain GPU thread. We will use a simple array with length equal to the number of triangles, in which the indices denote the ID of the triangle, and the data contained in each position of the array will be the thread ID if it has been taken by that GPU thread, or -1 if the triangle is free (threads ID do not have negative values). We will call this array, the *flags array*.

Both arrays can be seen graphically in Figure 4.4, where in the edges array the contraction cost associated to each edge is stored, and in the triangle array, the ID of the GPU thread that has control over each triangle is stored.

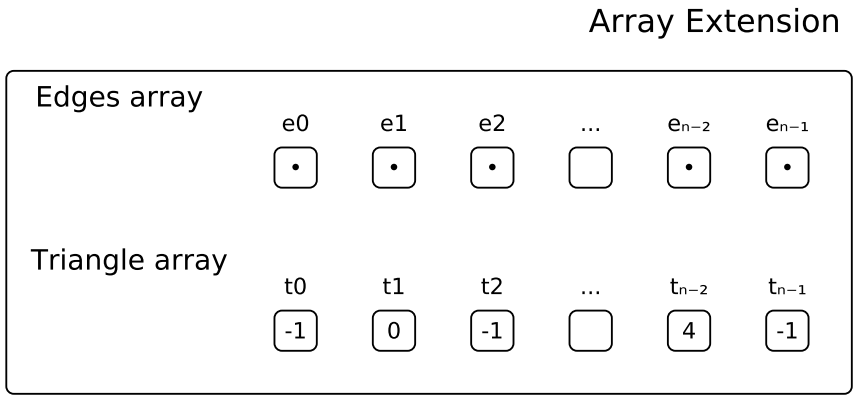


Figure 4.4: Array extensions of the main data structure.

4.3 Candidate Selection

Next, we need to determine how to select which edges will be simplified above all others, i.e., the metric to be used. For this purpose, we include the Quadric error metric from Garland, reviewed in Section 3.2.2, and the Edge length metric from Melax, which we reviewed in Section 3.2.1. The first one has been selected because it is used in most of the algorithms reviewed in 3.3, under different kinds of contexts (types of input meshes and focus of the algorithms), and this metric is a good cost-time effective starting point. Also this metric can be calculated in parallel, as shown in Sections 3.3.3 and 3.3.6.

Algorithms 1 and 2 shows the main steps to compute this metric in parallel. With the first Algorithm, for each triangle, we deduce the coefficients of the plane equation that holds each triangle. Next, we calculate the matrix K associated to this plane (Equation 3.3 on Section 3.2.2). Finally, we compute the sum of planes for each vertex. Each thread handling a triangle sum its respective previous calculated plane to a common array (each position of the array stores the sum of all planes associated to the vertex associated to that position (Equation 3.2 on Section 3.2.2) To ensure that there are no concurrency problems, we use the atomic CUDA primitive *atomicAdd* which adds a number to the designated memory space. if two or more threads across the GPU try to do the sum operation to the same memory space, then this function serializes the access to that space, so no updates are performed concurrently.

Algorithm 1: Edge Collapse Algorithm: Quadric metric for vertices Kernel

```
   Data: vertices[ ], triangles[ ], vertexQuadric[ ], triangleCount
   Result: Void
 1 threadID = getID();
 2 if threadID < triangleCount then
 3 |   v1 = triangles[3*threadID];
 4 |   v2 = triangles[3*threadID+1];
 5 |   v3 = triangles[3*threadID+2];
 6 |   plane = createPlane(vertices[v1], vertices[v2], vertices[v3]);
 7 |   K = plane * transpose(plane);
 8 |   atomicAdd(vertexQuadric[v1], K);
 9 |   atomicAdd(vertexQuadric[v2], K);
10 |   atomicAdd(vertexQuadric[v3], K);
11 end
12 return;
```

Algorithm 2: Edge Collapse Algorithm: Quadric metric for edges Kernel

```
   Data: vertices[ ], edges[ ], triangles[ ], vertexQuadric[ ], cost[ ], edgeCount
   Result: Void
 1 threadID = getID();
 2 if threadID < triangleCount then
 3 |   v1 = edges[threadID].n.x;
 4 |   v2 = edges[threadID].n.y;
 5 |   midV.x = (vertices[v1].x + vertices[v2].x)/2;
 6 |   midV.y = (vertices[v1].y + vertices[v2].y)/2;
 7 |   midV.z = (vertices[v1].z + vertices[v2].z)/2;
 8 |   Q = vertexQuadric[v1] + vertexQuadric[v2];
 9 |   cost[threadID] = transpose(midV) * Q * midV;
10 end
11 return;
```

Finally, we calculate the metric cost per edge with Algorithm 2. For each edge, we calculate the midpoint of it, then we compute the Q quadric matrix associated to this edge as the sum of the matrices associated to the edge endpoints. We calculate the metric cost as the operation $v_{mid}^T Q v_{mid}$ (Equation 3.4 on Section 3.2.2).

On the other hand, the second metric has been selected because it is easy to calculate, and serves as a comparison point for the performance of the other metric, and also can be calculated in parallel, because it only involves calculations with data stored on each edge. Algorithm 3 shows the main steps to compute this metric in parallel. The function *magnitude* helps to calculate the magnitude of the vector created from points v_1 and v_2.

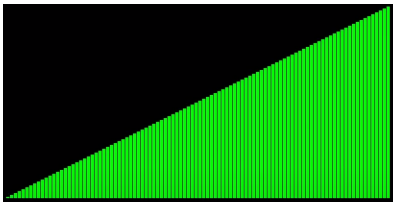
Algorithm 3: Edge Collapse Algorithm: Edge-length metric Kernel

```
   Data: edges[ ], cost[ ], edgeCount, threshold
   Result: Void
1  threadID = getID();
2  if threadID < edgeCount then
3  |   v1 = edges[threadID].n.x;
4  |   v2 = edges[threadID].n.y;
5  |   cost[threadID] = magnitude(v1,v2);
6  end
7  return;
```

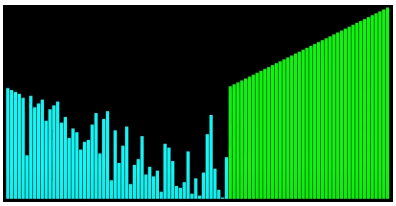
When the cost is calculated and stored for all the edges in the *cost array*, then the array is sorted by cost, from the least to most expensive, while keeping track of which cost is associated with each edge, with the help of an auxiliary array. At this stage, the review in Section 3.3 shows that the main selection criteria for the candidates is to select the candidates with the lowest cost, and collapse them until a certain quota of deleted elements is reached (as a fixed number of elements or a percentage of total elements). We have chosen to have a pool of candidate edges to evaluate. This pool is a subset of the first k edges in the sorted array, where k is a fraction of the total number of edges (between 5% and 80%).

In early experimentation, we saw that edges with the same cost tends to be relatively near, and choosing a batch of contiguous candidate edges on the *cost array*, results in some problems for the parallel exclusion mechanism. The principal problem was that a low number of threads could lock-down its assigned edge vicinity (less than 10%), leading to a high number of idle threads, wasting GPU parallel resources. Because of this, we have decided to shuffle the pool of candidate edges, and then take the first N edges on the pool, where N is the number of parallel threads, then execute the simplification algorithm in this set, and then take the next group of N edges, execute, and so on.

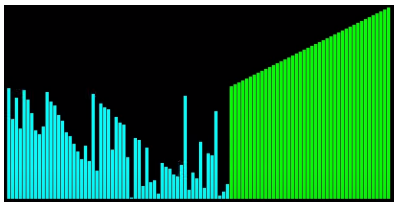
The shuffle is made using the Fisher-Yates algorithm[3], which generates a random permutation of a sequence of elements (in this case, the sequence is the error metric for each edge), where every possible permutation is equally likely to be generated. In Figure 4.5 we can see a graphical example of the input of the algorithm (Figure 4.5a), and two possible outputs (Figures 4.5b and 4.5c). The green section represents a group of sorted elements, while the cyan section is the shuffled one, and represents our pool of candidates.



(a) A sorted array of elements.



(b) A possible output of the algorithm.



(c) Another possible output of the algorithm.

Figure 4.5: Fisher-Yates Algorithm example. Images taken from `https://github.com/vbohush/SortingAlgorithmAnimations`

Algorithm 4 shows an implementation of this shuffle. A more detailed description of how this algorithm can be implemented, and detailed explications of how this algorithms works, can be found in the book *Seminumerical algorithms, The Art of Computer Programming*, written by Donald Knuth [12].

Algorithm 4: Edge Collapse Algorithm: Fisher-Yates shuffle algorithm

```
   Data: cost[ ], maxCandidates
   Result: Void
 1 srand(time(NULL));
 2 i = maxCandidates;
 3 while i > 0 do
 4 |   limit = RAND_MAX - RAND_MAX%(i+1);
 5 |   r = limit;
 6 |   while r >= limit do
 7 |   |   r = rand();
 8 |   end
 9 |   j = r%(i+1);
10 |   tmp = cost[j];
11 |   cost[j] = cost[i];
12 |   cost[i] = tmp;
13 |   i–;
14 end
15 return;
```

Doing this shuffle and then taking a group of contiguous elements is similar to randomly choosing N candidates in the sorted array (random pick), with the advantage that we do not need to handle cases where two or more threads randomly choose the same edge, or choose an edge that has already been processed in a previous pass of the algorithm. Another advantage is that there is no need to rearrange the array to separate the processed edges from the candidate edges, because the processed edges always will be on the left side of the array, in contrast to random pick, where the processed edges and the candidate edges are mixed in the array, and may lead to randomly choosing a processed or discarder edge more often as more passes of the algorithm are executed.

4.4 Mutual Exclusion Mechanism

As for the Mutual Exclusion mechanism to handle parallel operations, our review on Section 3.3 shows that there are 3 popular approaches:

- Launch the algorithm over the critical section, and use locks or flags to ensure that a thread has control over a critical section and no concurrent updates are performed on it while the thread holds the locks on that section.
- Execute a pre-routine that determines which thread is assigned to each critical section and in which order, to ensure that no concurrent updates are performed on the main algorithm.
- Launch the algorithm over the critical section, without a lock or serializing mechanism, and allowing concurrent updates. Then execute a routine to repair inconsistencies in the data structure.

For our algorithm, we have chosen a mechanism similar to the Shontz-Nistor algorithm shown in Section 3.3.1., which falls into the category of the second approach. An edge can be collapsed if and only if another edge-collapse is not being performed in its neighborhood in the same iteration. As for the neighborhood definition, we also take the Shontz-Nistor vicinity seen in Figure 3.6, which is that a given edge can be allowed to be collapsed, if and only if the thread handling it has control or a lock-down over all the incident triangles to the vertices that belongs to that edge and all the incident triangles to the opposite vertices to that edge.

To support this mechanism, only an array of flags for the triangles is needed (and that is why an additional data structure for triangles was added on Section 4.2). This mechanism works by concurrently editing the ownership of a triangle by the active threads (in which each thread represents a certain edge) on the array.

In the beginning, each element on the flags array represents the i-th triangle, where i is the index on the array. All elements start with the value -1, which indicates that the triangle has not been taken by a thread. Then, when the N pre-candidate edges have been selected, each thread handling a pre-candidate tries to take control over its neighborhood, writing its thread ID on each of the triangles that belongs to it.

To check and mark each triangle concurrently without raising race conditions, we use the CUDA atomic primitive *atomicCAS(Compare and Swap)*, which in a single operation does the following:

- Reads a memory space and check if a certain expected value is stored.
- If the stored value is the expected one, then it writes a new value on that memory space, and returns the old value.
- If the stored value is not the expected one, then it returns the value stored on that space.

With *atomicCAS*, the thread can ask a triangle in a single operation if it has been taken by a thread. If it is free or taken by the asking thread, it assigns its ownership to the asking thread. With this, the threads try to write its ID value for all the triangles on the respective neighborhood. If at a certain point the thread cannot take control over all its neighboring triangles (because another thread had taken control over a common triangle previously), then it releases all captured triangles, so another thread may take control of them if needed. Also this edge is discarded as a candidate for the current iteration of the algorithm.

If two or more threads try to read/modify the same triangle memory space at the same time, then *atomicCAS* serializes the access to this memory space, so it can guarantee that only one thread will take control over a triangle if it is free at a given time.

Another reason to choose this kind of mutual exclusion is that it only needs to be checked before the edge-collapse steps are performed, so there is no need to do a constant check for the neighborhood of an edge while doing the simplification procedure on the main part of the algorithm.

Algorithm 5 shows the main steps of the lock-down mutual exclusion mechanism. The function *getVertices* on line 4 returns the vertices-ID of the edge endpoints (v_1 and v_2), and the ID of the vertices opposite to the edge (v_{op1} and v_{op2}). In line 11 and 12, if the thread cannot take ownership of the current triangle, then deletes itself from the pool of candidate edges, and release all of the triangles taken (if any), writing -1 with *atomicCAS* in its respective space on the *triangleFlags* array.

Appendix A displays a graphical example of the lock-down mechanism work.

Algorithm 5: Edge Collapse Algorithm: Parallel Lock-down Kernel

```
   Data: edges[ ], triangles[ ], adjTable[ ], triangleFlags[ ], candidateEdges[ ], edgeCount
   Result: Void
 1 threadID = getID();
 2 if threadID < edgeCount and candidateEdges[threadID] != -1 then
 3 |   edgeID = candidateEdges[threadID];
 4 |   v1,v2,vop1,vop2 = getVertices(edges[edgeID]);
 5 |   for v in [v1,v2,vop1,vop2] do
 6 |   |   for triangle in adjTable[v] do
 7 |   |   |   lock = atomicCAS(triangleFlags[triangle], -1, edgeID);
 8 |   |   |   if lock == -1 or lock == edgeID then
 9 |   |   |   |   continue;
10 |   |   |   else
11 |   |   |   |   candidateEdges[threadID] = -1;
12 |   |   |   |   releaseTriangles(edgeID,adjTable);
13 |   |   |   |   return;
14 |   |   |   end
15 |   |   end
16 |   end
17 |   return;
18 end
```

4.5 Integrity Test

Before we do the edge-collapse procedure over the candidate edges, we have to test if its simplification does not greatly affect the quality of the resulting mesh, such as changes that affect the overall geometry or topology of the mesh. A review by Martin Kraus and Thomas Ertl [13], establishes that to prevent geometry changes, it is necessary to perform a test over the mesh and its neighborhood, and only do the edge-collapse with those edges that pass the test.

First, it is desirable to do the edge-collapse on convex meshes, while preserving the original convex-hull. If the mesh is non-convex, then it is desirable to do a pre-process named *convexification*, which fills the voids on the mesh with imaginary faces and edges, and lets the algorithms and test treat it as a convex mesh, handling fewer special cases present on non-convex meshes. An example of *convexification* can be seen in Figure 4.6, where the white space is filled with imaginary faces and edges, until all the void spaces inside the convex hull are filled.

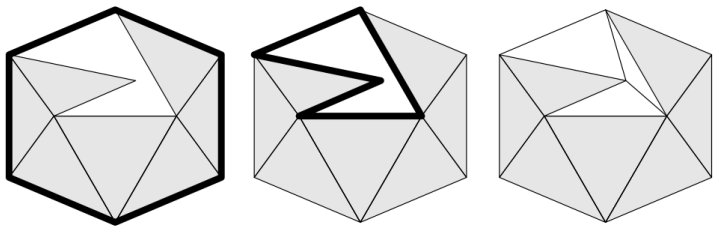


Figure 4.6: Description of the Convexification process. Image taken from *Simplification of Nonconvex Tetrahedral Meshes*[13], Figure 4.

Then, to preserve the geometry, it is necessary to do a test on the neighborhood of the edge to be collapsed, and observe that the faces on that neighborhood does not change its orientation before and after the collapse (phenomenon called *inversion*). The orientation is defined as the normal vector of the face, following the *Right-hand rule* in the order of how the vertices are stored in the data structure.

An example of this phenomena can be seen in Figure 4.7, where the edge-collapse causes an inversion of the dark-gray face. A simulation is performed previous to the simplification, to prevent the situation when an edge-collapse is performed and then realize it was not viable. This simulation can be done doing the orientation calculations of the faces, but considering the position of the vertices in the places that they will be after the edge-collapse is performed.

Finally, to preserve the topology, it is necessary to perform a test on the edge neighborhood, and detect if the edge-collapse will connect two previously disconnected faces, or disconnect two or more previously connected faces.

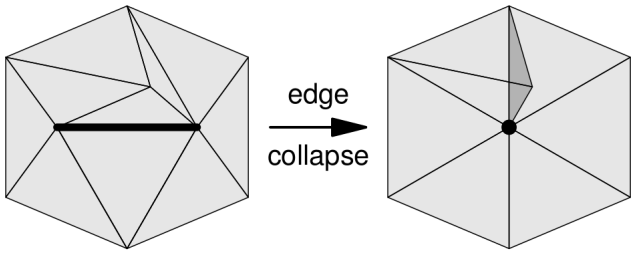
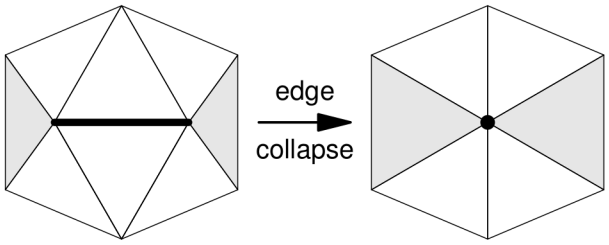


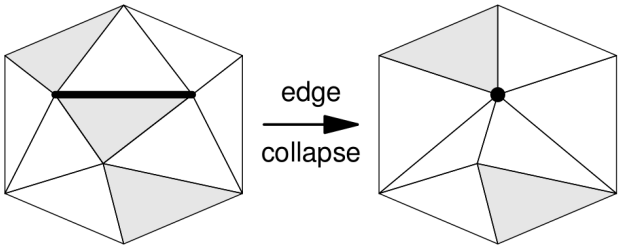
Figure 4.7: An edge-collapse that produces a face inversion. Image taken from *Simplification of Nonconvex Tetrahedral Meshes*[13], Figure 2.

An example of the two cases can be seen in Figure 4.8, where in 4.8a a connection is made between the two previously disconnected gray faces, and in 4.8b a disconnection is made between two previously gray connected faces (via a face that was deleted because of the edge-collapse).

Two or more faces are connected if they share at least one vertex, while two or more faces are disconnected if they are not connected (this is equivalent to be connected by at least one imaginary edge). Then, the idea of this test is to classify the faces of the mesh as real or imaginary (to give sense to the connected-disconnected faces definition), and then detect if the edge-collapse will change the pre-existent connectivity between the faces on the neighborhood.



(a) An edge collapse that produces a connection.



(b) An edge collapse that produces a disconnection.

Figure 4.8: Edge-collapse examples that change the topology of the mesh. Images taken from *Simplification of Nonconvex Tetrahedral Meshes*[13], Figures 7 and 8.

Inspired in the Kraus-Ertl review and to ensure good edge-collapses, we perform validation tests over all candidate edges, only keeping the candidate edges that pass all tests. Those tests are:

1. Inversion Test
2. Border-Edge Test
3. Neighbors Test

In the next subsections we explain those tests in more detail.

4.5.1 Inversion Test

To prevent geometry changes, and following the Kraus-Ertl suggestions, and because almost all algorithms reviewed in Section 3.3 do some kind of procedure to prevent or repair face inversions, we perform an *Inversion Test* over the candidate edges. This test consists of registering the face orientation of the elements on the neighborhood of the candidate edges (with its normal vector relative to the *right-hand rule*), and then comparing the orientation after a simulation of the edge-collapse is performed (as described in Section 4.5).

This is equivalent to calculating the signed area of the face, where the sign is provided by the normal vector relative to the right hand rule, and then compare if the sign of the face changes after the edge-collapse. A description of this test is:

1. Calculate the coordinates of the resultant vertex v_n produced by the edge collapse (v_1, v_2)
2. For each face in the neighborhood of the candidate edge, calculate in parallel, the area of the face before the edge-collapse operation takes place.
3. For the calculation of the area post edge-collapse, we calculate just as we did in step 2, but before the calculation is performed, we detect if the face has one of the deleted vertices (v_1 or v_2) as one of its vertices. In this case, we perform the calculation of the new area, using the value of v_n instead of the originals v_1 or v_2 accordingly. If the face has two of the deleted edges, this means that is one of the deleted triangles of the edge-collapse operation, so it is excluded from this comparison.
4. If after the comparison at least one face has different area sign, then there was an inversion, so this candidate edge cannot be collapsed and it is discarded.

Algorithm 6 shows the main steps of the inversion test. The function *calculateMidPoint* on line 5 calculates and creates a new vertex that will be the midpoint of the edge with endpoints v_1 and v_2. The function *inversionHelper* is explained with more detail on Algorithm 7. This function calculates the normal vector of each triangle that belongs to the vicinity of the candidate edge, then calculates the normal vector as if the edge has been collapsed, and finally, compares both vectors. If the vectors are different it means that this collapse will produce an inversion, and cannot be performed. The check on line 2 is to exclude the triangles from the test that will be deleted because of the possible edge-collapse.

Algorithm 6: Edge Collapse Algorithm: Inversion Test

```
Data: vertices[ ], edges[ ], triangles[ ], adjTable[ ], candidateEdges[ ], edgeCount
Result: Void
threadID = getID();
if threadID < edgeCount and candidateEdges[threadID] != -1 then
    edgeID = candidateEdges[threadID];
    v1,v2,vop1,vop2 = getVertices(edges[edgeID]);
    vmid = calculateMidPoint(v1,v2);
    triD1, triD2 = getAdjacentTriangles(edgeID);
    for v in [v1,v2,vop1,vop2] do
        if inversionHelper(vertices, edges, triangles, vmid, v1, v2, v, triD1, triD2, adjTable)
          == -2 then
            candidateEdges[threadID] = -1;
            return;
        end
    end
end
```

Algorithm 7: Edge Collapse Algorithm: Inversion Test Helper

```
Data: vertices[ ], edges[ ], triangles[ ], vmid, v1, v2, v, triD1, triD2, adjTable[ ],
      candidateEdges[ ]
Result: int
vcoord1 = vertices[v1];
vcoord2 = vertices[v2];
for triangle in adjTable[v] do
    if triangle == triD1 or triangle == triD2 then
        continue;
    end
    pre-normal = calculateNormal(triangle);
    a,b,c = verticesOf(triangle);
    if a == vcoord1 or a == vcoord2 then
        a = vmid;
    end
    if b == vcoord1 or b == vcoord2 then
        b = vmid;
    end
    if c == vcoord1 or c == vcoord2 then
        c = vmid;
    end
    newTriangle = makeTriangle(a,b,c);
    post-normal = calculateNormal(newTriangle);
    if pre-normal != post-normal then
        return -2;
    end
end
return 1;
```

4.5.2 Border-Edge Test

To prevent topology changes, instead of doing a full topology test as described by Kraus-Ertl, we do a simplified custom test, because if the mesh is originally convex (like the terrain meshes we are working on), then all elements on the mesh are connected and there is no need to introduce imaginary elements to fill gaps. Then, the only way to risk the mesh convexity, is to delete vertices that belong to the convex-hull of the mesh, or equivalently, collapse edges that belong to the border of the mesh or have at least one of its vertices in the border/convex-hull. We want to avoid the collapse of these edges, so for this objective, we detect and mark them as non-collapsable.

To detect these edges, we use the relationship information between edges and vertices provided by the Cleap data structure. First, we look in parallel for the edges that have only one triangle associated (each edge has references to its adjacent triangles in its local memory), because if there is only one triangle, this edge lies in the border of the mesh (and hence, its vertices). Then, we mark the vertices of this edges with a flag in a temporary array (similar to the triangle *flags array*, but for the vertices). This procedure can be seen in detail in Algorithm 8.

Finally, for each candidate edge, we look in parallel for the vertices of this edge, and the edge will pass the test if and only if both of its vertices are not marked in the temporary array. The implementation of this step can be seen in Algorithm 9.

The steps of this test summarize as follows:

1. For all edges in parallel, look for edges that have only one adjacent triangle. Mark or store in a temporary array the ID of the vertices associated to these edges.
2. For each candidate edge, in parallel, look if its vertices are marked in the temporary array, and discard the candidate if at least one of its vertices is flagged.

In Figure 4.9, we see a mesh with an example of the edges that cannot be collapsed and will be excluded (in red), according to this test.

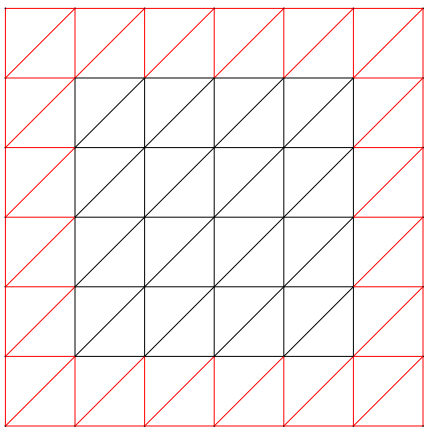


Figure 4.9: Example of edges flagged by the border-edge test.

Algorithm 8: Edge Collapse Algorithm: Border-Edge Test - Vextex-Flag step

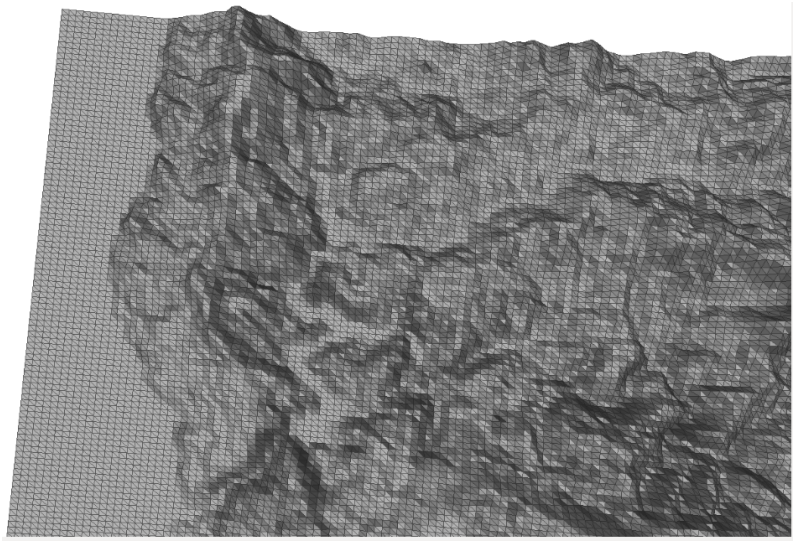
```
Data: vertices[ ], edges[ ], vertexFlags[ ], edgeCount
Result: Void
threadID = getID();
if threadID < edgeCount then
    edgeID = threadID;
    TriA, TriB = getAdjacentTriangles(edges[edgeID]);
    if TriA == -1 or TriB == -1 then
        v1,v2 = getVertices(edges[edgeID]);
        vertexFlags[v1] = 1;
        vertexFlags[v2] = 1;
    end
    return;
end
```

Algorithm 9: Edge Collapse Algorithm: Border-Edge Test

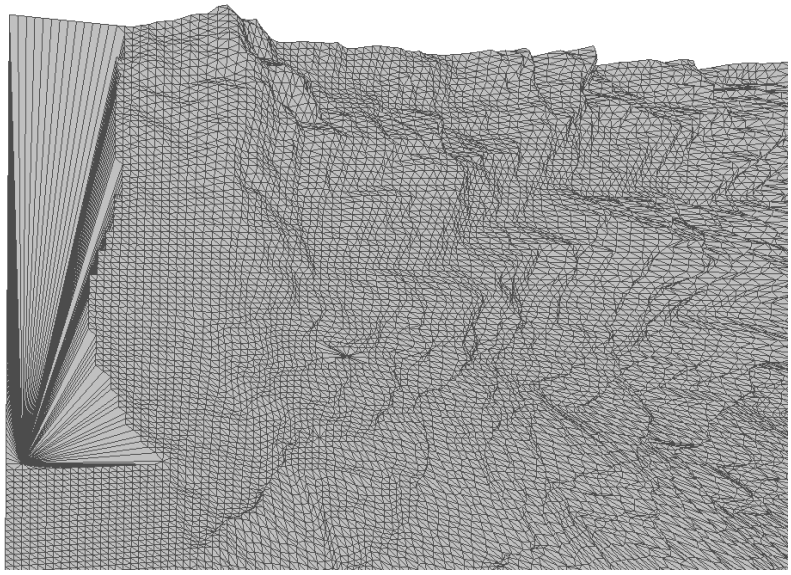
```
Data: vertices[ ], edges[ ], candidateEdges[ ], vertexFlags[ ], edgeCount
Result: Void
threadID = getID();
if threadID < edgeCount and candidateEdges[threadID] != -1 then
    edgeID = candidateEdges[threadID];
    v1,v2 = getVertices(edges[edgeID]);
    if vertexFlags[v1] == 1 or vertexFlags[v2] == 1 then
        candidateEdges[threadID] = -1;
    end
    return;
end
```

4.5.3 Neighbors Test

From the review in Section 3.3, we observed that many of the solutions try to simplify the mesh evenly, in order to avoid zones of the mesh that are more simplified than others. Without taking measures against this, some situations like the one in Figure 4.10 may arise, where a sequence of edge-collapses contracts over a single vertex consecutively, generating a large number of sliver triangles and vertices with too many connected edges, which can affect the quality of the result and may give rise to some rendering problems. This kind of behavior is not addressed by the previous test or the Kraus-Ertl review, because they consider it a legal theoretical configuration. Because of this, it is necessary to do a custom test to prevent this situation from happening.



(a) Terrain mesh.



(b) Terrain mesh after 100 consecutive edge-collapses.

Figure 4.10: A Sequence of edge-collapses that consecutively contract to the same vertex.

We address this problem by restricting the maximum number of triangles that the neighborhood of a vertex generated by an edge collapse may have. In particular, when an edge collapse is performed, the number of triangles incident to its resulting vertex is the sum of the triangles incident to the edge endpoints, minus four, because the edge-collapse removes the two adjacent triangles of the edge, and each one of these removed triangles is counted twice (one per vicinity of the edge endpoints). If we define $I(v)$ as the number of triangles incident to the new vertex v (the vicinity of the vertex), we have the formula shown in 4.1, where v_1 and v_2 are the endpoints of the collapsed edge.

$$I(v_n) = I(v_1) + I(v_2) - 4 \tag{4.1}$$

This Equation works under the assumption that the edge-collapse is performed on a non-border edge - a condition we have for the Border-Edge test defined in Section 4.5.2. nevertheless, if we want to handle the border-edges case as well, we have to subtract two instead of four, because the border edges only have one incident triangle.

The condition that we want to test is if the size of the vicinity of the vertex generated by an edge-collapse does or does not exceed some constant threshold value c, the maximum size of a vicinity that a vertex may have at any time. This is expressed mathematically as testing the condition $I(v_n) < c$.

Thanks to the Vertices table extension described in Section 4.2, we can know the size of the vicinity that an edge-collapse will generate, counting the number of incident triangles that each vertex has. Then, using Equation 4.1 with the vertices of the edge as input, we will know the number of incident triangles that the resultant vertex will have.

The test implementation in Algorithm 10 follows:

1. For each candidate edge, in parallel, obtain its vertex ID.
2. Obtain the size of the vicinity of each of its vertices, using the Vertices table extension.
3. Use Equation 4.1 and obtain the size of the resulting vicinity.
4. If the size is smaller or equal than c, then the candidate edge passes this test, otherwise, the candidate is discarded.

Because the initial mesh comes from a very special kind of triangulation, initially all non-border vertices have $I(v) = 6$. Preliminary experiments shown that a value of $c = 14$ allows a fair and homogenized simplification across all the mesh, because it heuristically allows at most, 2 consecutive collapses or 3 chain collapses on a certain fixed neighborhood.

Algorithm 10: Edge Collapse Algorithm: Neighbors Test

```
   Data: vertices[ ], edges[ ], candidateEdges[ ], adjTable[ ], edgeCount, threshold
   Result: Void
 1 threadID = getID();
 2 if threadID < edgeCount and candidateEdges[threadID] != -1 then
 3 |   edgeID = candidateEdges[threadID];
 4 |   v1,v2 = getVertices(edges[edgeID]);
 5 |   h = 0;
 6 |   for v in [v1,v2] do
 7 |   |   for triangle in adjTable[v] do
 8 |   |   |   h++;
 9 |   |   end
10 |   end
11 |   if h > threshold then
12 |   |   candidateEdges[threadID] = -1;
13 |   end
14 |   return;
15 end
```

4.6 Simplification Algorithm

After we have the candidate edges that passed all the previous tests, and the mutual exclusion mechanism, we have a group of edges that can be collapsed in parallel, without interfering with one another, requesting writing access to common memory addresses or generating inconsistencies on the resultant mesh.

In this section, we describe the stages of the parallel edge-collapse algorithm, which are:

1. Elimination Phase.
2. Triangle Repair Phase.
3. Edge Repair Phase.
4. Connectivity Repair Phase.

Because each edge-collapse at this stage is independent of each other, we give examples of how the edge-collapse is performed and alter the data structures for the case of a single edge-collapse. Nevertheless, the description of the algorithms for each phase will be for the general case.

This algorithm relies on a clever use of the memory space of deleted elements between phases of the algorithm, to send information to the other phases of what elements they should delete or modify. This approach works, because a deleted element should not be referenced by any other element of the mesh after all stages are performed, so it should not cause inconsistencies on the represented mesh. In Figure 4.11 we see the overall steps of the algorithm. Each arrow represents a thread and each dotted arrow represents a discarded thread, that did not pass one of the previously explained tests. In this Section, we explain the *Parallel Edge-Collapse* algorithm steps in more detail.

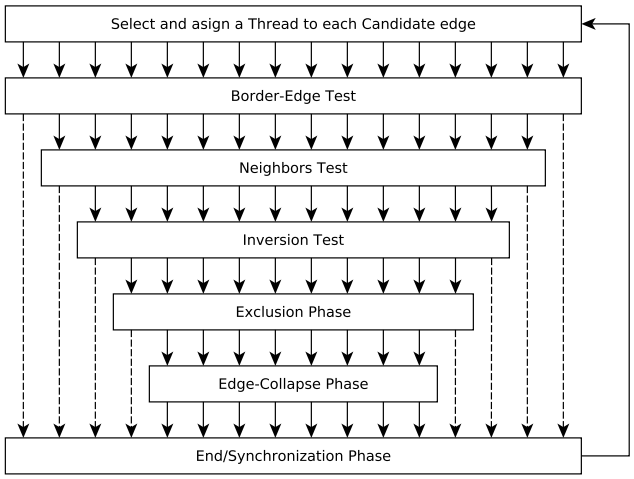


Figure 4.11: Overall Steps of the Simplification algorithm.

4.6.1 Elimination Phase

In this phase, we delete the data of the candidate edges from the Cleap data structure. This means deleting the data of its adjacent triangles and one of its vertices, and modifying the data of the other vertex.

For this objective, we launch a parallel kernel over all the candidate edges, as shown in Algorithms 11 and 12, performing the following steps:

1. Calculate $v_n = (v_1 + v_2)/2$, where v_1 and v_2 are the vertices of the edge.
2. Between v_1 and v_2, select the vertex which has the lowest ID in the vertex array.
   - In the lowest-ID vertex array, delete its coordinates by rewriting in that space the coordinates: $(INF, v_{\text{id}}, v_{\text{id}})$, where INF is a macro representing the value Infinity (which we use as a flag for the next phase) and v_{id} is the ID of the highest-ID vertex.
   - In the highest-ID vertex space, write the coordinates of v_n.
3. Obtain the IDs of the adjacent triangles of the candidate edge, and delete its associated information on the triangles array (rewriting them as $(0, 0, 0)$)
4. Finally, delete the information stored on this edge, rewriting all its fields with -1.

Algorithm 11: Edge Collapse Algorithm: Elimination Phase

```
   Data: vertices[ ], edges[ ], triangles[ ], candidateEdges[ ], adjTable[ ], windowSize
   Result: Void
 1 threadID = getID();
 2 if threadID < windowSize and candidateEdges[threadID] != -1 then
 3 |   edgeID = candidateEdges[threadID];
 4 |   v1,v2 = getVerticesID(edges[edgeID]);
 5 |   if v1 > v2 then
 6 |   |   vtemp = v1;
 7 |   |   v1 = v2;
 8 |   |   v2 = vtemp;
 9 |   end
10 |   TriA, TriB = getTrianglesID(edges[edgeID]);
11 |   neighbors table repair;
12 |   vertices[v2].x = (vertices[v1].x + vertices[v2].x)/2;
13 |   vertices[v2].y = (vertices[v1].y + vertices[v2].y)/2;
14 |   vertices[v2].z = (vertices[v1].z + vertices[v2].z)/2;
15 |   vertices[v1].x = INFINITY;
16 |   vertices[v1].y = v2;
17 |   vertices[v1].z = v2;
18 |   cleanUp();
19 |   return;
20 end
```

Algorithm 12: Edge Collapse Algorithm: Elimination Phase - Clean up step

```
Data: edges[ ], triangles[ ], edgeID, triangleID
Result: Void
1  triangles[3*triA] = 0;
2  triangles[3*triA+1] = 0;
3  triangles[3*triA+2] = 0;
4  triangles[3*triB] = 0;
5  triangles[3*triB+1] = 0;
6  triangles[3*triB+2] = 0;
7  edges[edgeID].a.x = -1;
8  edges[edgeID].a.y = -1;
9  edges[edgeID].b.x = -1;
10 edges[edgeID].b.y = -1;
11 edges[edgeID].n.x = -1;
12 edges[edgeID].n.y = -1;
13 edges[edgeID].op.x = -1;
14 edges[edgeID].op.y = -1;
15 return;
```

4.6.2 Triangle Repair Phase

In this phase, we repair all the inconsistencies that may be present in the triangles array. These inconsistencies are triangles that have a reference to a deleted vertex in the previous phase. To repair them, we use the flagged vertices $(INF, v_{\text{id}}, v_{\text{id}})$ from the previous phase. We construct the flagged vertices in that way because the INF value says that this vertex has been deleted and has relevant information to us, while the v_{id} value corresponds to the destination vertex or generated vertex resultant from the edge-collapse.

Then, with this information, a triangle can then detect if it has an inconsistent reference, and repair itself replacing the inconsistent vertex ID with the new v_{id}.

For this objective, we launch a parallel kernel over all the triangles, as shown in Algorithm 13, performing the following steps:

1. If we find a triangle with its 3 vertex ID equal to zero, then this means that this triangle has been deleted at some point in a previous iteration, and thus it is ignored.
2. If we find a triangle with its 3 vertex ID all distinct, then we have to look up its vertex references, and detect if they are flagged vertices or not (if a vertex ID references to a INF value).
    - If a vertex is not flagged, then the vertex still exists and is consistent. We do nothing.
    - If a vertex is flagged, then the vertex was deleted, and we need to repair that reference. We write the v_{id} stored on the deleted vertex, on the respective space in the triangles array.

Algorithm 13: Edge Collapse Algorithm: Triangle Repair Phase

```
   Data: vertices[ ], triangles[ ], triangleCount
   Result: Void
 1 threadID = getID();
 2 if threadID < triangleCount then
 3 |   triangleID = threadID;
 4 |   v1 = triangles[3*triangleID];
 5 |   v2 = triangles[3*triangleID+1];
 6 |   v3 = triangles[3*triangleID+2];
 7 |   if v1 == 0 or v2 == 0 or v3 == 0 then
 8 |   |   return;
 9 |   end
10 |   flag = vertices[v1].x;
11 |   if isinf(flag) then
12 |   |   newVertex = vertices[v1].y;
13 |   |   triangles[3*triangleID] = newVertex;
14 |   end
15 |   flag = vertices[v2].x;
16 |   if isinf(flag) then
17 |   |   newVertex = vertices[v2].y;
18 |   |   triangles[3*triangleID+1] = newVertex;
19 |   end
20 |   flag = vertices[v3].x;
21 |   if isinf(flag) then
22 |   |   newVertex = vertices[v3].y;
23 |   |   triangles[3*triangleID+2] = newVertex;
24 |   end
25 |   return;
26 end
```

4.6.3 Edge Repair Phase

In this phase, we repair all the inconsistencies that may be present on the edges array. These inconsistencies may be edges that have references to deleted vertices and/or deleted triangles.

For the case of references to deleted vertices, we look up the vertex references of the edge, and in a similar fashion to the previous phase, we detect if it is a flagged vertex or not. If it is flagged with the INF value, then we repair the reference on the edge, writing the consistent v_{id} value.

The second case is more complex, because if we find a triangle reference pointing to a deleted triangle (the ones that holds 3 zero references $(0, 0, 0)$), then this means, as a product of the edge-collapse, this edge has to merge with another edge (the other one that has the same inconsistent triangle reference). In addition, as we are performing a parallel procedure across all edges, the threads can not access and modify the data associated with other threads without incurring on race conditions that may alter the consistency of the mesh.

Instead of trying to repair the inconsistency now, we leave the necessary information so that a thread on the next phase can take a pair of edges that need to be merged and do the merge. Coincidentally, each of the deleted triangles on this pass of the algorithm are only referenced by the two edges that need to be merged, so we use these memory space to register the ID of these edges. To this purpose, the thread writes the edge ID on the two references to the triangle array that it has. It should be noted that one of the two references to the triangle is shared with the other edge that we have to merge, and writing the edge ID may cause some race conditions. This is not a problem, because at least one of each edge ID will be written on the triangle array (two references for one edge, and one reference to the other edge), leaving the triangle space with the data: (ID_A, ID_A, ID_B) or (ID_A, ID_B, ID_B) or any other similar permutation.

In summary we launch the parallel kernel shown in Algorithm 14 over all the edges and perform the next steps:

1. Look at vertices references, and detect if they are flagged vertices or not (if a vertex ID references to a INF value).
    - If a vertex is not flagged, then the vertex still exists and is consistent. We do nothing.
    - If a vertex is flagged, then the vertex was deleted, and we need to repair that reference. We write the v_{id} stored on the deleted vertex, on the respectively space in the edges array.
2. Look the triangle references, and detect if they reference a deleted triangle or not.
    - If the triangle has 3 distinct vertex ID, then the triangle still exists and it is consistent. We do nothing.
    - If the triangle has 3 zero IDs, then it is a deleted triangle, and means that two edges need to be merged in the next phase. We write the edge ID on the memory spaces pointed by the triangle references.

Algorithm 14: Edge Collapse Algorithm: Edge Repair Phase

```
   Data: vertices[ ], edges[ ], triangles[ ], edgeCount
   Result: Void
 1 threadID = getID();
 2 if threadID < edgeCount then
 3 |   edgeID = threadID;
 4 |   v1 = edges[edgeID].n.x;
 5 |   v2 = edges[edgeID].n.y;
 6 |   if v1 == v2 then
 7 |   |   return;
 8 |   end
 9 |   if isinf(vertices[v1].x) then
10 |   |   edges[edgeID].n.x = vertices[v1].y;
11 |   end
12 |   if isinf(vertices[v2].x) then
13 |   |   edges[edgeID].n.y = vertices[v2].y;
14 |   end
15 |   if edges[edgeID].n.x == -1 and edges[edgeID].n.y == -1 then
16 |   |   return;
17 |   end
18 |   TriA1 = edges[edgeID].a.x;
19 |   TriA2 = edges[edgeID].a.y;
20 |   TriB1 = edges[edgeID].b.x;
21 |   TriB2 = edges[edgeID].b.y;
22 |   if TriA1 != -1 and TriA2 != -1 then
23 |   |   Trindex = (int) TriA1/3;
24 |   |   v1 = triangles[3*Trindex];
25 |   |   v2 = triangles[3*Trindex+1];
26 |   |   v3 = triangles[3*Trindex+2];
27 |   |   if v1 = v2 or v2 == v3 or v3 == v1 then
28 |   |   |   triangles[triA1] = edgeID;
29 |   |   |   triangles[triA2] = edgeID;
30 |   |   end
31 |   end
32 |   if TriB1 != -1 and TriB2 != -1 then
33 |   |   Trindex = (int) TriB1/3;
34 |   |   v1 = triangles[3*Trindex];
35 |   |   v2 = triangles[3*Trindex+1];
36 |   |   v3 = triangles[3*Trindex+2];
37 |   |   if v1 = v2 or v2 == v3 or v3 == v1 then
38 |   |   |   triangles[triB1] = edgeID;
39 |   |   |   triangles[triB2] = edgeID;
40 |   |   end
41 |   end
42 |   return;
43 end
```

4.6.4 Connectivity Repair Phase

In this phase, we repair the last inconsistency left from the previous phase, the partially duplicated edges that need to be merged. For this purpose, we search across all triangles, the ones that have the shape (A, A, B) or (A, B, B) or other permutations. Instead of searching all possible combinations, we equivalently search for the triangles that have two identical vertex ID references distinct from zero on two of its three ID spaces.

This case means that this triangle is not a triangle, but the memory space that we constructed on the previous phase, and this thread has the job of merging the edges referenced by the IDs stored in this triangle. This merge extracts the two edge IDs, and writes the non-deleted-triangles references of the edge with the highest edge-ID into the deleted-triangle references of the edge with smallest edge-ID. Finally, it deletes all the data associated with the edge with the highest ID (to delete it effectively), and rewrites the references of the triangle handled by this thread to $(0, 0, 0)$, to denote that this triangle is finally deleted.

In summary, we launch a parallel kernel over all triangles, performing the following steps:

1. Look if two of the three vertex ID that the triangle has are the same.
    - If all three are distinct, then the triangle is consistent and we do nothing.
    - Otherwise, this triangle stores two edge IDs and needs to merge its references.
        - On the memory space of the deleted triangle of the edge with the smaller ID, write the triangle references of the edge with higher ID that is not deleted.
        - Delete all the references on the edge with higher ID
        - Rewrite this triangle references to zero.

The code related to this step is shown on Algorithm 15. The *caseDetection* routine on line 32 is shown on Algorithm 16, and helps detect which one of the four merge cases the edges are, and perform it. The *cleanUp* routine on line 33 is shown on Algorithm 17, and helps to effectively delete all the edges and triangles because of the edge-collapse at the end of the parallel steps.

When this last phase is performed, it concludes a parallel iteration of the edge-collapse algorithm. Appendix C displays a graphical example of how the Cleap data structure that holds the information of vertices, triangles and edges is updated in each of the four steps of the parallel simplification algorithm.

Algorithm 15: Edge Collapse Algorithm: Connectivity Repair Phase

```
   Data: vertices[ ], edges[ ], triangles[ ], triangleCount
   Result: Void
 1 threadID = getID();
 2 if threadID < triangleCount then
 3 |   triangleID = threadID;
 4 |   v1 = triangles[3*triangleID];
 5 |   v2 = triangles[3*triangleID+1];
 6 |   v3 = triangles[3*triangleID+2];
 7 |   if v1==v2 and v2!=v3 then
 8 |   |   edgeID1 = v1;
 9 |   |   edgeID2 = v3;
10 |   else
11 |   |   if v2==v3 and v1!=v3 then
12 |   |   |   edgeID1 = v2;
13 |   |   |   edgeID2 = v1;
14 |   |   else
15 |   |   |   if v3==v1 and v2!=v3 then
16 |   |   |   |   edgeID1 = v3;
17 |   |   |   |   edgeID2 = v2;
18 |   |   |   else
19 |   |   |   |   return;
20 |   |   |   end
21 |   |   end
22 |   end
23 |   if edgeID1 > edgeID2 then
24 |   |   temp = edgeID1;
25 |   |   edgeID1 = edgeID2;
26 |   |   edgeID2 = temp;
27 |   end
28 |   repAe1 = (int) edges[edgeID1].a.x/3;
29 |   repBe1 = (int) edges[edgeID1].b.x/3;
30 |   repAe2 = (int) edges[edgeID2].a.x/3;
31 |   repBe2 = (int) edges[edgeID2].b.x/3;
32 |   caseDetection(edges, edgeID1, edgeID2, repAe1, repBe1, repAe2, repBe2);
33 |   cleanUp(edges, triangles, edgeID1, triangleID);
34 |   return;
35 end
```

Algorithm 16: Edge Collapse Algorithm: Connectivity Repair Phase - Case detection step

```
Data: edges[ ], edgeID1, edgeID2, repAe1, repBe1, repAe2, repBe2
Result: Void
if repAe1 == repAe2 then
    edges[edgeID2].a.x = edges[edgeID1].b.x;
    edges[edgeID2].a.y = edges[edgeID1].b.y;
    edges[edgeID2].op.x = edges[edgeID1].op.y;
else
    if repAe1 == repBe2 then
        edges[edgeID2].b.x = edges[edgeID1].b.x;
        edges[edgeID2].b.y = edges[edgeID1].b.y;
        edges[edgeID2].op.y = edges[edgeID1].op.y;
    else
        if repBe1 == repAe2 then
            edges[edgeID2].a.x = edges[edgeID1].a.x;
            edges[edgeID2].a.y = edges[edgeID1].a.y;
            edges[edgeID2].op.x = edges[edgeID1].op.x;
        else
            if repBe1 == repBe2 then
                edges[edgeID2].b.x = edges[edgeID1].a.x;
                edges[edgeID2].b.y = edges[edgeID1].a.y;
                edges[edgeID2].op.y = edges[edgeID1].op.x;
            else
                return;
            end
        end
    end
end
```

Algorithm 17: Edge Collapse Algorithm: Connectivity Repair Phase - Clean up step

```
Data: edges[ ], triangles[ ], edgeID, triangleID
Result: Void
edges[edgeID].a.x = -1;
edges[edgeID].a.y = -1;
edges[edgeID].b.x = -1;
edges[edgeID].b.y = -1;
edges[edgeID].n.x = -1;
edges[edgeID].n.y = -1;
edges[edgeID].op.x = -1;
edges[edgeID].op.y = -1;
triangles[3*triangleID] = 0;
triangles[3*triangleID+1] = 0;
triangles[3*triangleID+2] = 0;
return;
```

4.7 Algorithm Output

After all candidates are evaluated or all possible edge-collapses are performed, we need to generate the final output of the algorithm onto a file (the last step on Figure 4.1). The problem is we cannot make a dump of the data stored on the data structures to a .OFF or .PLY file as is, because we have seen in Section 4.6 that our parallel edge-collapse algorithm left behind empties spaces between consistent data, in the arrays of vertices, edges and faces, corresponding to the deleted elements.

These empties spaces (also referred to as gaps), especially in the vertices array, are a problem, because the information of edges and faces on the .OFF and .PLY files, are stored considering the relative position of the vertices in the file (as can be seen in Section 2.4), which does not allow a missing index between two existing indices.

It should be noted that these empty spaces does not affect the algorithm operation, because as shown in Section 4.6, each phase handles the cases when it encounters a deleted or missing element, ignoring it or using its space as a temporary memory for its own operation.

In order to fill these empty spaces and allow the mesh to be saved to a file as these gaps never existed, we need to perform a one-time step after the edge-collapse portion of the algorithm ends. For this purpose, we need to reorder the elements of the data structure to fill these gaps on the data structure, and then fix the relative references between the elements of the data structure. In the following subsections we explain how we do that.

4.7.1 Vertices Rearrange

First, we notice that the most critical gaps are the ones present on the vertices array, because the edges and triangles have relative references to the indices on this array. For the purpose of dumping the mesh into a .OFF or .PLY file, we can leave the gaps on the faces and edges array, because these files only need to store its relative references to the vertices array to save faces and edges data. Meanwhile, the vertices array need to be stored without gaps at all, so we need to repair this array, filling its gaps before saving it into a file.

In order to repair the vertices array, we need to do some kind of data shift with the elements of the array to fill the gaps. A first approach would be to shift the data from the right side of the array into the nearest empty space to its left. The problem is that this approach would need a high number of write and swap operations, and the number of rewrites needed in each shift grows linearly. A second approach (and the one that we will take), is to perform a swap of the elements on the right side of the array, into the empty spaces at the left side of the array. This approach works, because we only need the vertices array to be without gaps, and not necessarily preserve the previous relative order of the vertices. For this purpose, we call this procedure *vertex-swap*.

Each time a *vertex-swap* is performed, we have to update the swapped vertex reference across all triangles and edges on its respective arrays in the data structure, because the *vertex-swap* changes the relative vertex ID of the swapped vertex.

To do this rearrangement, we use a hybrid approach, using the CPU and the GPU (Algorithm 18), in which we iterate through all vertices with the CPU, and when a *vertex-swap* is performed, then we update the references on the edges and triangles array in parallel with the GPU.

We decided to use a hybrid approach, because doing the references updates on the CPU were consuming a lot of time on early experimentations. Each time a *vertex-swap* was performed, we needed to iterate across all edges and triangles updating the references, which escalates exponentially as the size of the mesh grows on elements. Because this step was not part of the main steps of the simplification algorithm, we decided to implement the update iterations in parallel with a fixed number of 256 threads to decrease the time needed to generate the output file of the algorithm.

This algorithm summarizes as follows:

1. We perform the *vertex-swaps* sequentially on the CPU (to prevent race conditions on concurrent swaps). We have two pointers: one starting from the left-side of the array, looking for the first empty space to its right (we call this the *red pointer*), and other one starting from the right-side on the array, looking for the first occupied space on the array (we call this the *blue pointer*).
2. When each pointer finds its objective, we perform a *vertex-swap*, writing the data of the vertex referenced by the *blue pointer* into the space referenced by the *red pointer*, and then delete the data on the *blue pointer*.
3. Then, in parallel with a GPU kernel over the edges and faces array, we update the vertices references referencing to the space of the *blue pointer*, to the space referenced by the *red pointer* (Algorithms 19 and 20).
4. Each pointer continues its respective search from the place where they are, and repeat the step 2.
5. If at some point during the search the *red pointer* surpasses the *blue pointer*, it means that all necessary swaps have been performed, and this phase ends.

Appendix D displays a graphical example of the previous steps. This rearrangement only need to be performed once, when all edge-collapses have been performed.

4.7.2 Save into a File

At this stage, the references on the data structure are repaired, and it is ready to be saved into a file. We need to create a file, following the specifications defined in Section 2.4. For the .OFF file case, as shown in Section 2.4.1, we write in the first line, the OFF header, then in the second line, the number of vertices, faces, and edges of the mesh. Then in the next V lines (where V is the number of vertices) we write the coordinates of each vertex, one per line, from the vertex array.

Finally, we read the triangles array, writing one triangle per line, as references to its vertices in the order they were written previously on the file. As for the .OFF file format, we do not need to write the edges data, because it is recorded implicitly in the vertices and faces.

For the .PLY file case, as shown in Section 2.4.2 and following a similar procedure of the .OFF case, we write the PLY header on the first line and the format on the second line (ASCII or binary). Then, we define the meta-data of vertices faces and edges (number of elements of each type, and additional properties, such as color, if needed). Next, in the following V lines (where V is the number of vertices) we write the coordinates of each vertex, one per line, from the vertex array. Then, in the next F lines (where F is the number of faces) we read the triangles array, writing one triangle per line, as references to its vertices in the order they were written previously on the file. Finally, in the next E lines (where E is the number of edges), we read the edges array, and write one edge per line, as the references to its respective vertices.

Algorithm 18: Algorithm Output: Vertices Rearrange

```
   Data: vertices[ ], triangles[ ], edges[ ], vertexCount, triangleCount, edgeCount
   Result: Void
 1 i = 0;
 2 j = vertexCount - 1;
 3 while i < j do
 4     while true do
 5         if isinf(vertices[i].x) // i >= j then
 6             break;
 7         end
 8         i++;
 9     end
10     while true do
11         if !isinf(vertices[j].x) // j >= i then
12             break;
13         end
14         j–;
15     end
16     if i >= j then
17         break;
18     end
19     vertices[i].x = vertices[j].x;
20     vertices[i].y = vertices[j].y;
21     vertices[i].z = vertices[j].z;
22     vertices[j].x = INFINITY;
23     vertices[j].y = INFINITY;
24     vertices[j].z = INFINITY;
25     parallelTriangleRepair(triangles,triangleCount,j,i);
26     parallelEdgeRepair(edges,edgeCount,j,i);
27 end
```

Algorithm 19: Algorithm Output: Vertices Rearrange - Parallel Triangle Repair

```
   Data: triangles[ ], triangleCount, oldIndex, newIndex
   Result: Void
 1 threadID = getID();
 2 if threadID < triangleCount then
 3 |   triangleID = threadID;
 4 |   if triangles[3*triangleID] == oldIndex then
 5 |   |   triangles[3*triangleID] = newIndex;
 6 |   end
 7 |   if triangles[3*triangleID+1] == oldIndex then
 8 |   |   triangles[3*triangleID+1] = newIndex;
 9 |   end
10 |   if triangles[3*triangleID+2] == oldIndex then
11 |   |   triangles[3*triangleID+2] = newIndex;
12 |   end
13 end
```

Algorithm 20: Algorithm Output: Vertices Rearrange - Parallel Edge Repair

```
   Data: edges[ ], edgeCount, oldIndex, newIndex
   Result: Void
 1 threadID = getID();
 2 if threadID < edgeCount then
 3 |   edgeID = threadID;
 4 |   if edges[edgeID].n.x == oldIndex then
 5 |   |   edges[edgeID].n.x = newIndex;
 6 |   end
 7 |   if edges[edgeID].n.y == oldIndex then
 8 |   |   edges[edgeID].n.y = newIndex;
 9 |   end
10 end
```

Chapter 5

Results

To test the performance of the proposed algorithm, we have planned a set of experiments to evaluate its robustness and efficiency. It is known from the algorithms review on Section 3.3 that if the algorithm employs a number of threads equal to the number of edges that need to be collapsed, the mutual exclusion mechanism can leave neighbor zones identified for simplification but later discarded without collapsing an edge at all. For this reason, the test considers measuring the time the algorithm takes using 64, 128, 256 and 512 threads per block.

We execute the algorithm under a test mesh (Figure 5.2), with different conditions and parameters on the algorithm. Because we could not get the implementation of the algorithms reviewed in Section 3.3, we do the comparison with an ad-hoc serial implementation of the algorithm. The test consists of:

- We have a mesh that represents a real scenario terrain (from Chile central valley), with 998001 vertices and 1992008 triangles.
- For this mesh, we execute the algorithm varying the following parameters:
  - Size of the maximum target edge-collapses over the mesh.
  - Size of the threads per block, for the case of the parallel algorithm.

Table 5.1 shows the characteristics of the machine where the test were realized.

Table 5.1: Hardware and S.O. specs of the test platform.

Item	Detail
CPU	Intel Core i5-6200U @2.3GHz
GPU	Nvidia GeForce 945M @928MHz with 2GB VRAM, 4 SMP, 512 CUDA cores
RAM	20GB DDR4-2133 @2133MHz
S.O.	Ubuntu 15.10 64bits kernel 4.2.0-42-generic

In the next sections, we show the results of our algorithm. We separate our results in Quantitative Results (focused on comparing execution times, number of iterations, etc) and Qualitative Results (focused on the output of the algorithms and how we perceive the quality of the resultant mesh). Finally, we show an analysis of those results.

5.1 Quantitative Results

In this Section, we show the results obtained by the parallel algorithm, and its comparison with the sequential implementation. We have measured the run-time of the algorithm in its edge-collapse phase and its final repair step (vertices rearrange phase) before generating the output, and the number of iterations performed under different maximum edge-collapse targets that we want to perform (50,000, 100,000, 200,000, 300,000 and 400,000 total collapsed edges).

For the parallel algorithm, we have measured its performance with a varying number of threads per block (64, 128, 256, 512). Additionally, we have measured the number of threads that effectively performed and edge-collapse passing all the filters and lock-downs (occupancy factor), with the objective to determine which number of threads per block reduces the number of idle threads per iteration, where an idle thread is defined as one that can not perform its assigned edge-collapse, because it failed one of the consistency test or concurrency test.

First, we present time measures (run-time and number of iterations) for the sequential algorithm and the parallel algorithm with different number of threads. Next, we present the *speed-up* factor between the sequential algorithm and each of the parallel variants of the algorithm. Next, we present some charts showing the run-time and number of iterations comparison of the algorithms under different edge-collapse targets. Finally, we present the number of effective edge-collapse performed per iteration, under different combinations of threads per iteration, and the maximum required edge-collapses.

All measures are performed over the mesh presented in Figure 5.2, which is a good test case, because it has a mixture of valley areas and mountain areas from the central valley of Chile (Figure 5.1), with the objective to find if our algorithm has a good performance over surface terrain meshes.

Figure 5.1: Overview of Chile central valley. Image taken from Google Maps.

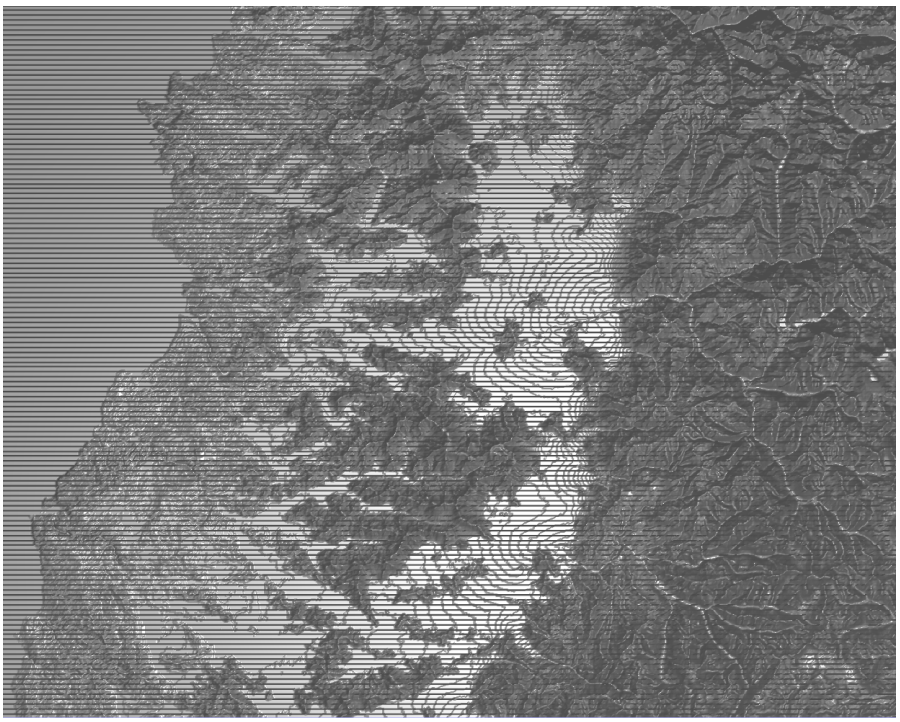


Figure 5.2: Overview of the mesh that represents the place on Figure 5.1

Table 5.2: Time measurement for the sequential edge-collapse algorithm.

	Time measures - Sequential algorithm	
Max edge-collapses	# Iterations	Algorithm time [s]
50,000	93,345	8,152
100,000	199,360	16,370
200,000	450,673	33,276
300,000	754,361	50,780
400,000	1,110,420	68,894

Table 5.3: Time measurement for the parallel edge-collapse algorithm at 64-thread window

	Time measures - Parallel algorithm 64 Threads	
Max edge-collapses	# Iterations	Algorithm time [s]
50,000	1,645	15.20
100,000	3,777	36.27
200,000	9,461	89.54
300,000	16,860	174.44
400,000	26,828	263.27

Table 5.4: Time measurement for the parallel edge-collapse algorithm at 128-thread window

	Time measures - Parallel algorithm 128 Threads	
Max edge-collapses	# Iterations	Algorithm time [s]
50,000	820	7.88
100,000	1,904	19.13
200,000	4,747	47.27
300,000	8,464	86.32
400,000	13,429	129.85

Table 5.5: Time measurement for the parallel edge-collapse algorithm at 256-thread window

	Time measures - Parallel algorithm 256 Threads	
Max edge-collapses	# Iterations	Algorithm time [s]
50,000	410	4.19
100,000	948	8.60
200,000	2,374	23.41
300,000	4,221	43.35
400,000	6,727	64.66

Table 5.6: Time measurement for the parallel edge-collapse algorithm at 512-thread window

	Time measures - Parallel algorithm 512 Threads	
Max edge-collapses	# Iterations	Algorithm time [s]
50,000	207	1.85
100,000	478	4.34
200,000	1,189	11.79
300,000	2,121	18.87
400,000	3,367	33.19

Table 5.7: Time measurement for the final vertices rearrange step, sequential and parallel with 256 threads.

	Time Measurements - Vertices Rearrange Step	
Max edge-collapses	Sequential [s]	Parallel [s]
50000	1,002	125
100000	1,908	253
200000	3,440	466
300000	4,597	616
400000	5,398	744

Table 5.8: *Speed-up* of the Parallel algorithm in comparison with the Sequential algorithm on its edge-collapse phase.

	Speed-up Parallel algorithm/ Sequential algorithm			
Max edge-collapses	64 threads	128 threads	256 threads	512 threads
50,000	536.31	1,034.51	1,945.58	4,406.48
100,000	451.33	855.72	1,903.48	3,771.88
200,000	371.63	703.95	1,421.44	2,822.39
300,000	291.10	588.27	1,171.39	2,691.04
400,000	261.68	530.56	1,065.48	2,075.74

Table 5.9: *Speed-up* of the Parallel algorithm in comparison with the Sequential algorithm, considering the vertices rearrangement step.

	Speed-up Parallel algorithm/ Sequential algorithm			
Max edge-collapses	64 threads	128 threads	256 threads	512 threads
50,000	65.29	68.88	70.85	72.16
100,000	63.18	67.16	69.87	71.02
200,000	66.09	71.53	75.02	76.84
300,000	70.05	78.84	83.98	87.22
400,000	73.75	85.01	91.87	95.59

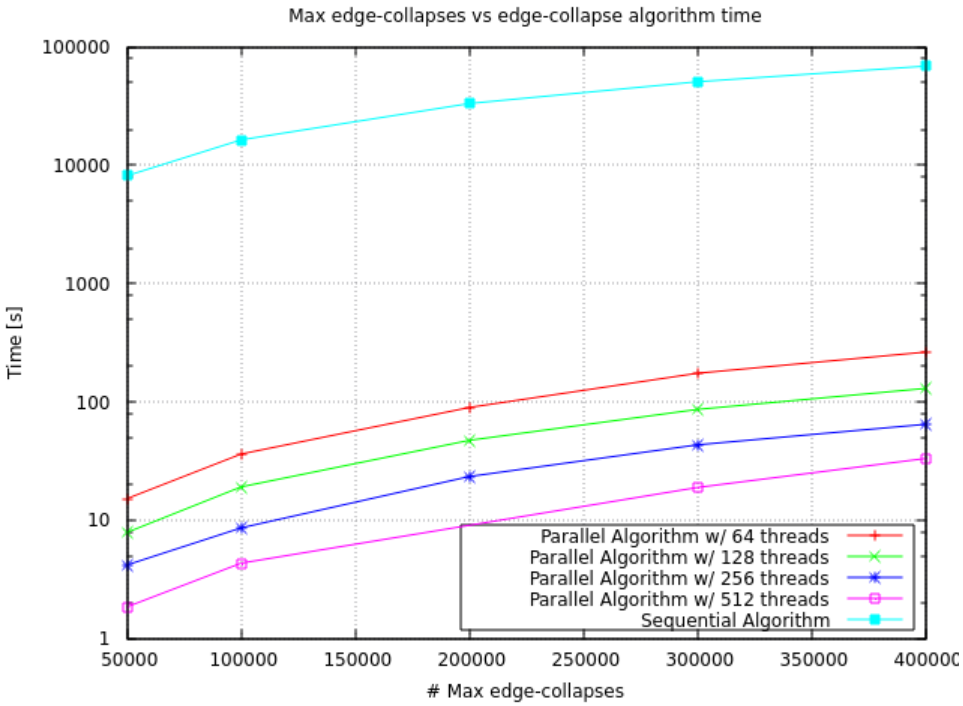


Figure 5.3: Run-time of different variants of the algorithm on its edge-collapse phase for different maximum edge-collapse targets. Time is on logarithmic scale.

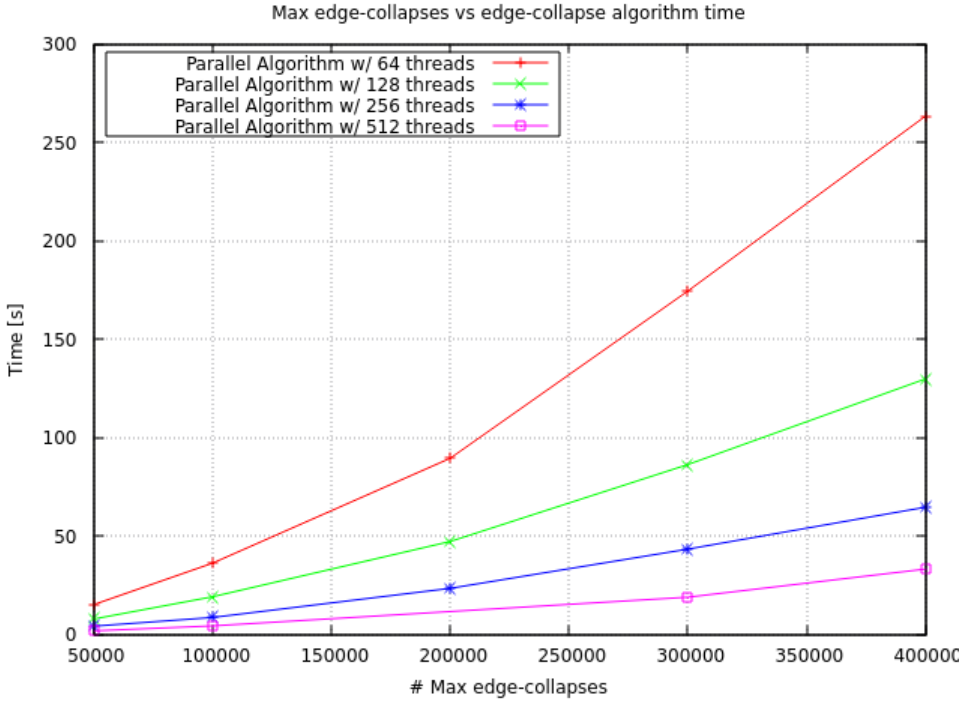


Figure 5.4: Run-time of parallel variants of the algorithm on its edge-collapse phase for different maximum edge-collapse targets.

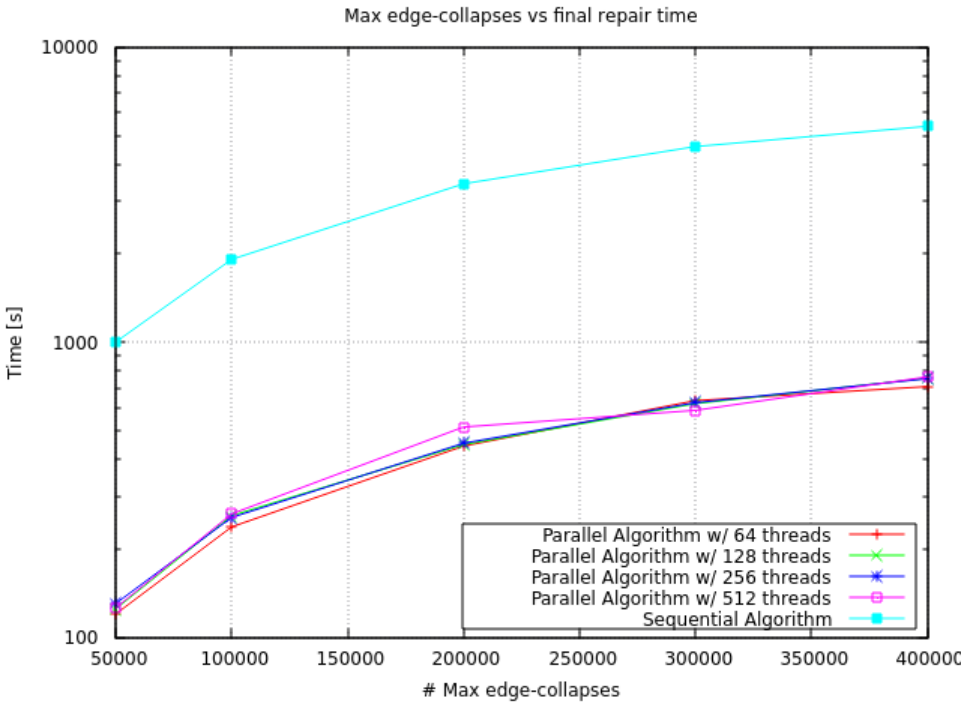


Figure 5.5: Run-time of different variants of the edge-collapse algorithm on its final vertices rearrange step for different maximum edge-collapse targets. Time is on logarithmic scale.

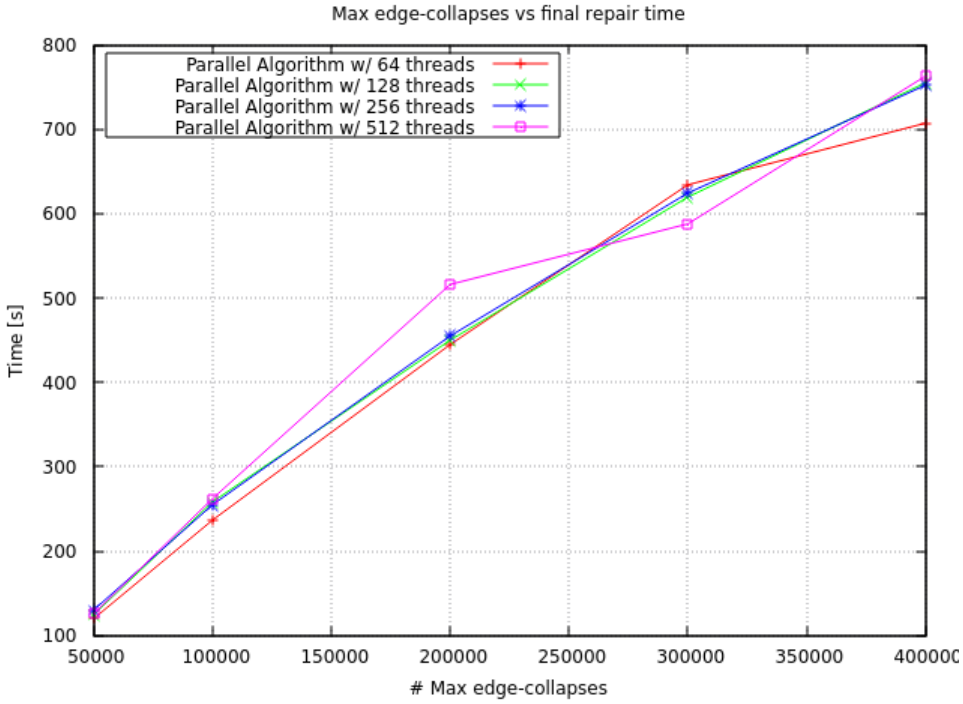


Figure 5.6: Run-time of parallel variants of the edge-collapse algorithm on its final vertices rearrange step for different maximum edge-collapse targets.

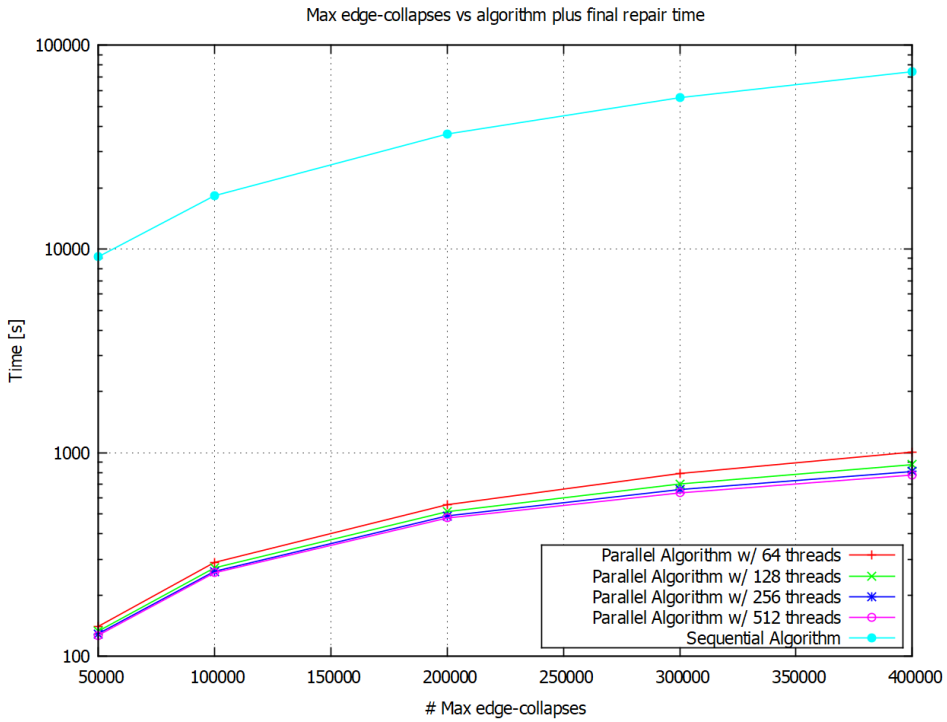


Figure 5.7: Run-time of different variants of the edge-collapse algorithm on its main phase plus final vertices rearrange step for different maximum edge-collapse targets. Time is on logarithmic scale.

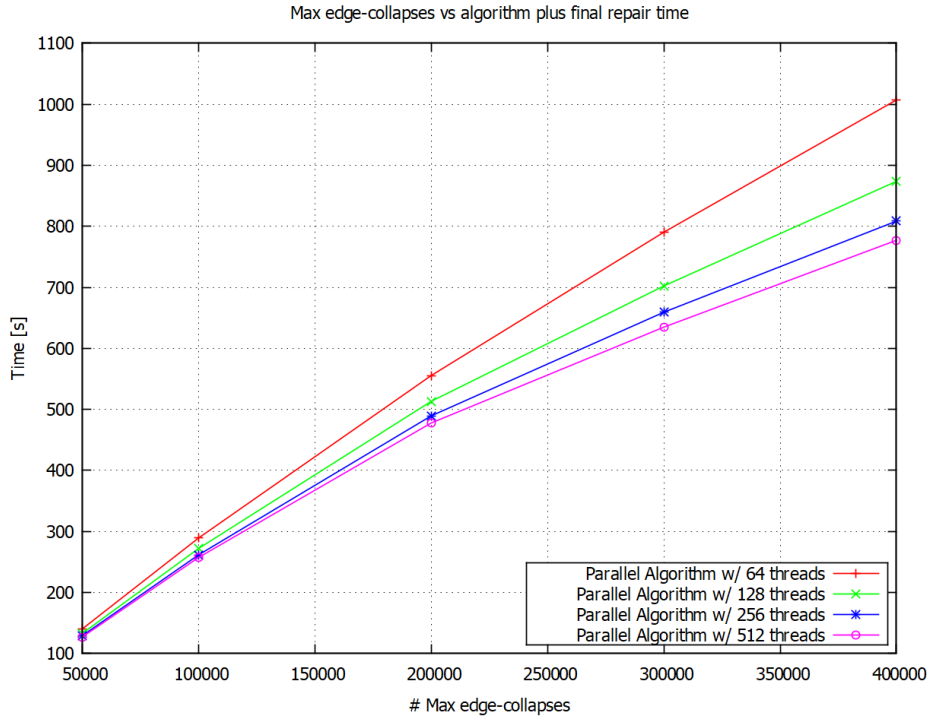


Figure 5.8: Run-time of parallel variants of the edge-collapse algorithm on its main phase plus final vertices rearrange step for different maximum edge-collapse targets.

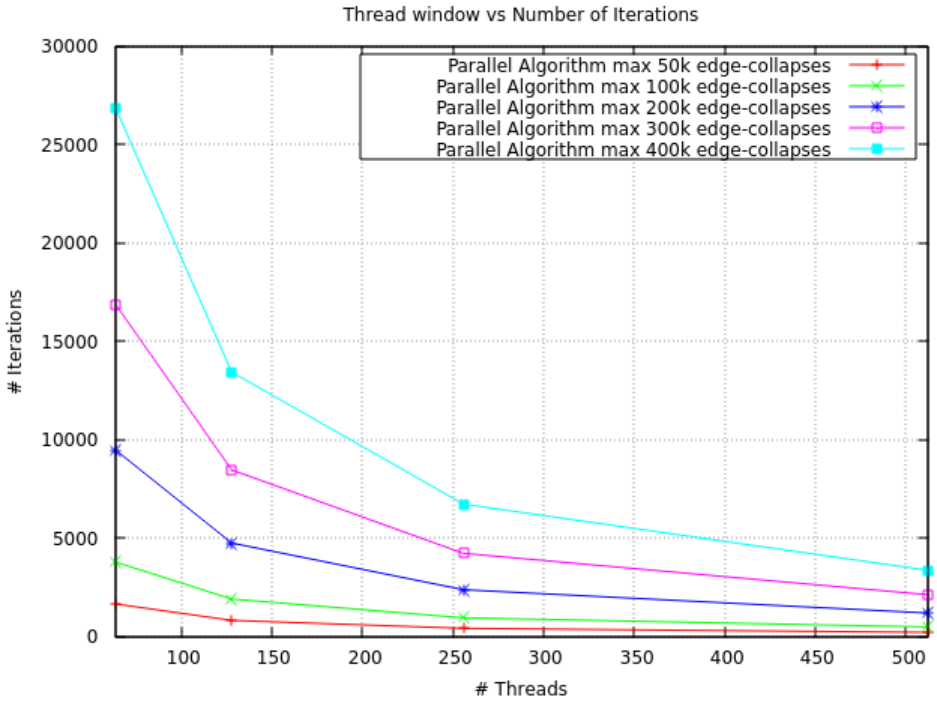


Figure 5.9: Number of iterations of the parallel algorithm needed for a determined number of threads with a fixed number of maximum edge-collapse targets.

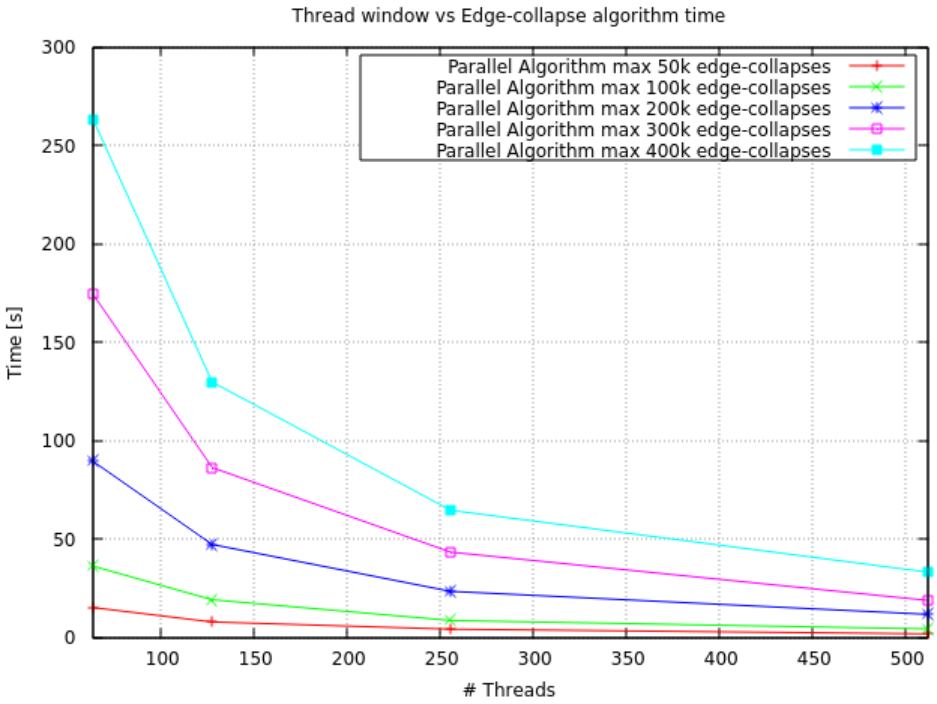


Figure 5.10: Run-time of the parallel algorithm (edge-collapse phase) for a determined number of threads with a fixed number of maximum edge-collapse targets.

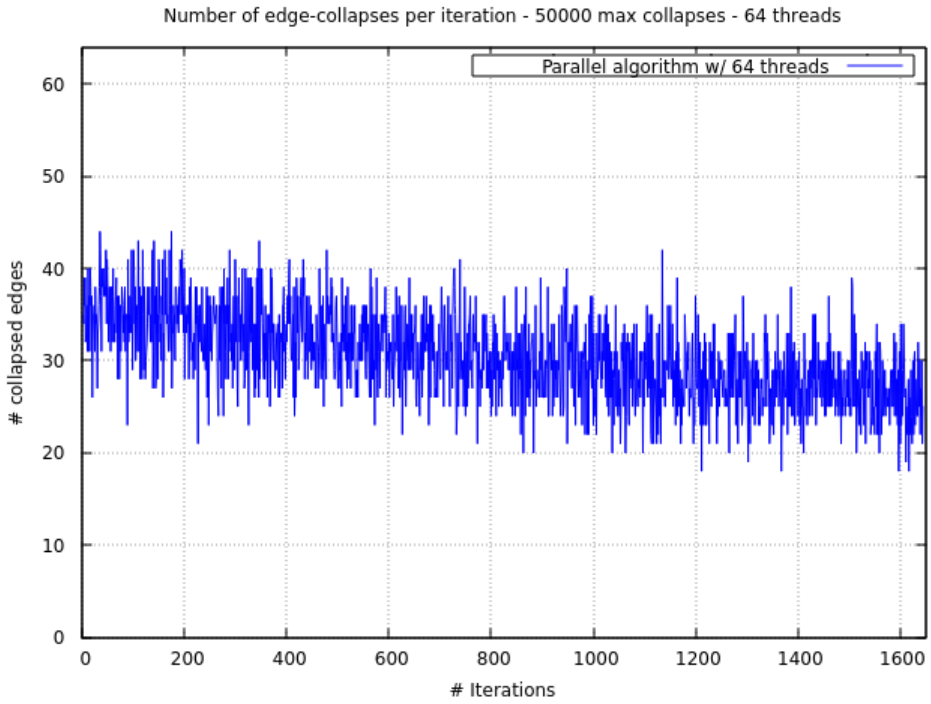


Figure 5.11: Number of edge-collapses per iteration of the Parallel algorithm with 64 threads. Target of 50,000 edge-collapses.

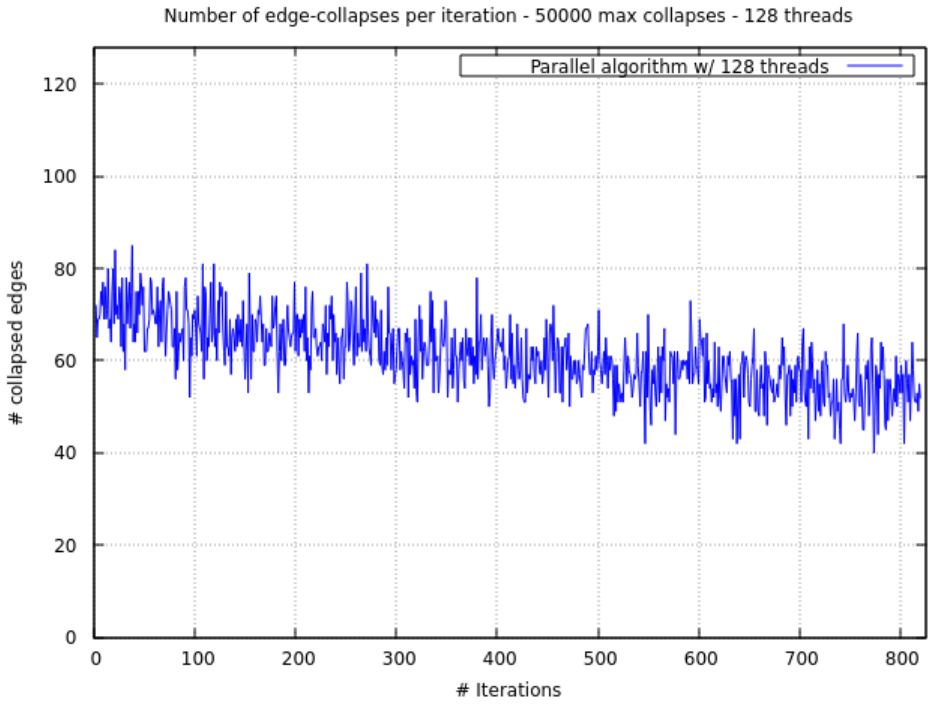


Figure 5.12: Number of edge-collapses per iteration of the Parallel algorithm with 128 threads. Target of 50,000 edge-collapses.

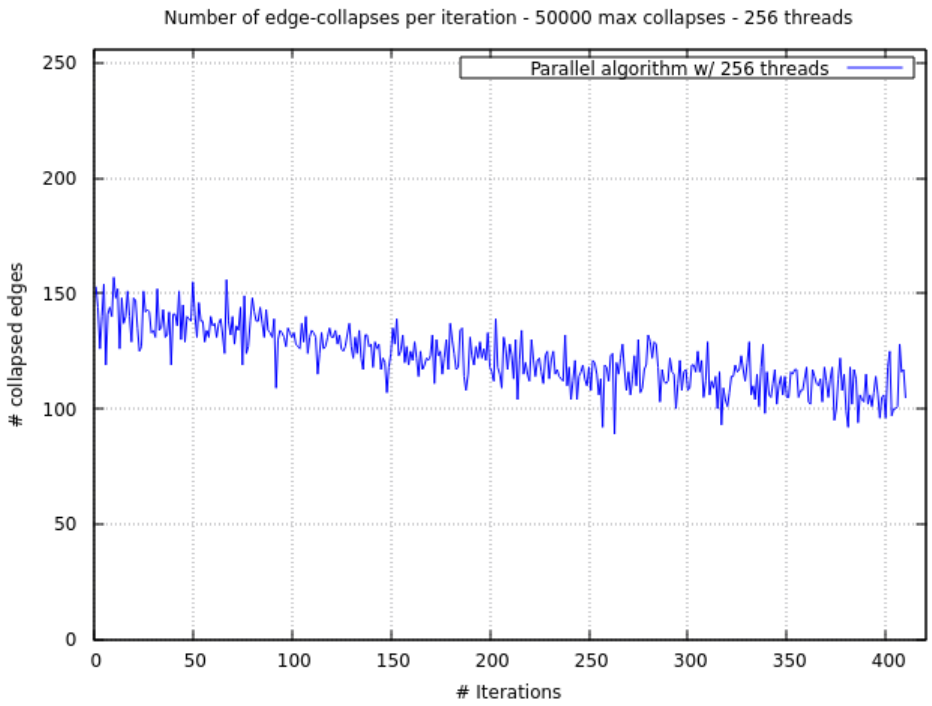


Figure 5.13: Number of edge-collapses per iteration of the Parallel algorithm with 256 threads. Target of 50,000 edge-collapses.

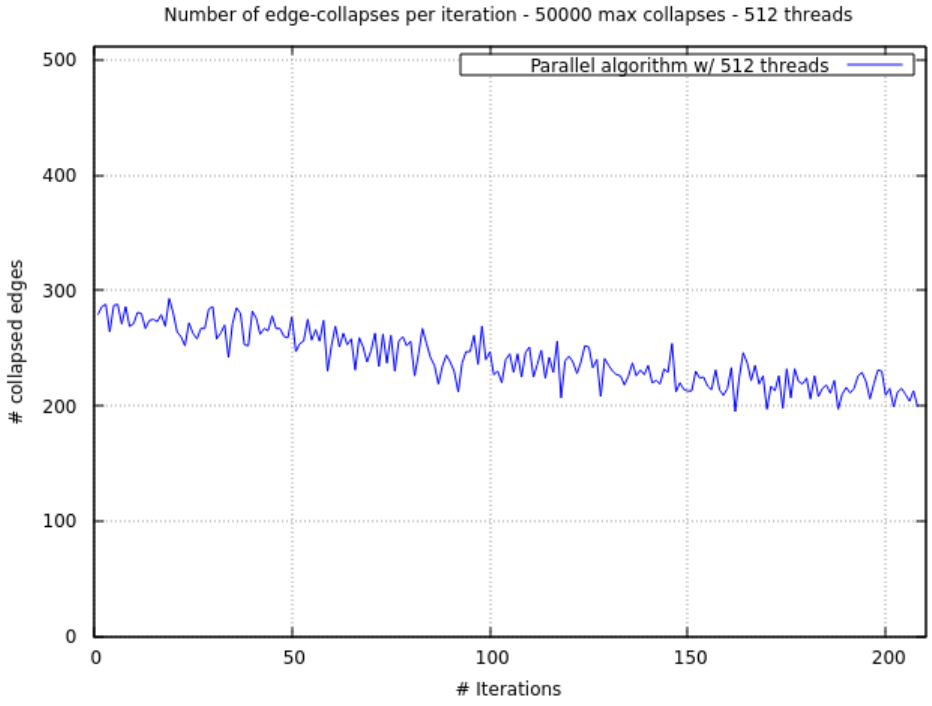


Figure 5.14: Number of edge-collapses per iteration of the Parallel algorithm with 512 threads. Target of 50,000 edge-collapses.

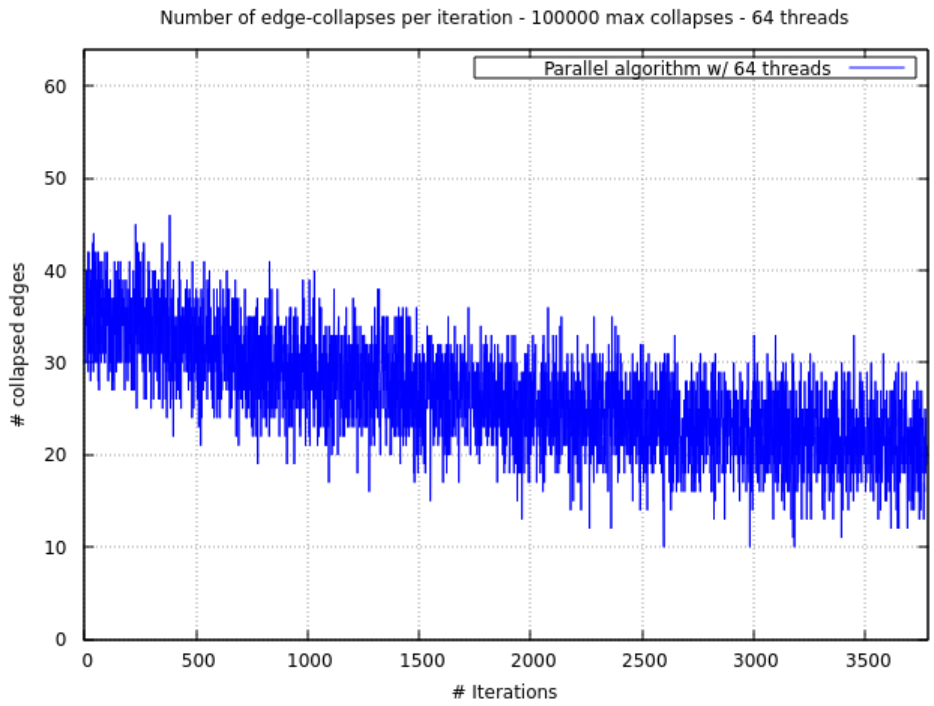


Figure 5.15: Number of edge-collapses per iteration of the Parallel algorithm with 64 threads. Target of 100,000 edge-collapses.

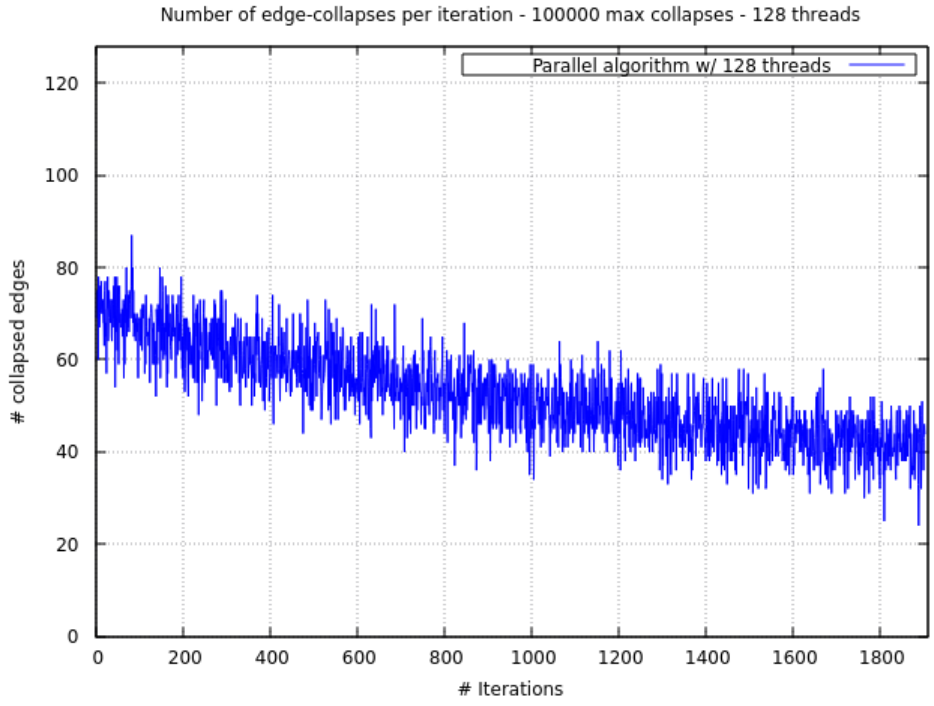


Figure 5.16: Number of edge-collapses per iteration of the Parallel algorithm with 128 threads. Target of 100,000 edge-collapses.

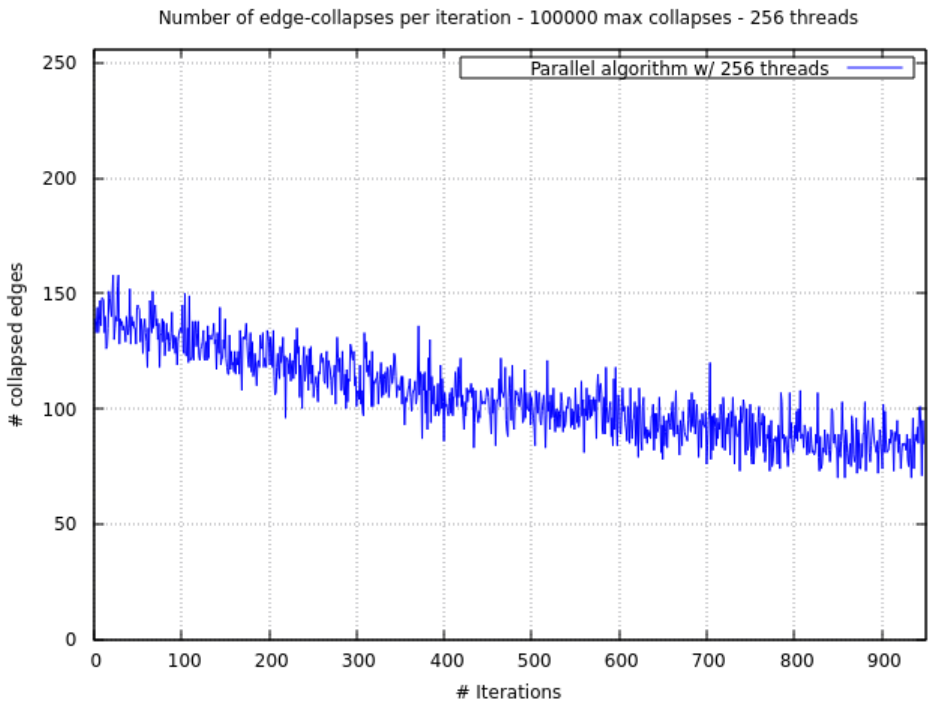


Figure 5.17: Number of edge-collapses per iteration of the Parallel algorithm with 256 threads. Target of 100,000 edge-collapses.

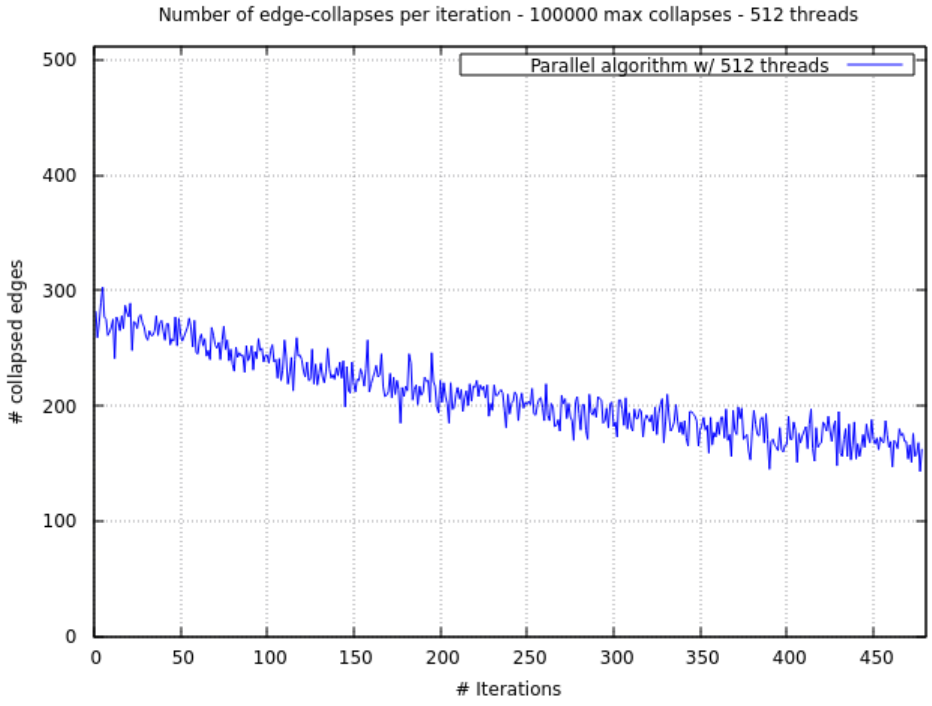


Figure 5.18: Number of edge-collapses per iteration of the Parallel algorithm with 512 threads. Target of 100,000 edge-collapses.

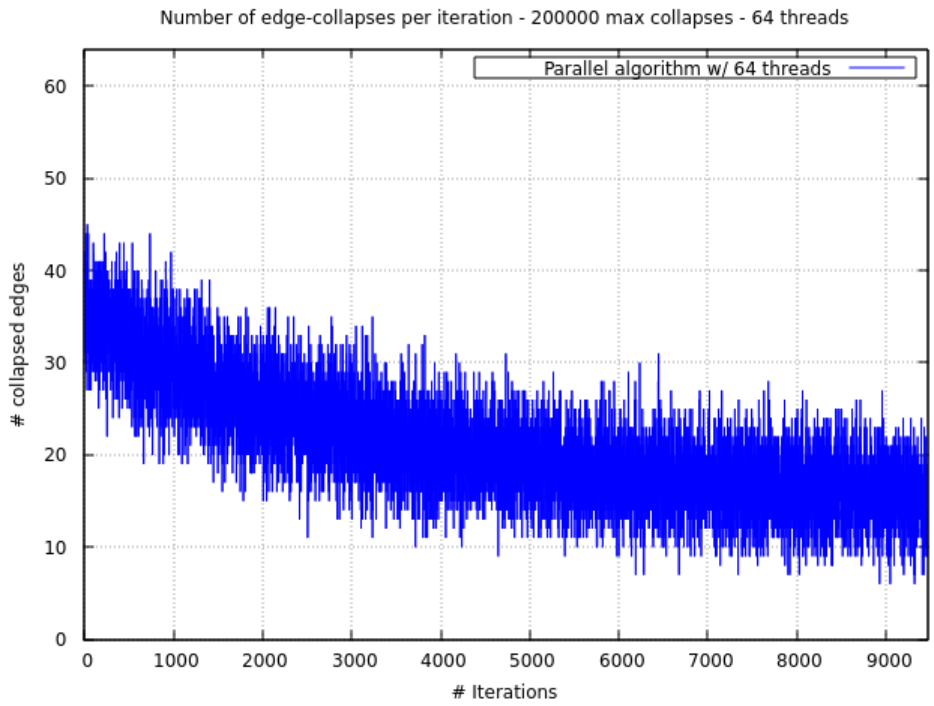


Figure 5.19: Number of edge-collapses per iteration of the Parallel algorithm with 64 threads. Target of 200,000 edge-collapses.

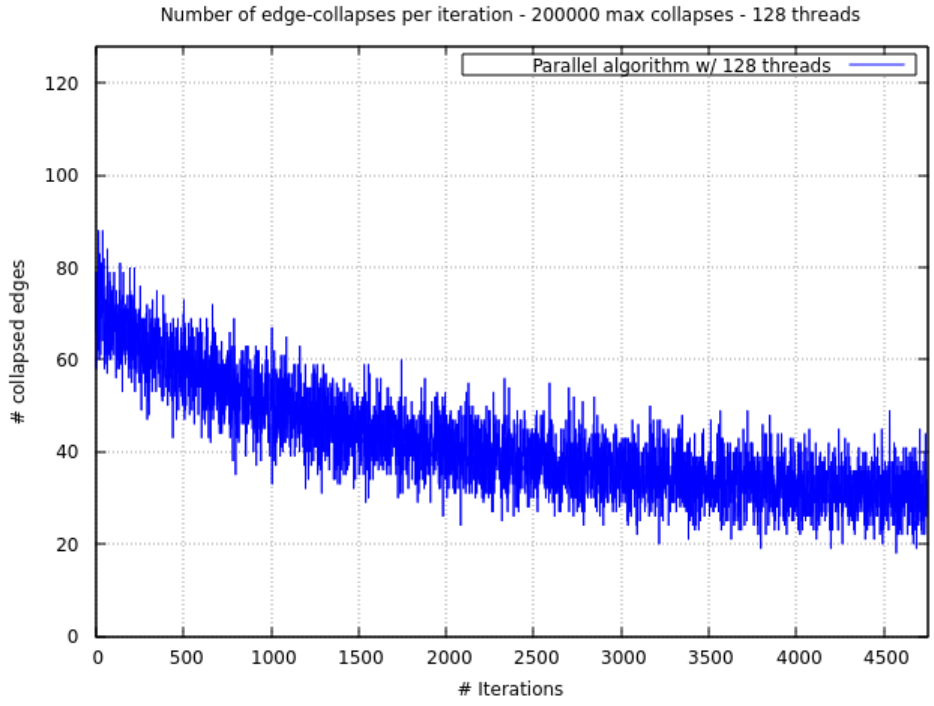


Figure 5.20: Number of edge-collapses per iteration of the Parallel algorithm with 128 threads. Target of 200,000 edge-collapses.

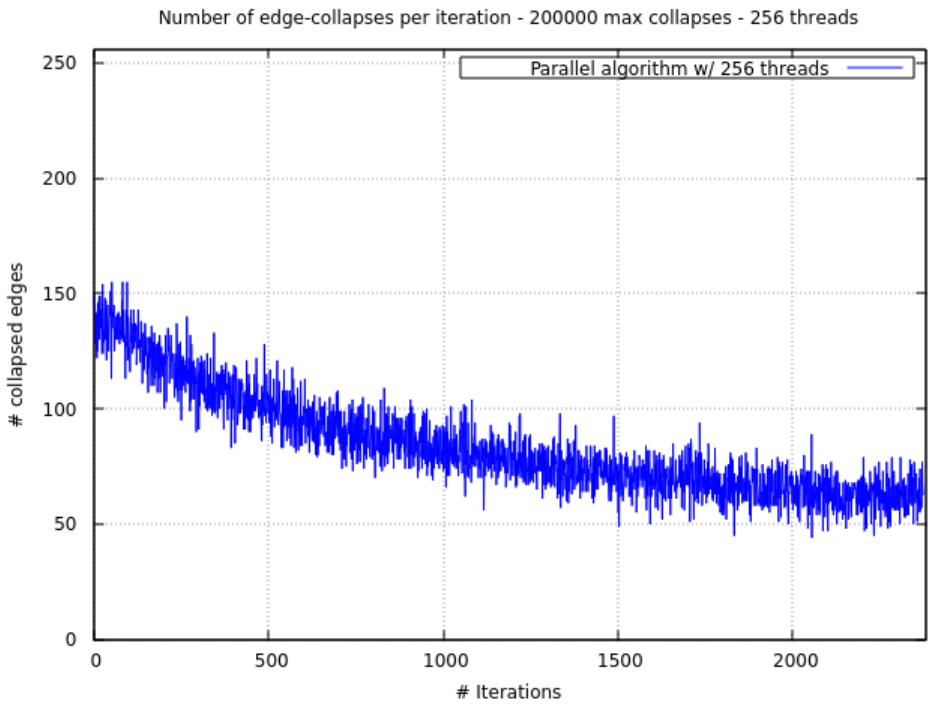


Figure 5.21: Number of edge-collapses per iteration of the Parallel algorithm with 256 threads. Target of 200,000 edge-collapses.

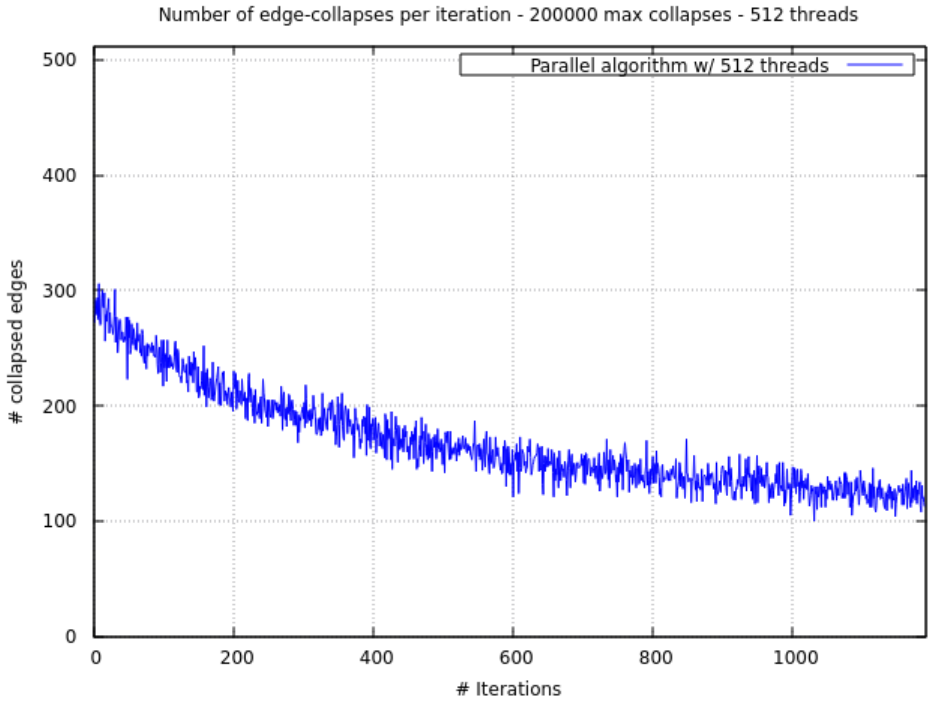


Figure 5.22: Number of edge-collapses per iteration of the Parallel algorithm with 512 threads. Target of 200,000 edge-collapses.

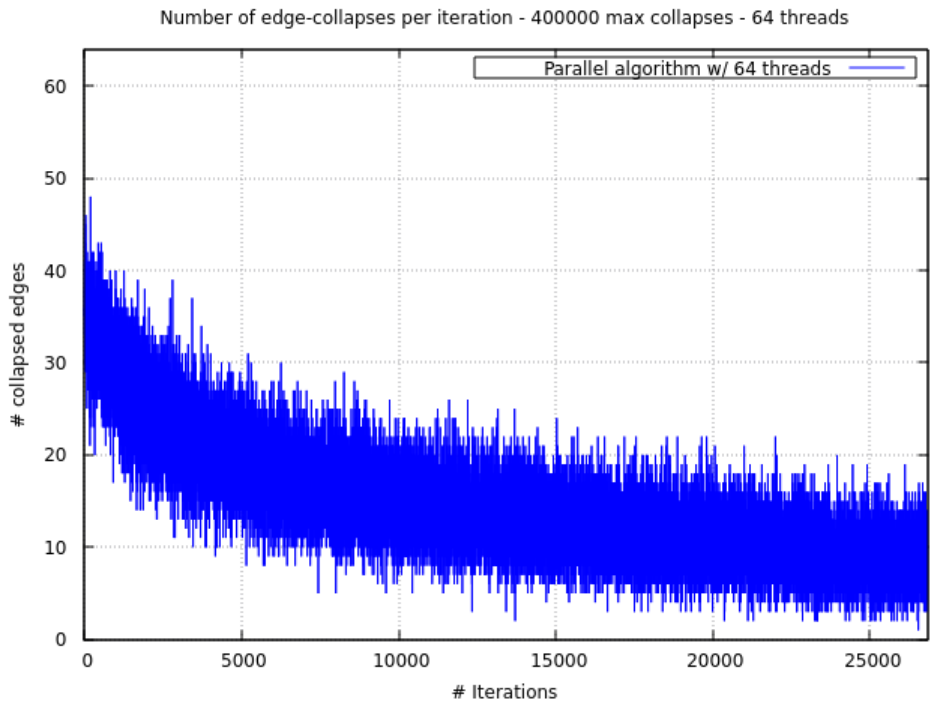


Figure 5.23: Number of edge-collapses per iteration of the Parallel algorithm with 64 threads. Target of 400,000 edge-collapses.

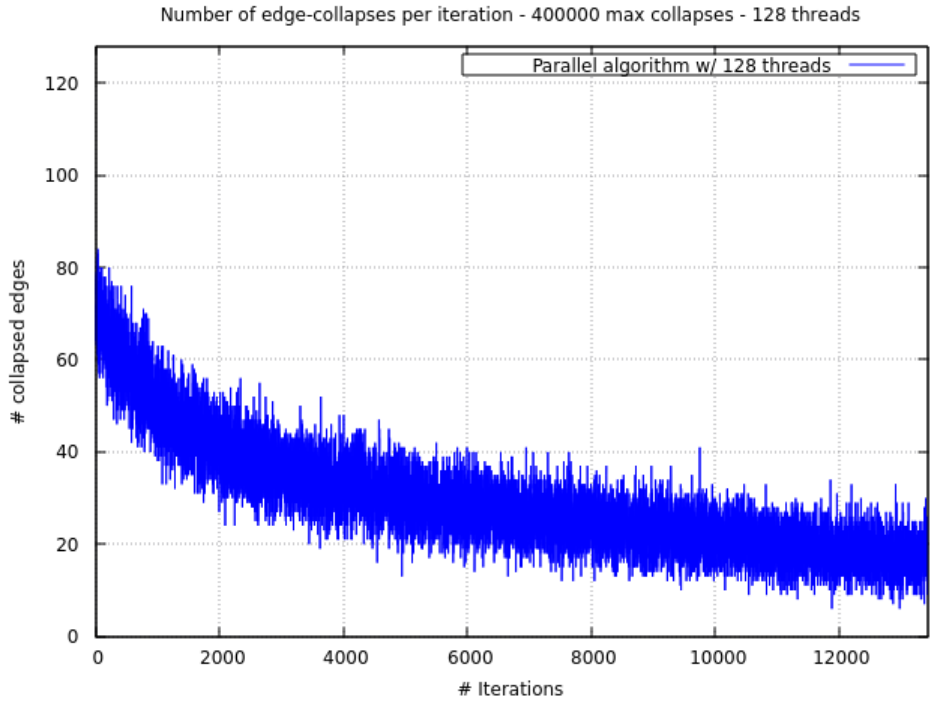


Figure 5.24: Number of edge-collapses per iteration of the Parallel algorithm with 128 threads. Target of 400,000 edge-collapses.

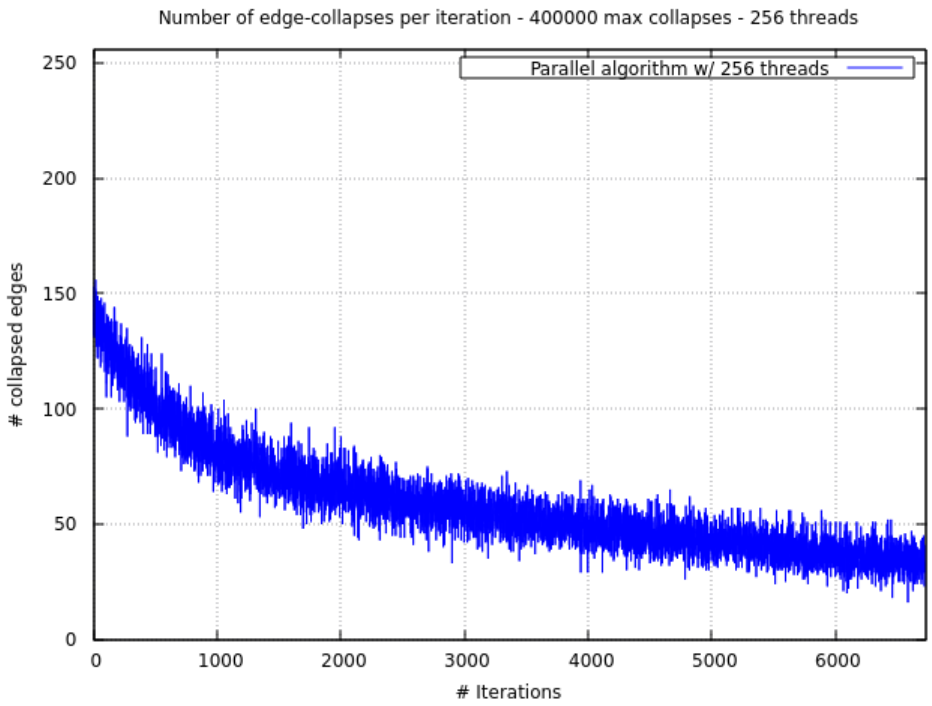


Figure 5.25: Number of edge-collapses per iteration of the Parallel algorithm with 256 threads. Target of 400,000 edge-collapses.

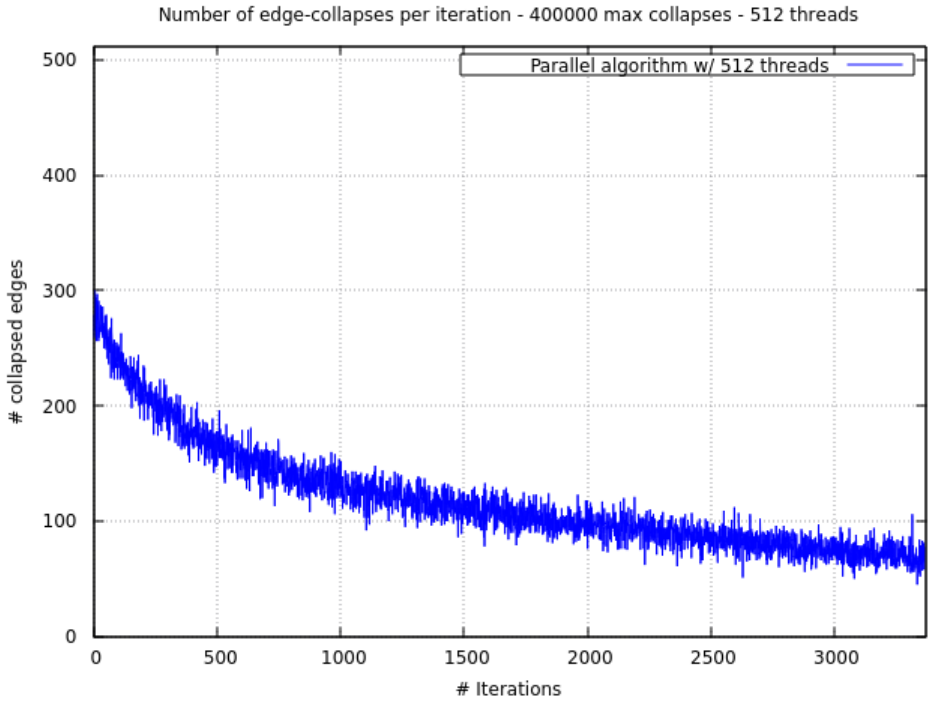


Figure 5.26: Number of edge-collapses per iteration of the Parallel algorithm with 512 threads. Target of 400,000 edge-collapses.

5.2 Qualitative Results

In this Section, we show the meshes obtained from the algorithm under the scenarios presented in the previous section. First, we show an overview of the mesh before and after the simplification procedure. Next, we show an overview of the edges marked by the metric to be candidates for a collapse under various thresholds. Next, we show an overview of the vertex density over the mesh, after consecutive edge-collapse targets are performed. Finally, we present a close-up of the meshes obtained under different edge-collapse target, for the central zone of the mesh, and the lower-left corner of the mesh.

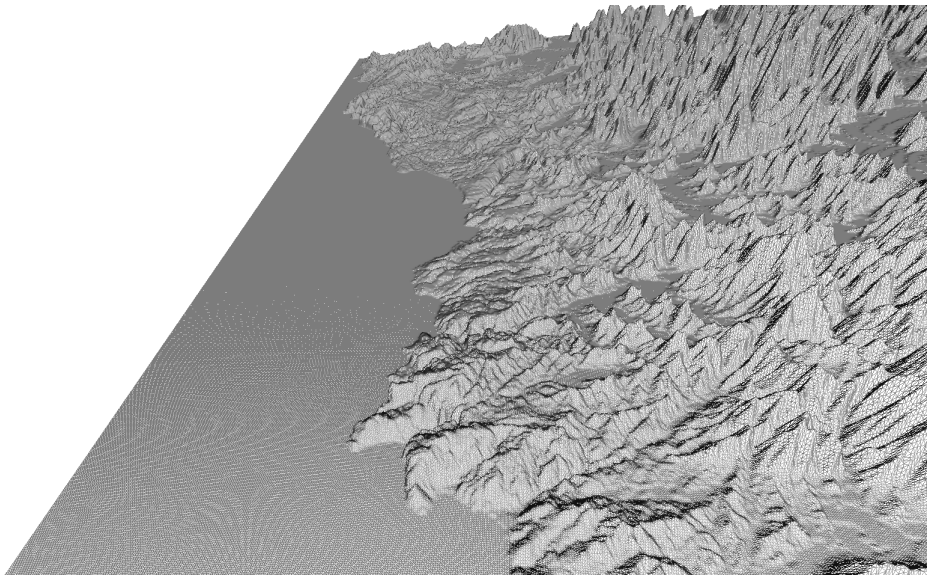


Figure 5.27: Overview of the original mesh. 998,001 vertices.

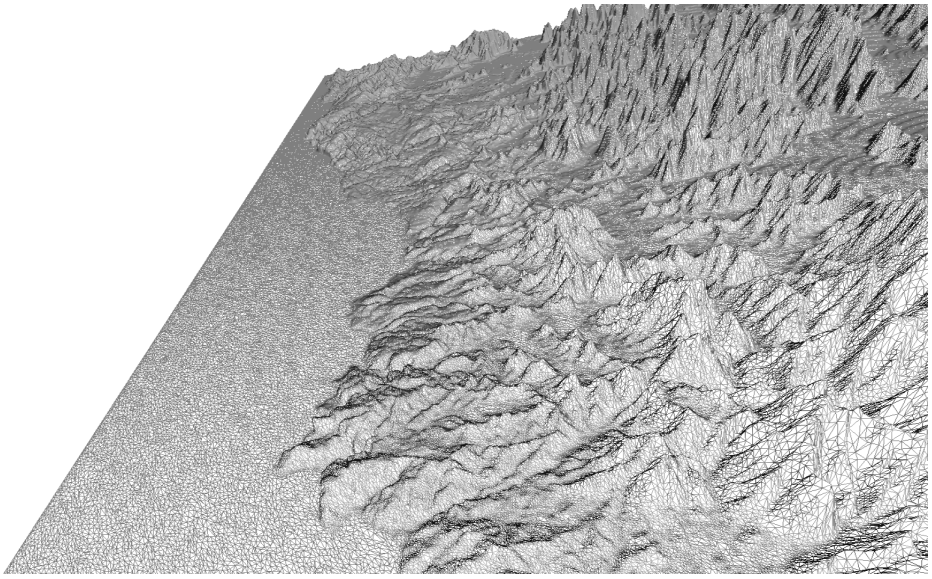


Figure 5.28: Overview of the mesh with 400,000 edge-collapses performed. 598,000 vertices.

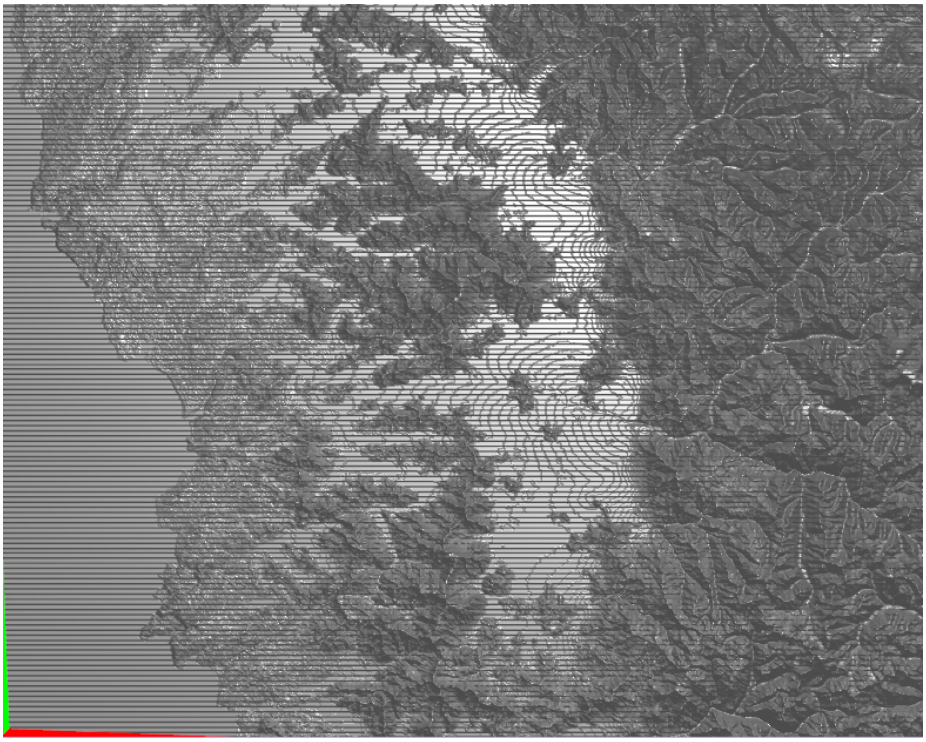


Figure 5.29: Overview of the original mesh.

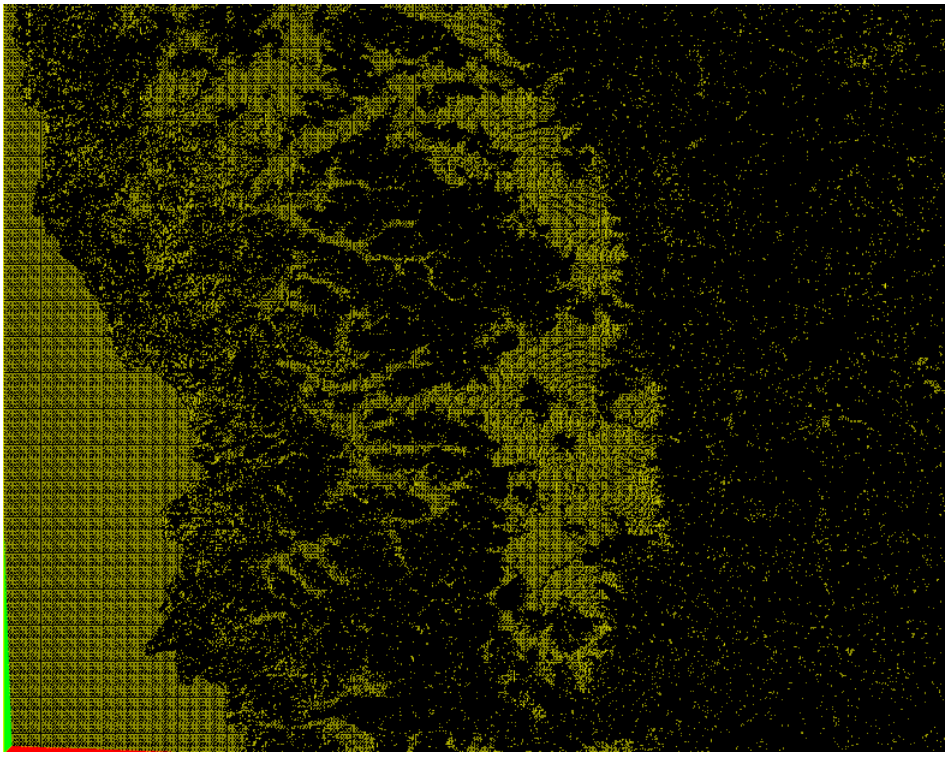


Figure 5.30: Overview of the candidate edges to be collapsed under the edge-length criterion with length < 0.5. Candidate edges marked in yellow.

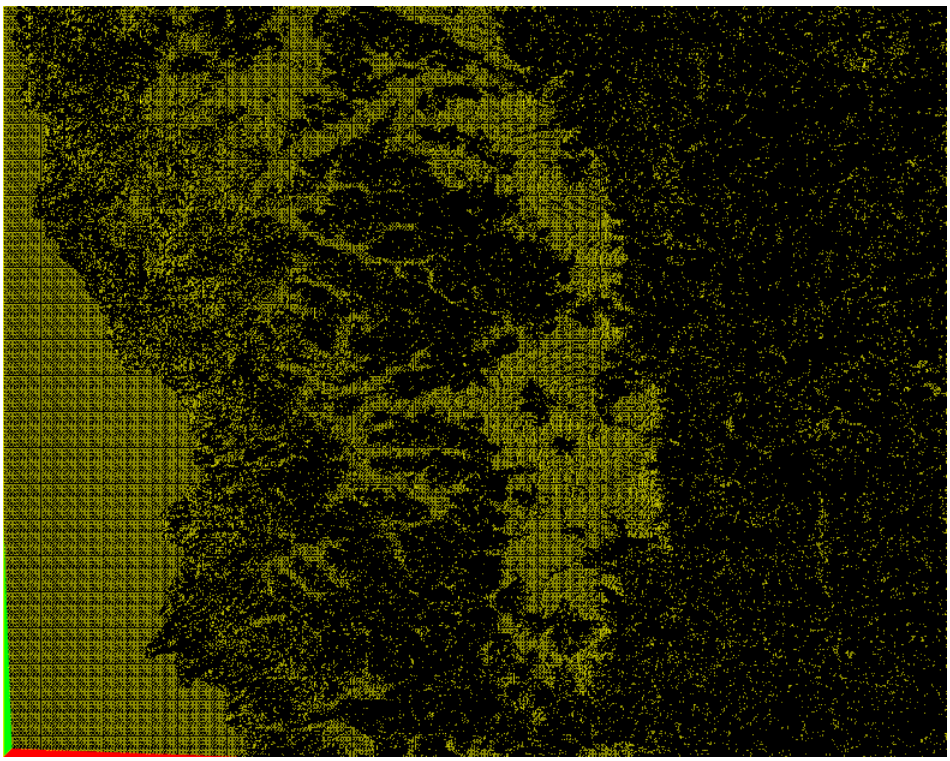


Figure 5.31: Overview of the candidate edges to be collapsed under the edge-length criterion with length < 0.6. Candidate edges marked in yellow.

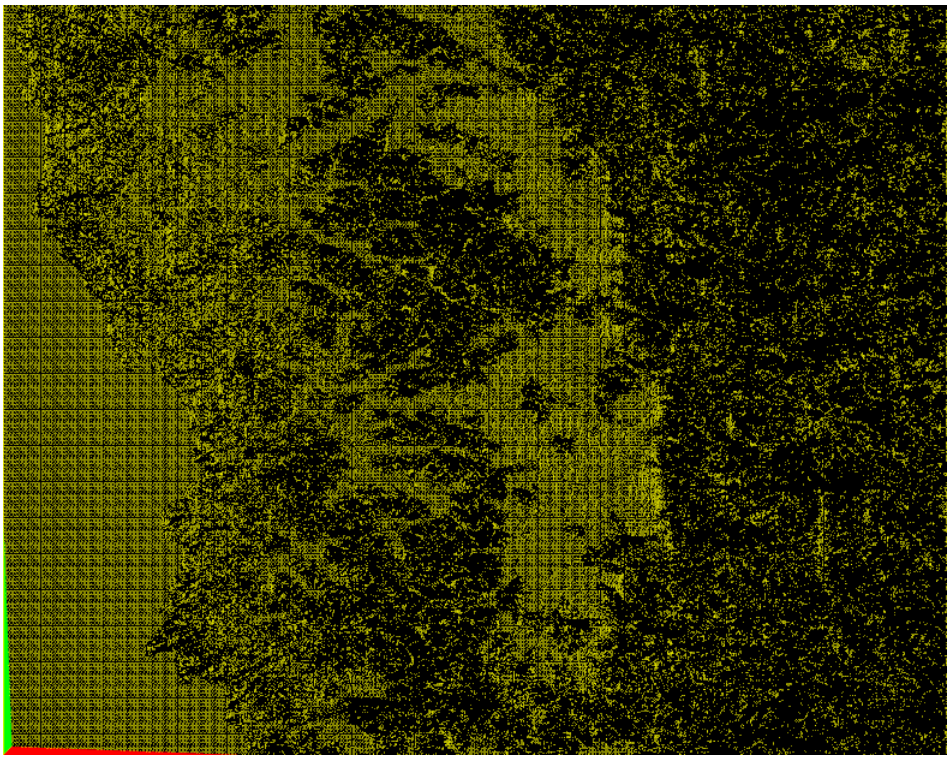


Figure 5.32: Overview of the candidate edges to be collapsed under the edge-length criterion with length < 0.8. Candidate edges marked in yellow.

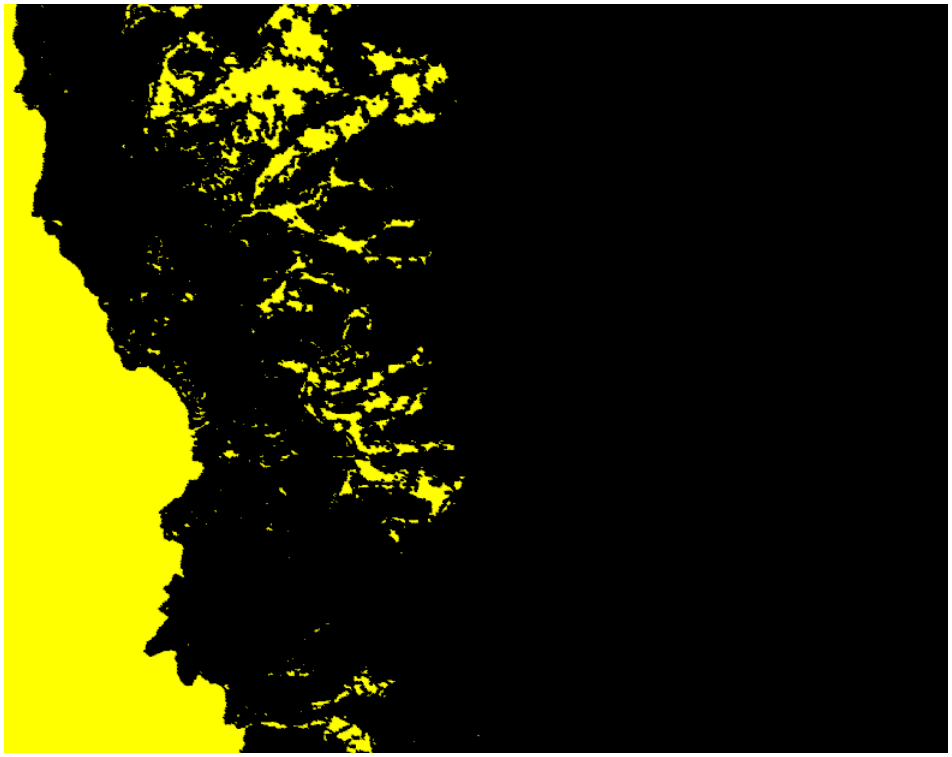


Figure 5.33: Overview of the candidate edges to be collapsed under the quadric metric criterion with threshold 0.2. Candidate edges marked in yellow.

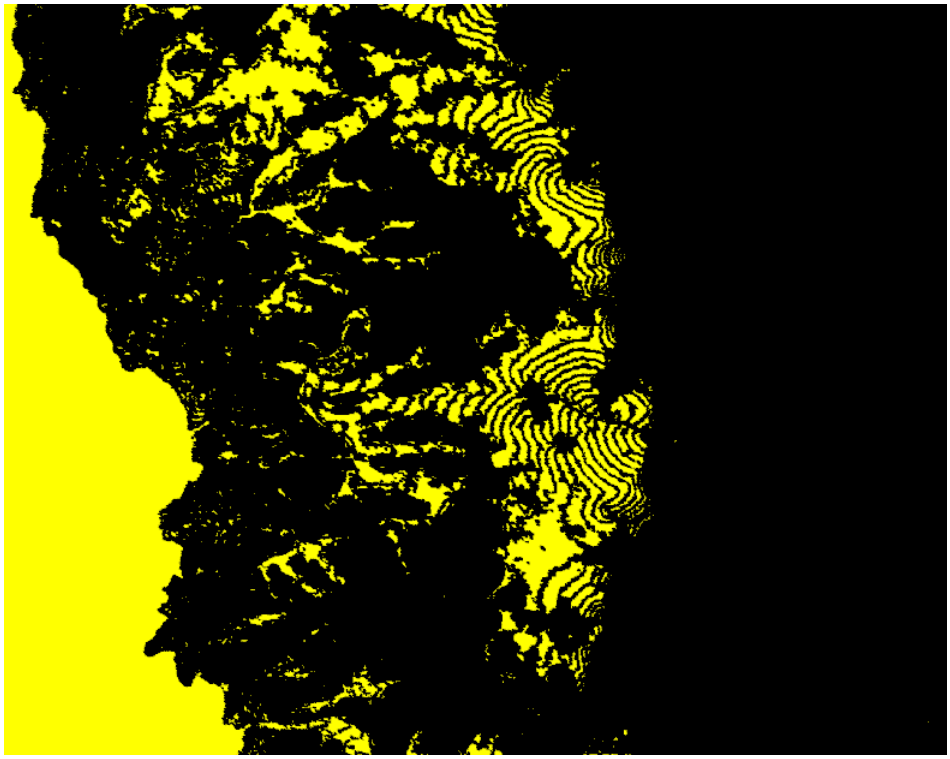


Figure 5.34: Overview of the candidate edges to be collapsed under the quadric metric criterion with threshold 0.4. Candidate edges marked in yellow.

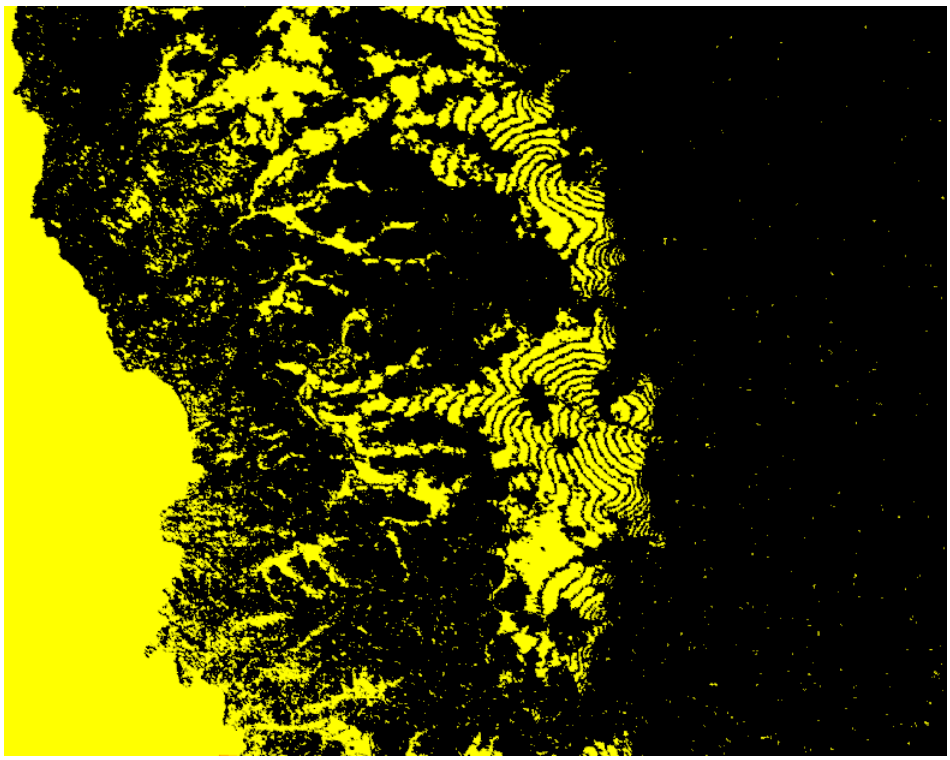


Figure 5.35: Overview of the candidate edges to be collapsed under the quadric metric criterion with threshold 0.6. Candidate edges marked in yellow.

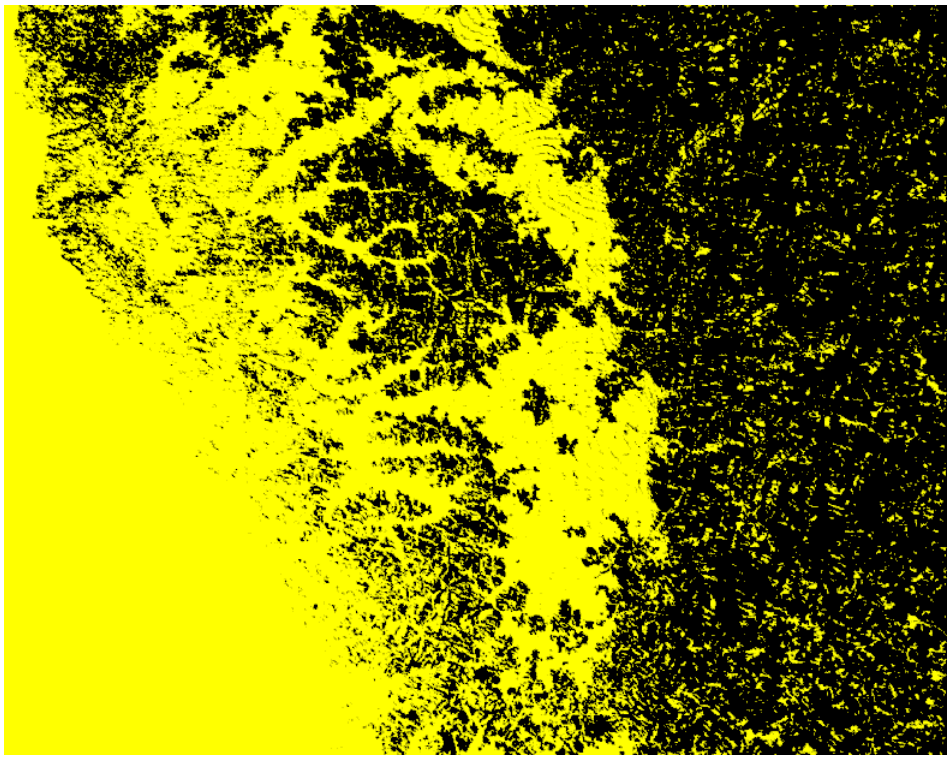


Figure 5.36: Overview of the candidate edges to be collapsed under the quadric metric criterion with threshold 0.8. Candidate edges marked in yellow.

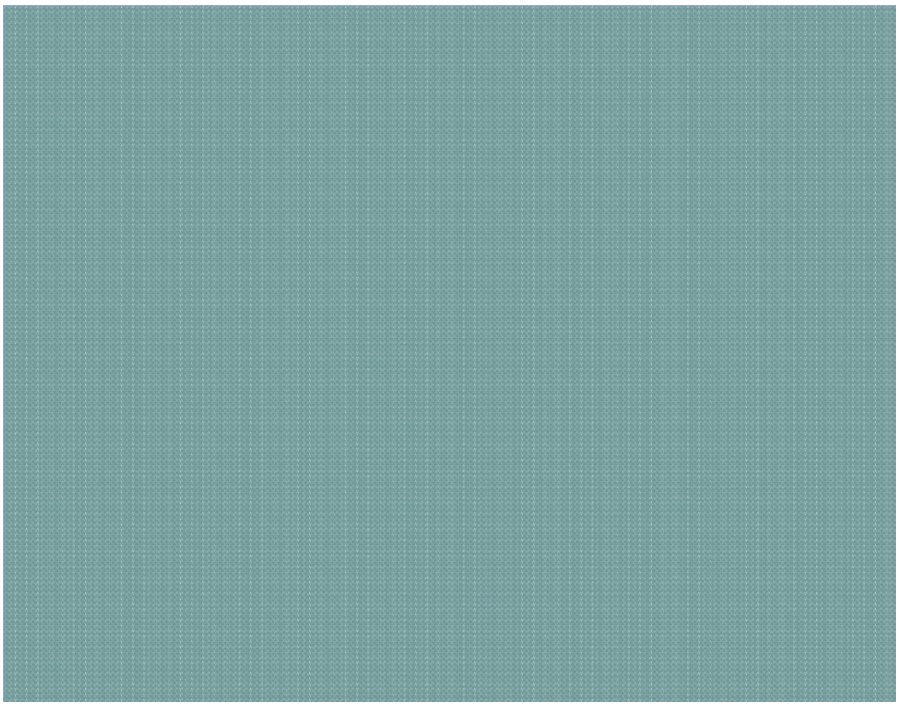


Figure 5.37: Overview of the vertex density over the mesh. Original mesh. 998,001 vertices.

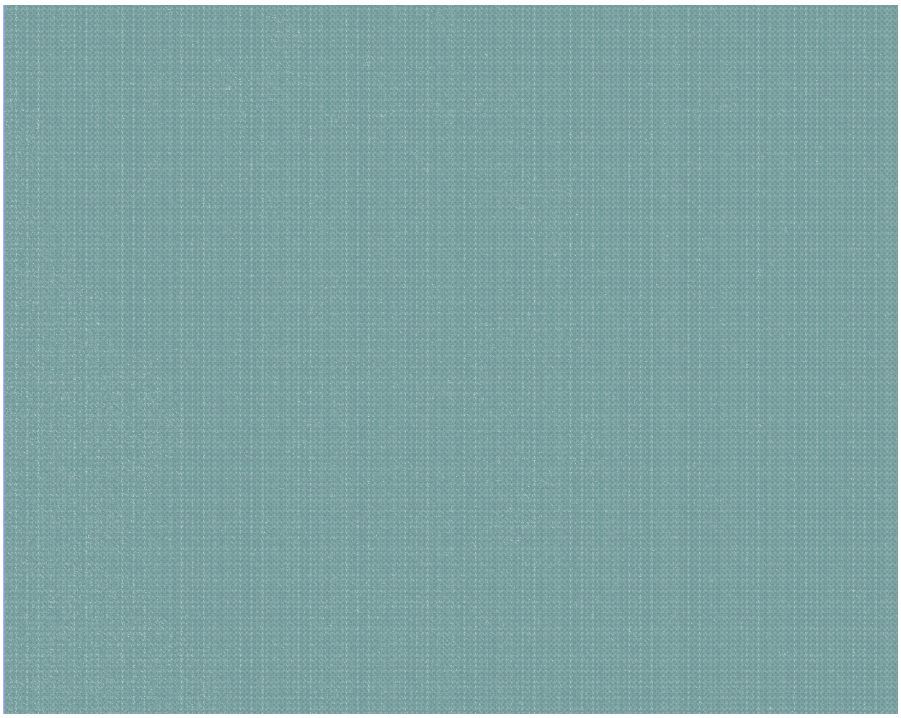


Figure 5.38: Overview of the vertex density over the mesh, 50,000 edge-collapses performed with the parallel algorithm with 512 threads, 947,805 vertices.

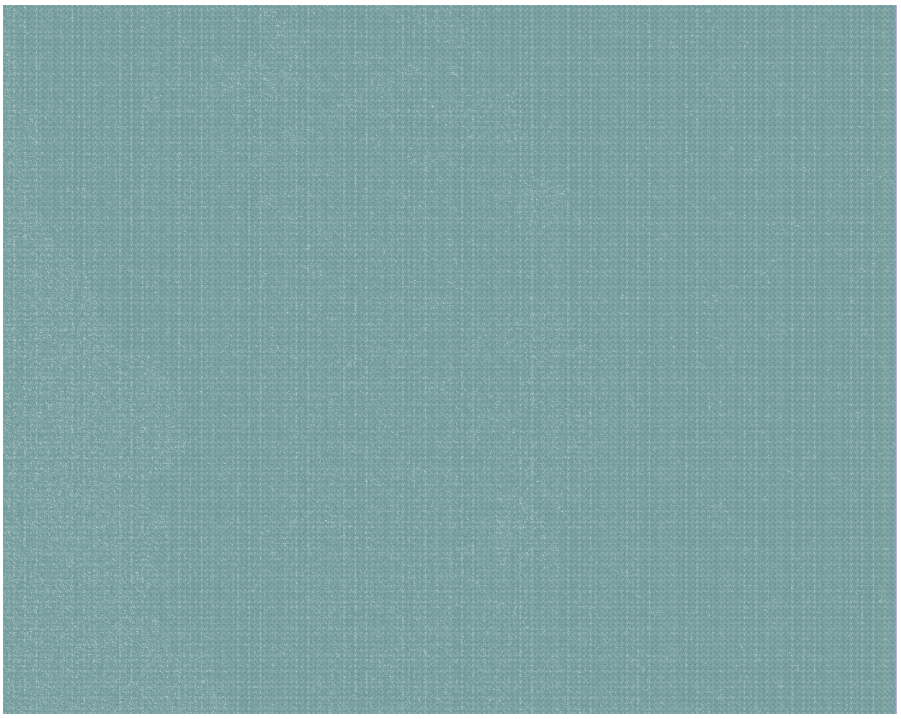


Figure 5.39: Overview of the vertex density over the mesh, 100,000 edge-collapses performed with the parallel algorithm with 512 threads, 897,879 vertices.

Figure 5.40: Overview of the vertex density over the mesh, 200,000 edge-collapses performed with the parallel algorithm with 512 threads, 797,935 vertices.

Figure 5.41: Overview of the vertex density over the mesh, 300,000 edge-collapses performed with the parallel algorithm with 512 threads, 697,961 vertices.

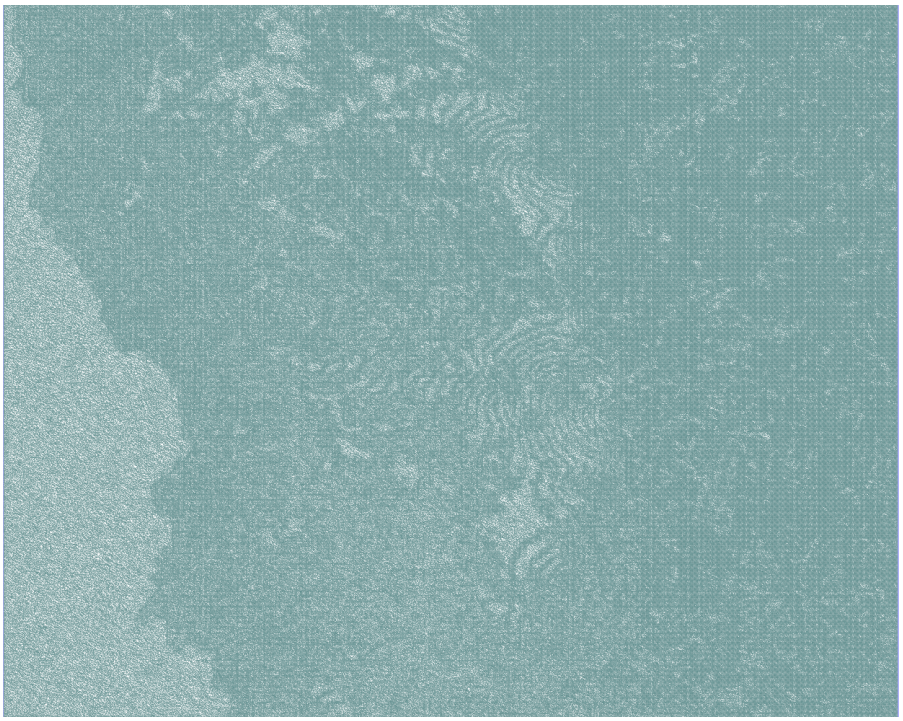


Figure 5.42: Overview of the vertex density over the mesh, 400,000 edge-collapses performed with the parallel algorithm with 512 threads, 597,999 vertices.

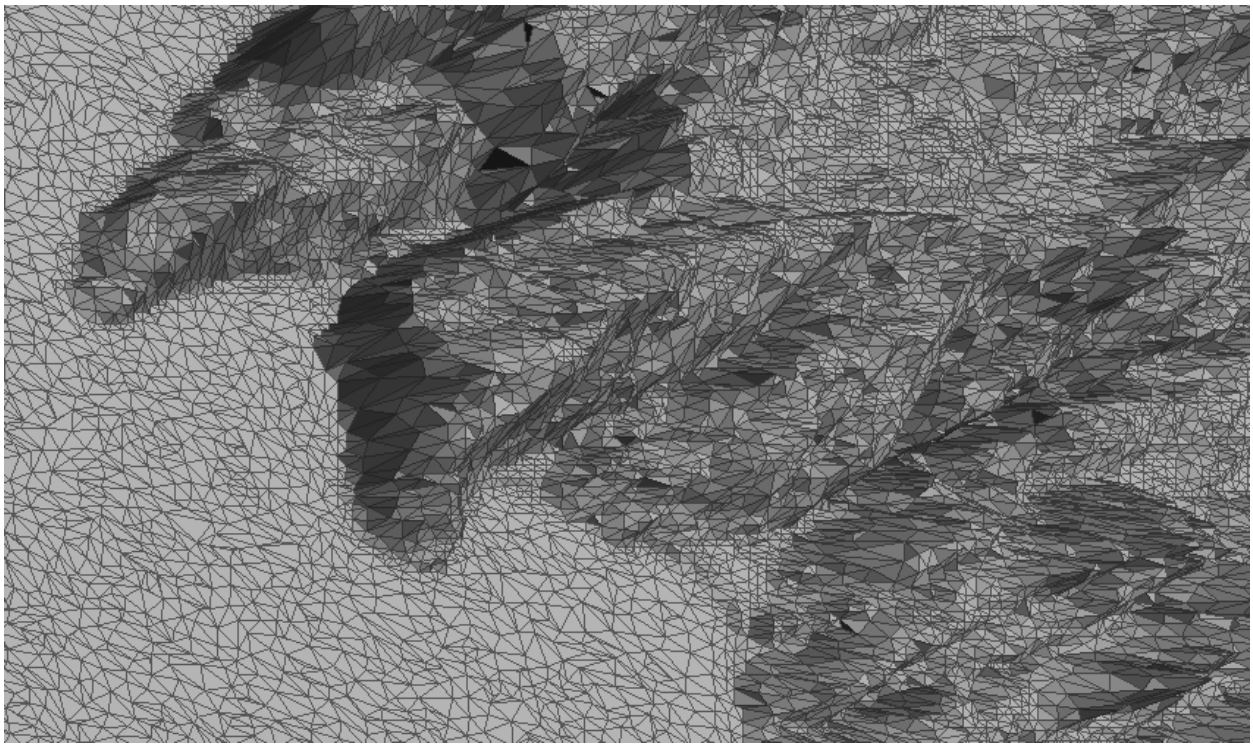


Figure 5.43: Close-up of the lower-left zone of the mesh, 400,000 edge-collapses performed with the sequential algorithm, 598,000 vertices.

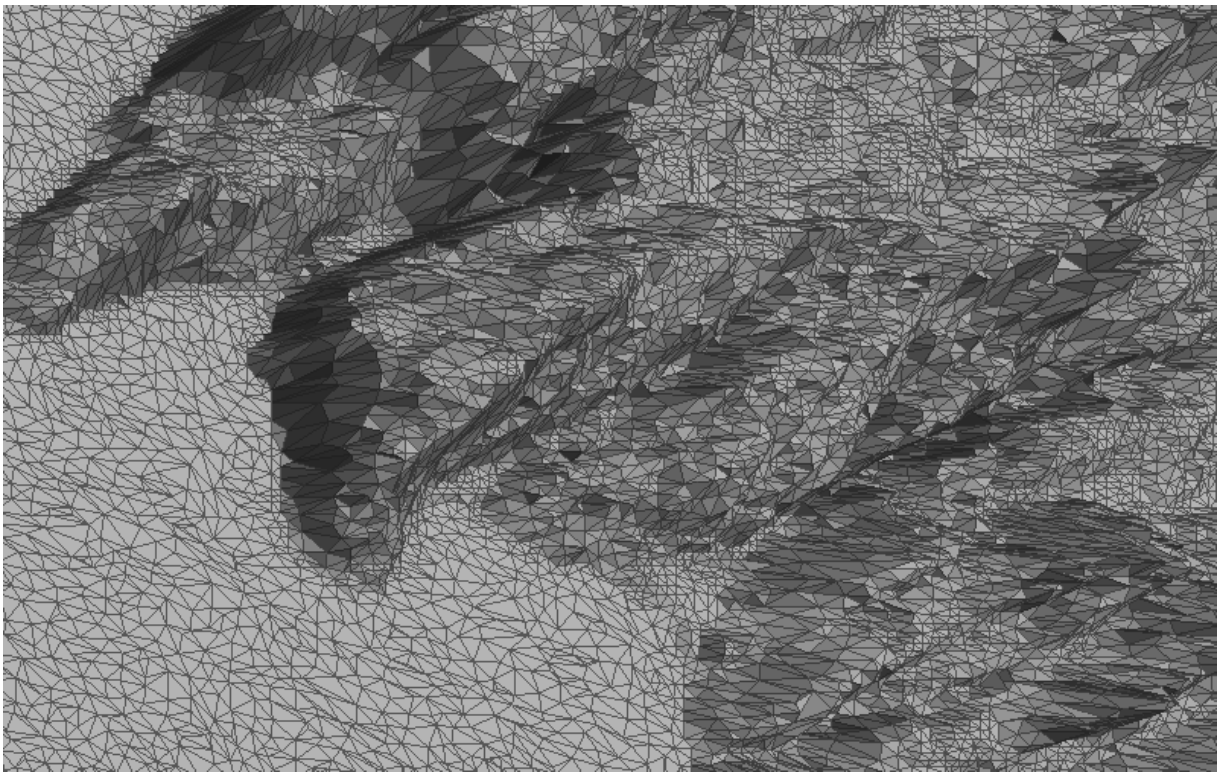


Figure 5.44: Close-up of the lower-left zone of the mesh, 400,001 edge-collapses performed with the parallel algorithm with 128 threads, 597,998 vertices.

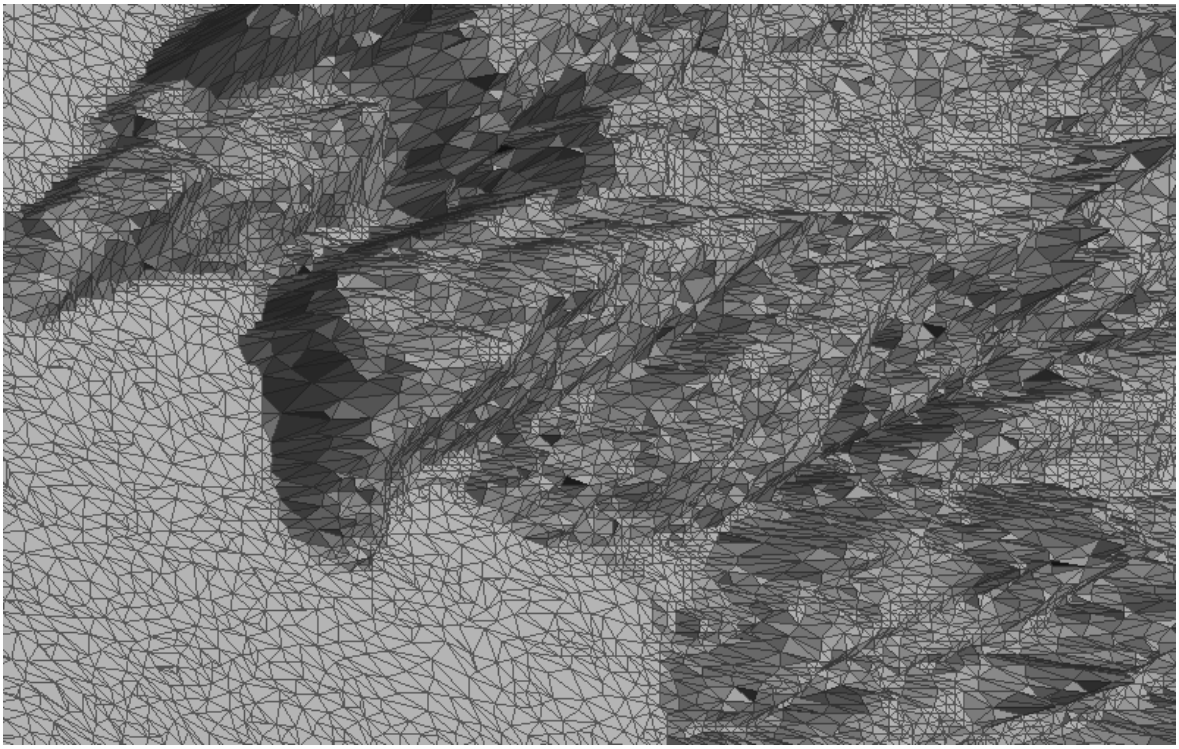


Figure 5.45: Close-up of the lower-left zone of the mesh, 400,002 edge-collapses performed with the parallel algorithm with 512 threads, 597,997 vertices.

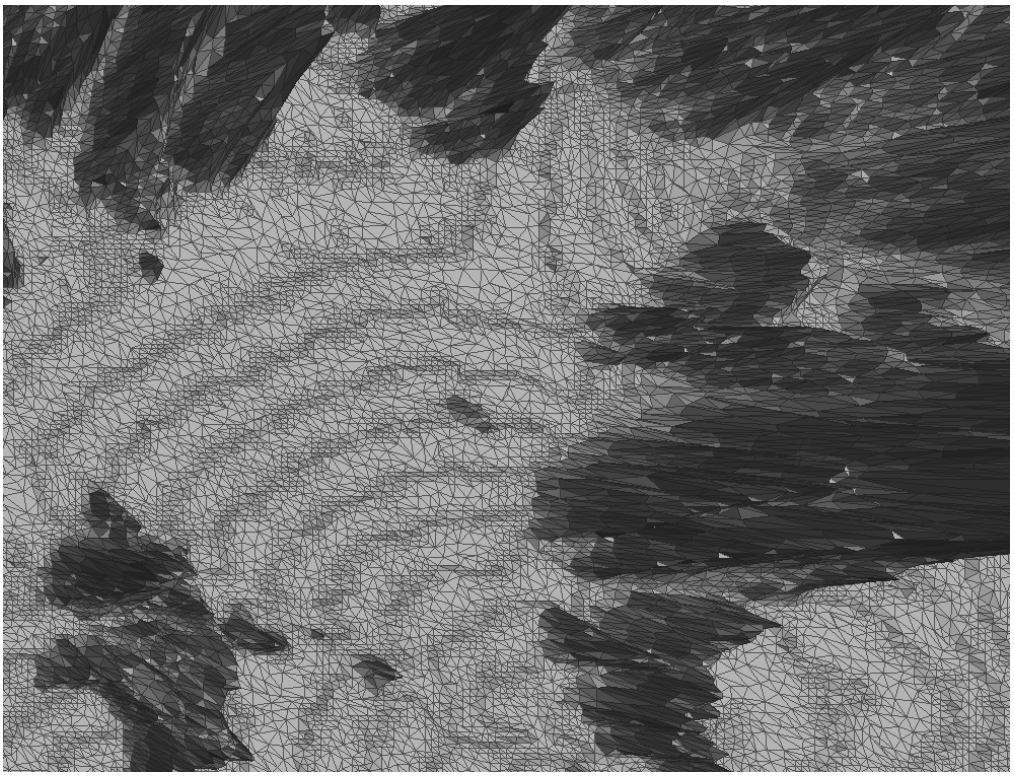


Figure 5.46: Close-up of the center zone of the mesh, 400,000 edge-collapses performed with the sequential algorithm, 598,000 vertices.

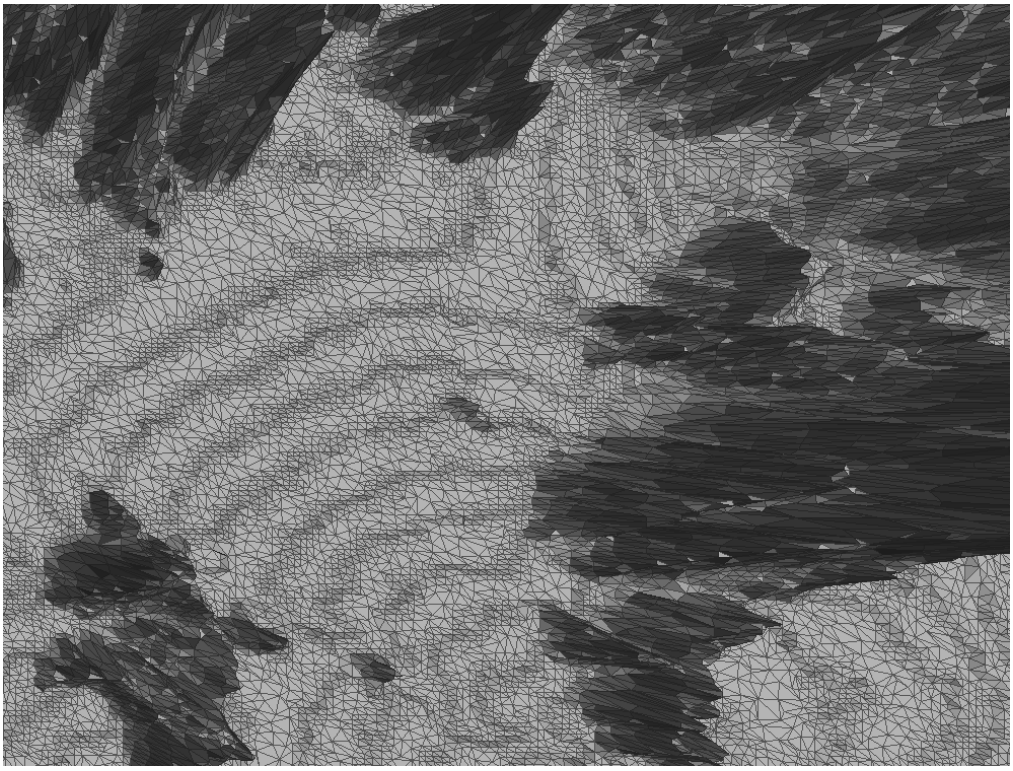


Figure 5.47: Close-up of the center zone of the mesh, 400,001 edge-collapses performed with the parallel algorithm with 128 threads, 597,998 vertices.

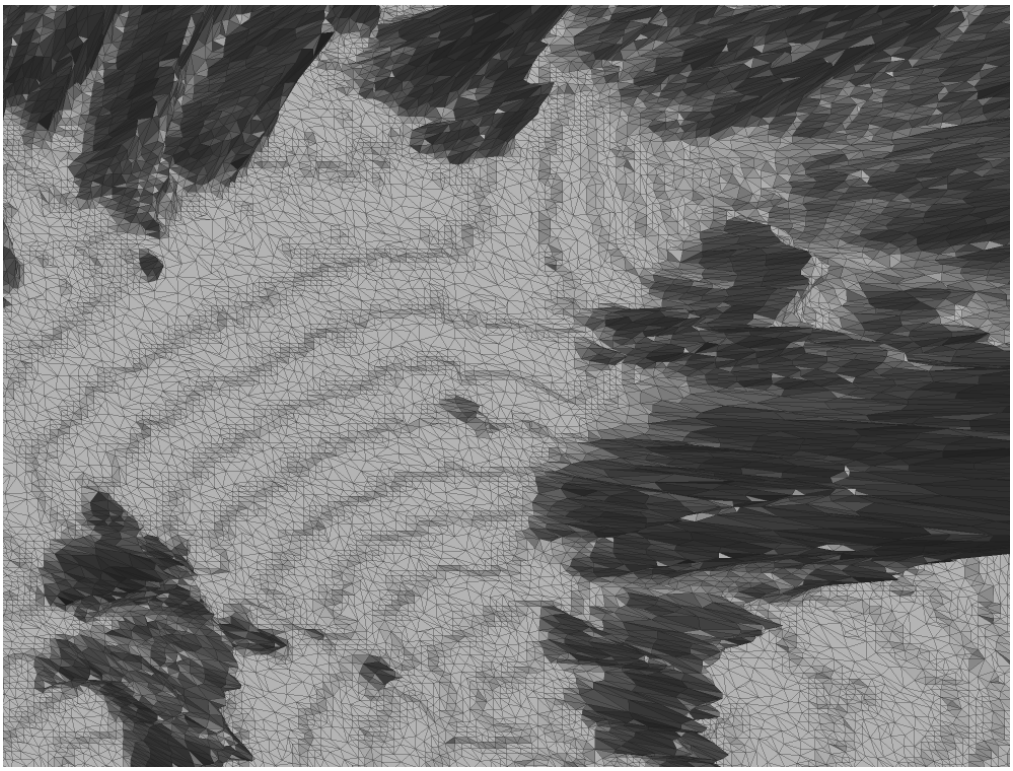


Figure 5.48: Close-up of the center zone of the mesh, 400,002 edge-collapses performed with the parallel algorithm with 512 threads, 597,997 vertices.

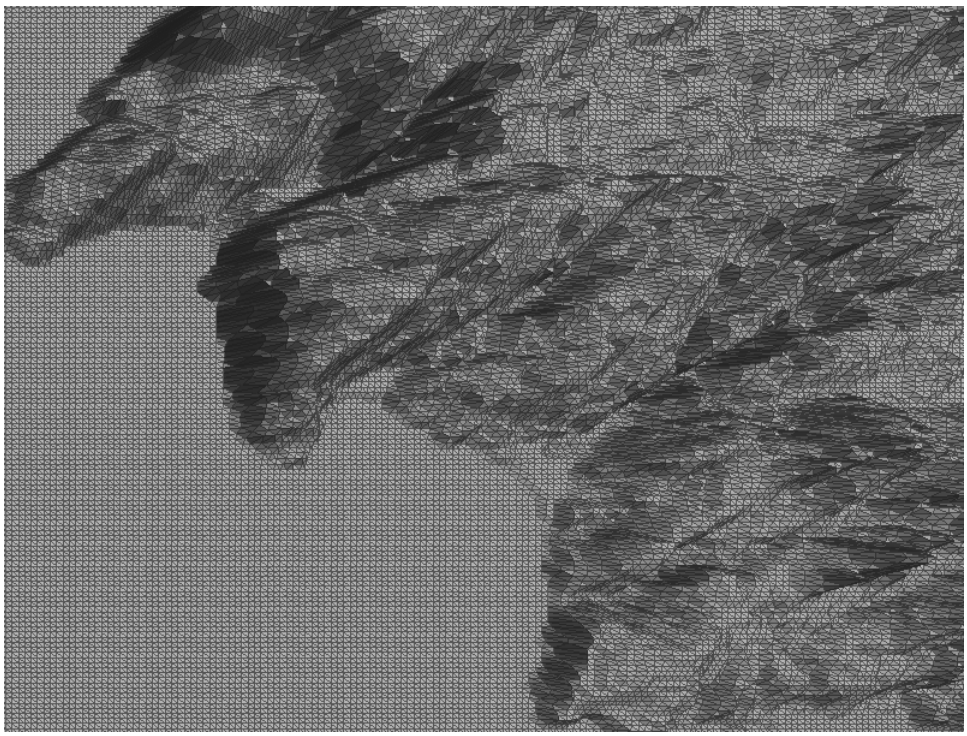


Figure 5.49: Close-up of the lower-left zone of the mesh. Original mesh. 998,001 vertices.

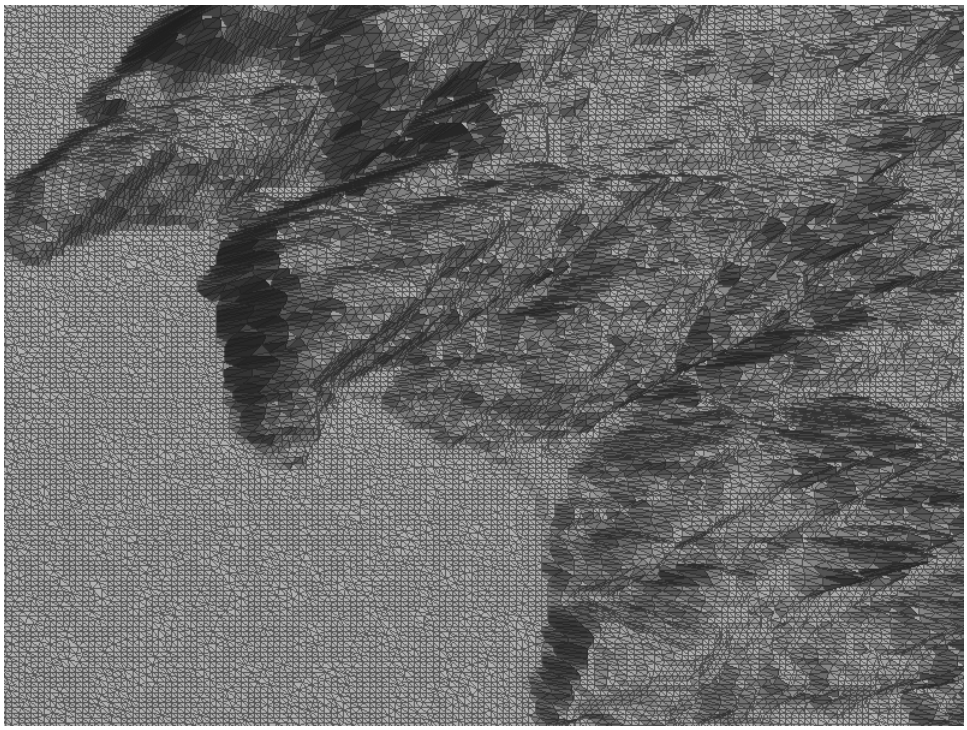


Figure 5.50: Close-up of the lower-left zone of the mesh, 50,000 edge-collapses performed with the parallel algorithm with 512 threads, 947,805 vertices.

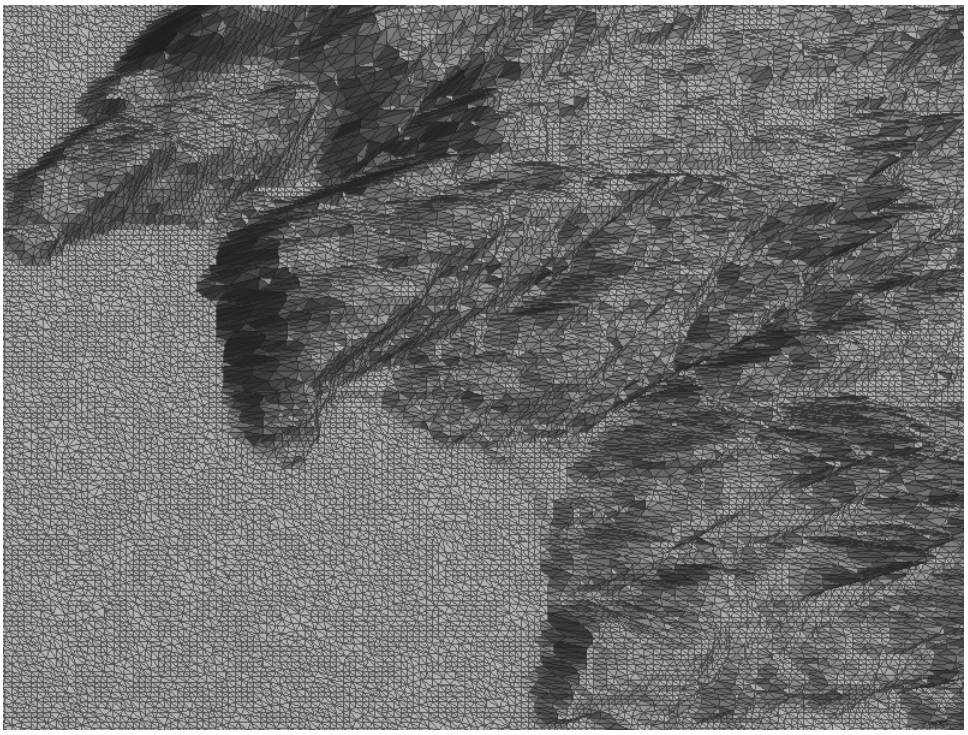


Figure 5.51: Close-up of the lower-left zone of the mesh, 100,000 edge-collapses performed with the parallel algorithm with 512 threads, 897,879 vertices.

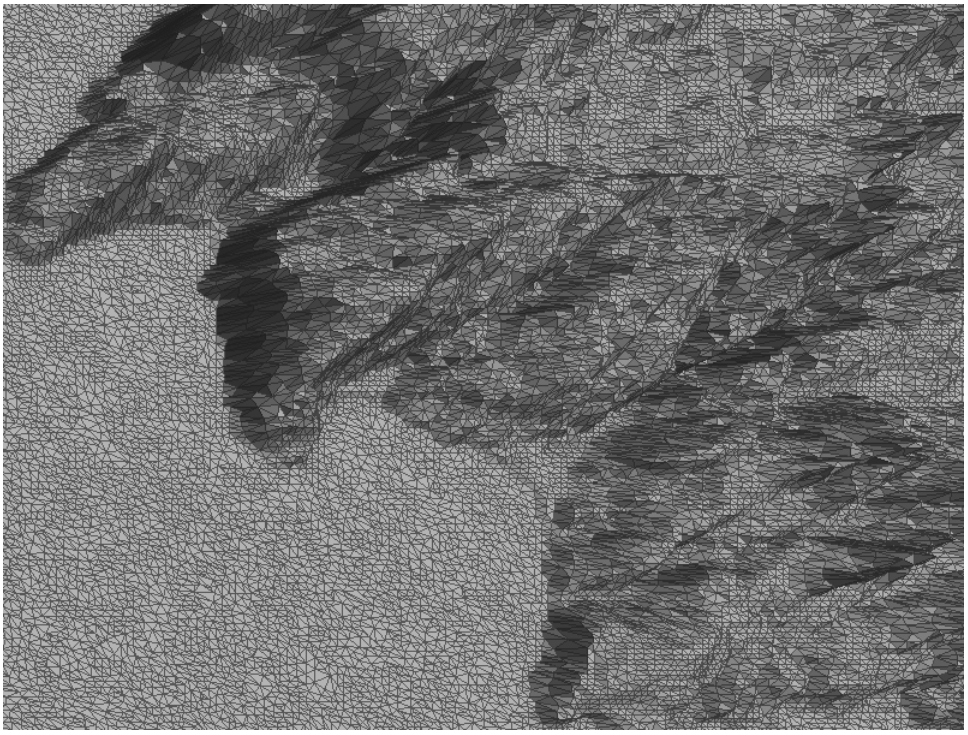


Figure 5.52: Close-up of the lower-left zone of the mesh, 200,000 edge-collapses performed with the parallel algorithm with 512 threads, 797,935 vertices.

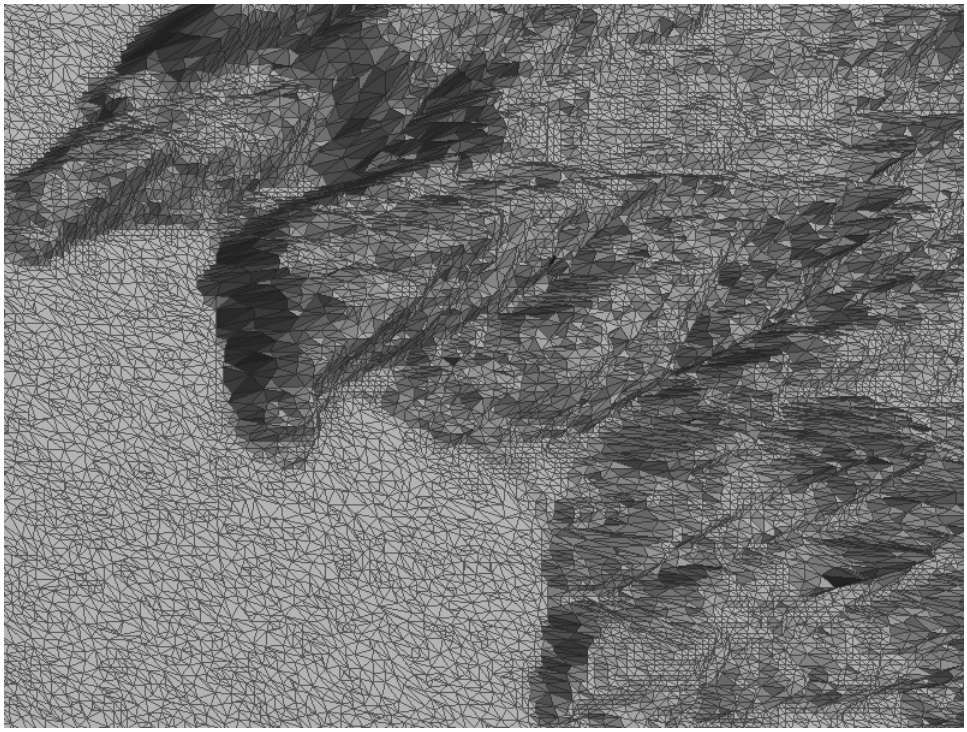


Figure 5.53: Close-up of the lower-left zone of the mesh, 300,000 edge-collapses performed with the parallel algorithm with 512 threads, 697,961 vertices.

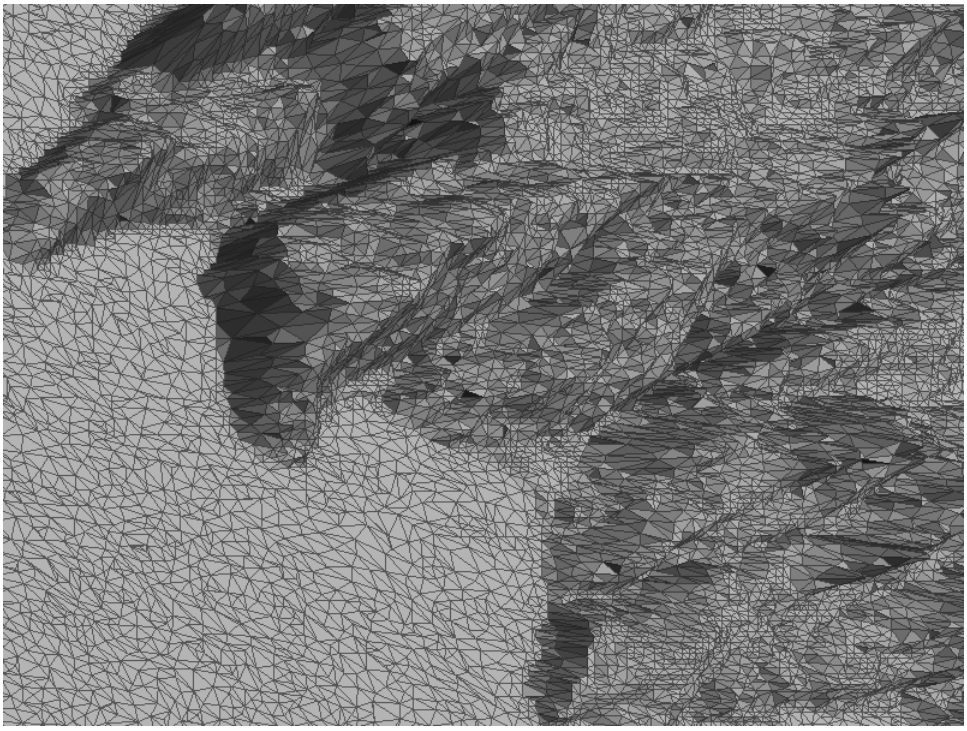


Figure 5.54: Close-up of the lower-left zone of the mesh, 400,000 edge-collapses performed with the parallel algorithm with 512 threads, 597,999 vertices.

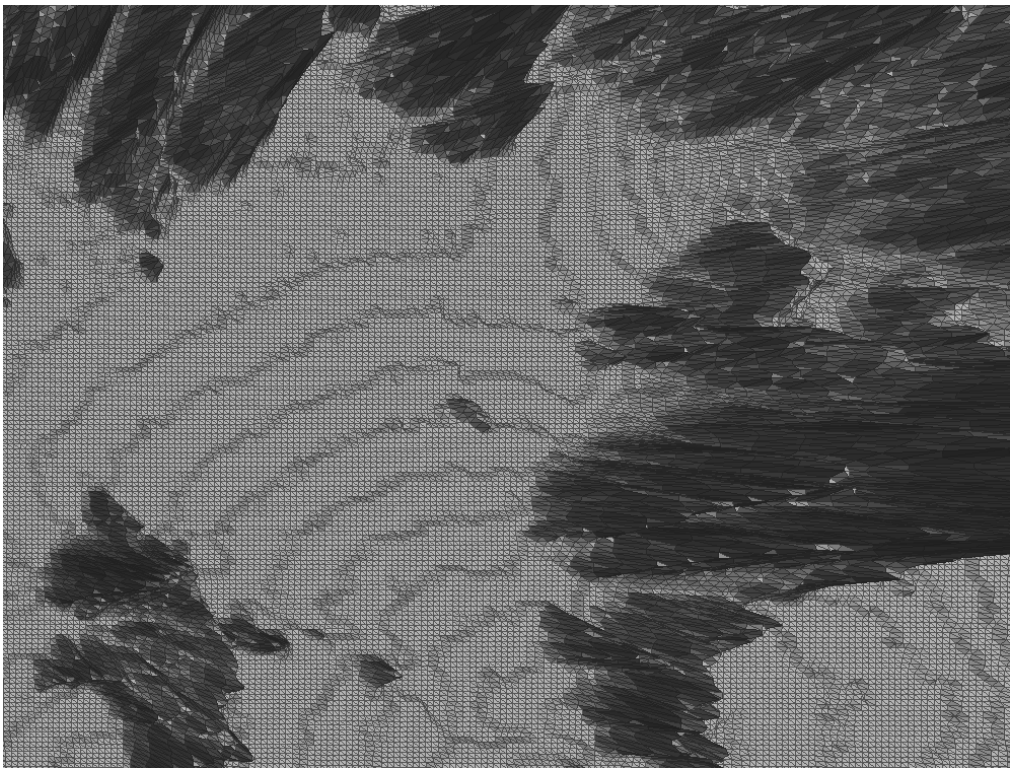


Figure 5.55: Close-up of the center zone of the mesh. Original mesh. 998,001 vertices.

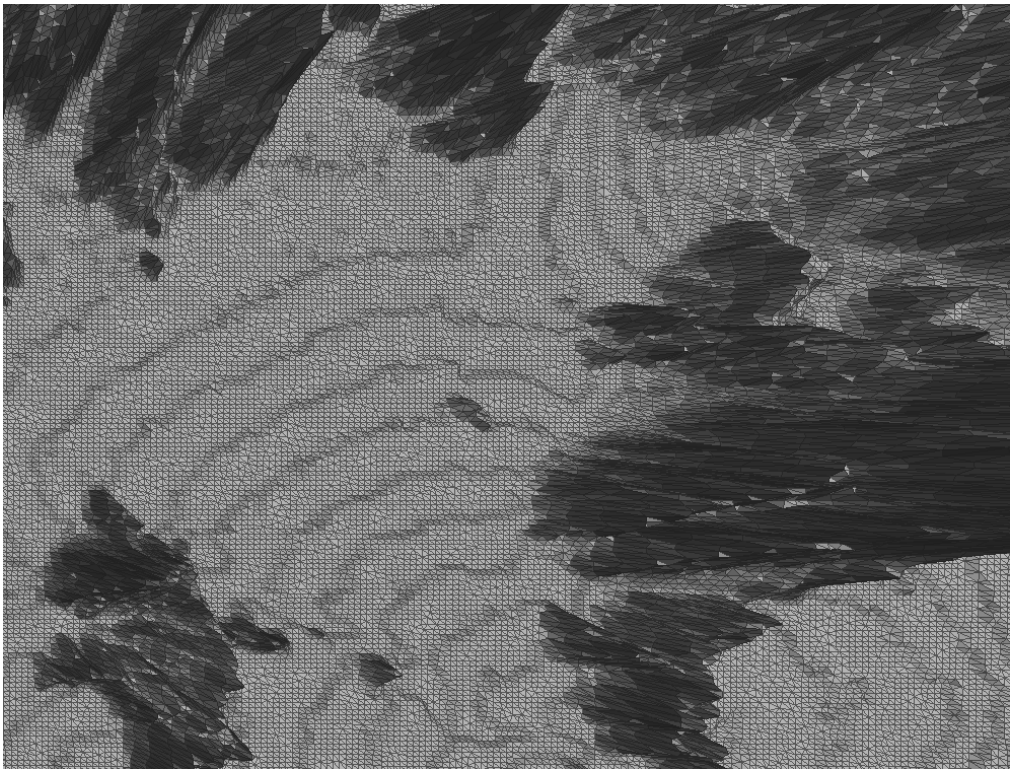


Figure 5.56: Close-up of the center zone of the mesh, 50,000 edge-collapses performed with the parallel algorithm with 512 threads, 947,805 vertices.

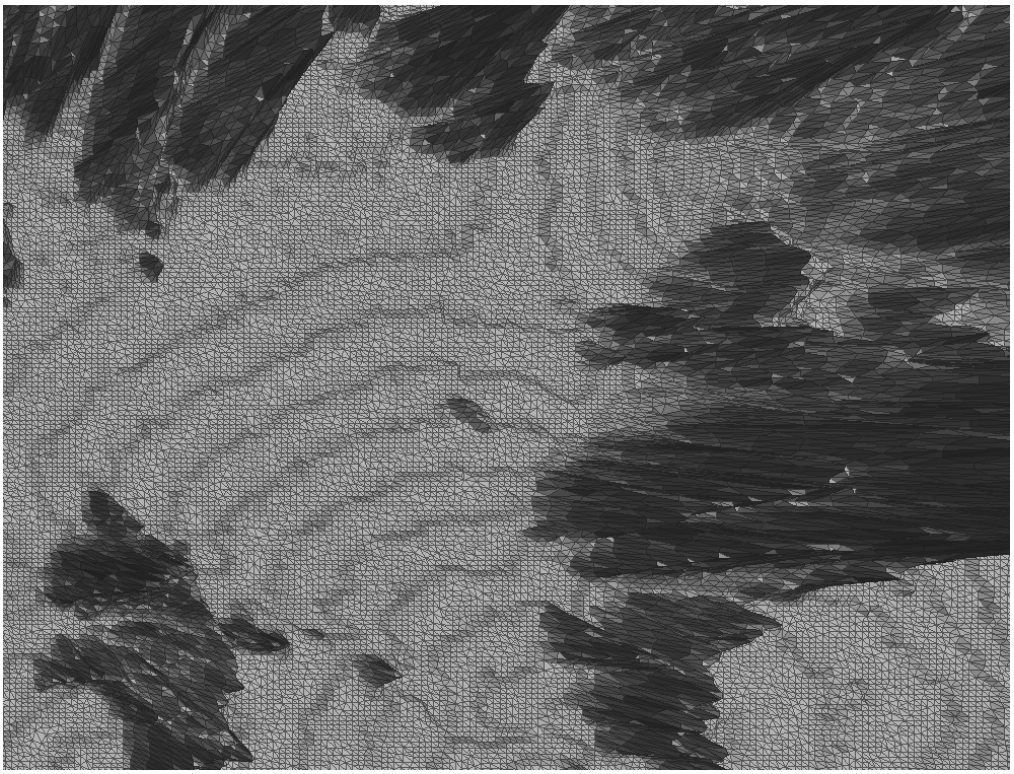


Figure 5.57: Close-up of the center zone of the mesh, 100,000 edge-collapses performed with the parallel algorithm with 512 threads, 897,879 vertices.

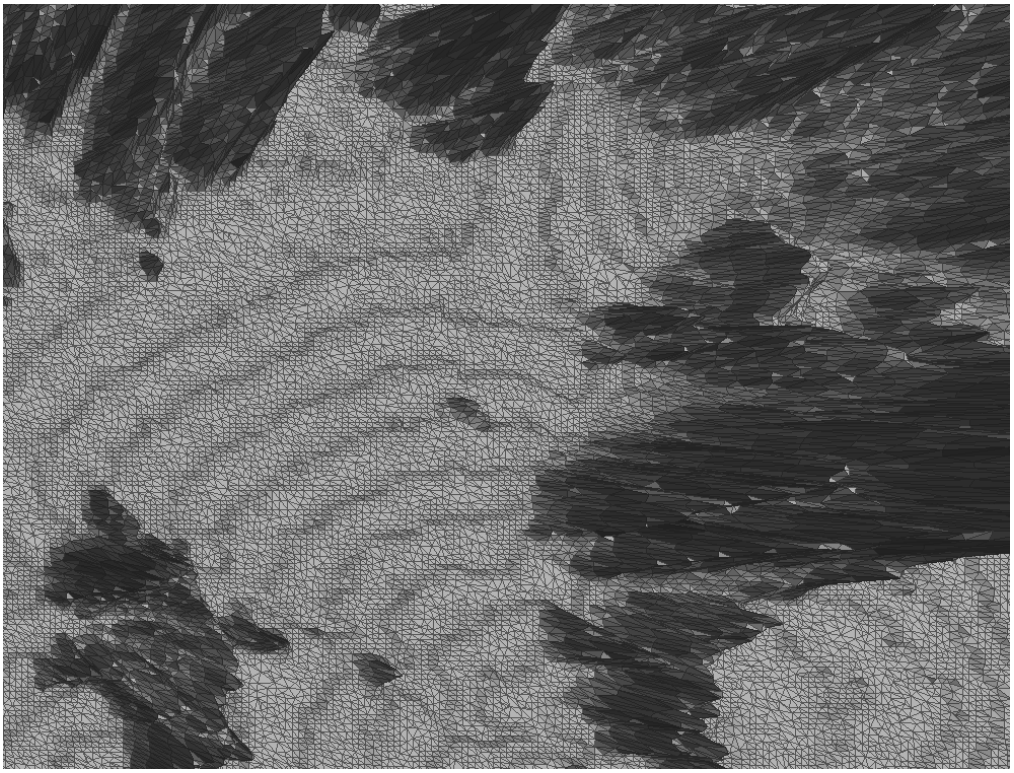


Figure 5.58: Close-up of the center zone of the mesh, 200,000 edge-collapses performed with the parallel algorithm with 512 threads, 797,935 vertices.

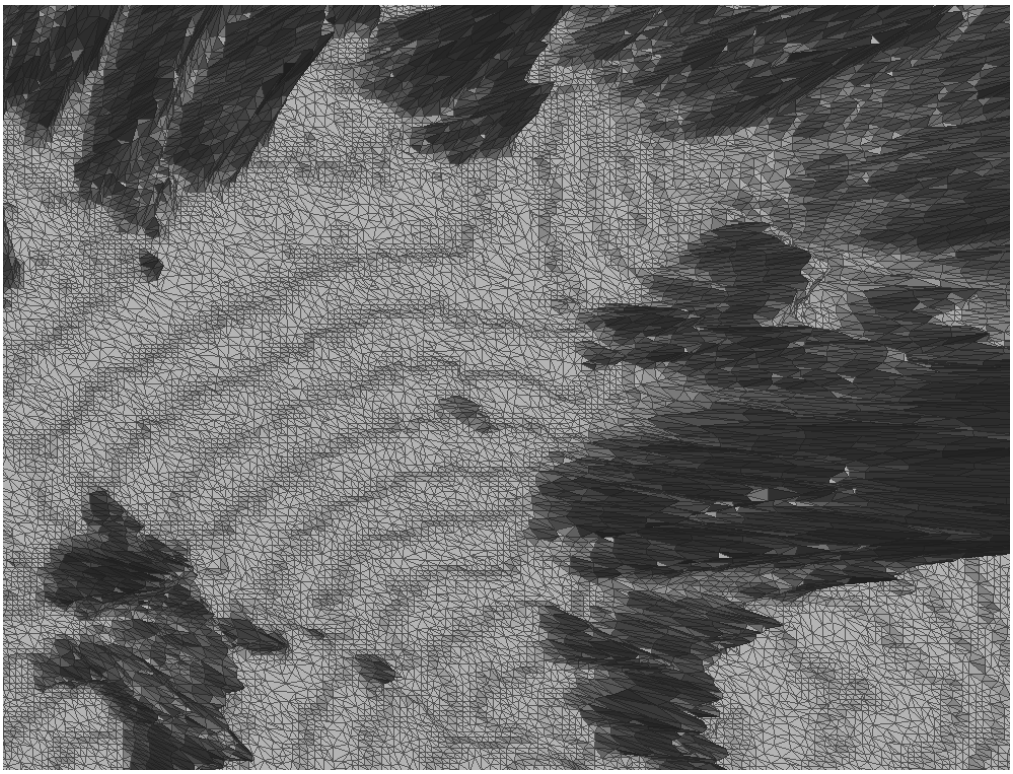


Figure 5.59: Close-up of the center zone of the mesh, 300,000 edge-collapses performed with the parallel algorithm with 512 threads, 697,961 vertices.

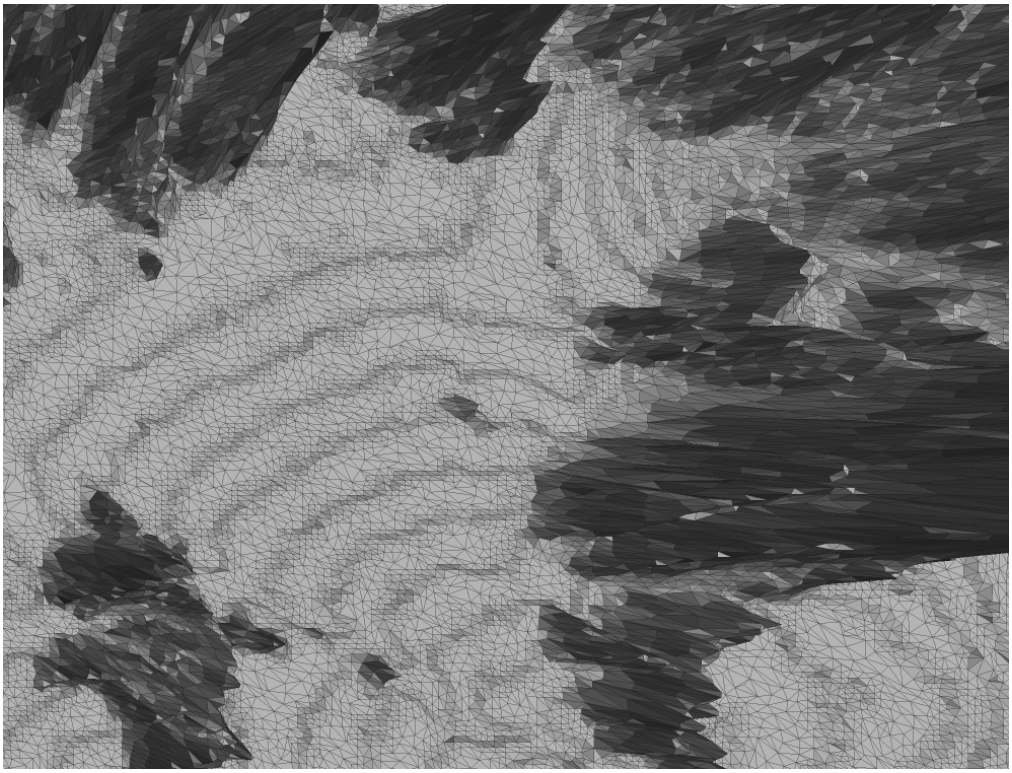


Figure 5.60: Close-up of the center zone of the mesh, 400,000 edge-collapses performed with the parallel algorithm with 512 threads, 597,999 vertices.

5.3 Analysis

As can be seen in Table 5.8 and Figure 5.3, the speedup of the parallel algorithm (using 512 threads) versus the ad-hoc sequential implementation reaches up to three orders of magnitude, being faster when performing a target of 50,000 edge-collapses, and slightly smaller when doing 400,000 edge-collapses. This speedup only considers the time spent on the edge-collapse portion of the algorithm. If we take in account the time spent on the edge-collapse steps and the final vertices rearrange step as a whole, then we obtain an speedup between one and two orders of magnitude, as can be seen in Table 5.9 and Figure 5.7. The other parallel configurations who use fewer threads that 512 show a time increase proportional to the fewer number of threads utilized.

We observe from Figure 5.11 to 5.26, that the number of threads that perform a successful edge-collapse decays as the number of iterations increases, starting from an approximate of 50% of successful edge-collapses per iteration (relative to the total number of threads), and ending with an approximate of $20\% - 33\%$ successful edge-collapse. This is expected, because as the number of iteration increases, the state of the mesh changes, so it is more likely to encounter more edges that do not fulfill the criteria to be collapsed, failing the tests explained on Section 4.5, especially the Neighbors test, because a successful edge-collapse increases the number of edges connected to a single vertex.

From Table 5.7 we observe that the time needed for the vertices rearrange step increases as the number of edge-collapses performed increases, both in the sequential and parallel case. This is expected, because as more edge-collapses are performed, more gaps are generated in the data structures, thus increasing the number of *vertex-swaps* needed to fill these gaps and generate the output file.

Additionally, from Figures 5.4, 5.6 and 5.8, we observe that the vertices rearrange step impacts negatively on the overall run-time of the algorithm, taking almost three times the time spent on the edge-collapse algorithm phase. This tend us to believe that this final step may not be implemented optimally, and we should find the possible bottlenecks to decrease the time spent on this step, or evaluate the use of another strategy to generate the output file.

Besides that, from Figures 5.9 and 5.10 we observe that the time needed to reach the edge-collapse target decreases as the number of threads increases. Additionally, we should take into account that the percentage of threads who are successful per iteration, and the number of edges evaluated and collapsed per iteration increases proportionally to the number of threads used by a certain configuration. We conclude that as the number of threads increases, the increase of unsuccessful and idle threads (and thus, leading to unused computational resources) is compensated with the decrease of the run-time of the algorithm (generating a trade-off between resource efficiency and run-time).

Although the percentage of candidate edges that can not be collapsed is higher with 512 threads than with a smaller amount of threads, the speedup is almost two times in favor of the 512 threads configuration versus configurations with fewer threads.

From Figure 5.33 to 5.36, we observe that the areas that are taken in account from which select edge candidates is increased, as the Quadric Error Metric threshold increases. Also, the areas are selected, prioritizing zones with zero or low height change between continuous elements. In contrast to the Edge Length Metric shown in Figure 5.30 to 5.32, who leaves unmarked edges between candidate edges, and may arise quality problems (and was the principal motive to discard the use of this metric in the final test scenarios).

Additionally, from Figure 5.37 to 5.42 we observe that as the number of edge-collapses performed increases, zones with fewer density of vertices begin to appear (light-colored zones), similar to the areas marked by the metric on the previous Figures. The areas of the mesh that lose element density are the areas who represent plain terrains, such as the sea and the valleys, while the areas that need a high level of detail, such as mountains, maintains a high level of density. This means that the election of the Quadric Error Metric to select which edges have to be collapsed after other was a good metric choice for our purpose.

From Figures 5.43, 5.44 and 5.45, we observe in this close-up of the lower-left zone of the mesh, that the resultant mesh is almost equal across those three scenarios (sequential and parallel with 128 and 512 threads). This tend us to believe that the results obtained from the algorithm are consistent between sequential and parallel implementations, and between the number of threads used on the parallel implementation. From Figures 5.46, 5.47 and 5.48 we observe a similar scenario with the center zone of the mesh, which let us to conclude the same as before.

From Figure 5.49 to 5.54 we observe that for the same algorithm configuration (parallel with 512 threads), as the number of edge-collapses performed increases, we see that the majority of the edge-collapses are performed on the zones of the mesh with low complexity (like the sea or the valleys), while the zones with high complexity (like the mountains or high slope areas) tends to have fewer edge-collapses performed on them. This behaves as expected, because the metric prioritizes zones with fewer complexity in order to preserve the overall shape and geometry of the mesh, and we observe that these properties are preserved as the number of edge-collapses performed increases. We can conclude the same from observing Figures 5.55 to 5.60.

Overall, we can observe that the qualitative results are very similar when comparing the same edge-collapse target with the sequential result and different thread configurations. We can conclude that the parallel configuration leads to similar results that the sequential configuration in fewer time, and when comparing only the parallel variants of the algorithm, the configurations with higher number of threads reach similar results that the configurations with lower number of threads, but in fewer time.

Chapter 6

Conclusions

The main objective of this thesis was to design and implement a novel parallel mesh simplification algorithm using the parallel processing power of the GPU, and evaluate its performance under the scenario of terrain meshes.

In this thesis, we have presented how to construct a parallel mesh simplification algorithm based on edge-collapse under a GPU architecture, with the use of redundant data structures to represent a mesh. Additionally, we provided a survey/review about the "state of the art" of mesh simplification algorithms (techniques, algorithms and considerations). Also, we integrated this parallel simplification algorithm on the Cleap library and evaluated its performance against an ad-hoc sequential simplification algorithm implementation.

First, we discuss the main conclusions about the main objectives mentioned at the beginning of this work, the contribution and relevance of this work, and finally, what can be improved in the future of this work.

6.1 Conclusions

The main contribution of this work is the presentation of a parallel edge-collapse algorithm that can take advantage of the high data-parallel performance of GPUs and can also repair inconsistencies on its data structuresin parallel as well. A great part of the benefits of the method come from the mesh data structure and its small redundant information that allows easy access among vertices, edges and triangles, as well as validation of the mesh primitives. Furthermore, the decision of using this data structure allows the integration of edge-collapse with other methods such as the transformation of any triangulation onto a Delaunay one via parallel edge-flips, entirely done in GPU.

We are satisfied with the results obtained, because we can generate simplified meshes with a parallel algorithm, with similar quality to the ones obtained with the sequential algorithm, in less time.

One limitation of the current approach, which is a potential performance bottleneck, is the sorting stage of the candidate edges by metric done on CPU. Clearly, the memory bandwidth bottleneck in data transfers between CPU and GPU can manifest at some point as the problem size increases. Fortunately there are GPU implementations of sorting algorithms such as radix sort, bitonic sorting networks and quicksort that can be used to improve this part of the overall algorithm.

The experimental results showed that the speedup over the sequential implementation was up to three orders of magnitude on the edge-collapse phases. We believe that more optimized CPU implementations can indeed shorten the performance gap found in our results, and we expect that a fully optimized multi-core CPU implementation could lower the GPU speedup to between one to two orders of magnitude which is what is typically observed, as seen in the review on Section 3.3. If we consider the times of the final vertices rearrange step (which is the part that generates the algorithm output), then this performance gap shortens up to at most two orders of magnitude. Nevertheless, our results show that the use of parallel processing greatly helps reduce the overall time of the simplification process.

Other potential performance bottleneck, is the final vertices rearrange step on Section 4.7.1. We observe from Tables 5.2 to 5.7 that the time spent in the vertices rearrange step is substantially higher (one order of magnitude higher) that the time spent on the simplification phases of the algorithm. One possible cause of the bottleneck is that, at least in the parallel scenario, the vertices rearrange is done with a fixed number of 256 threads. We made this decision because we needed to accelerate the time it was taking to generate the output mesh, to be able to visualize the mesh resulting from the simplification algorithm. A quick approach was to implement parts of the vertex rearrange algorithm in parallel. We used 256 because it was a large number and did not cause major problems in the rearrange algorithm. We believe that taking the appropriate considerations, it is possible to increase the amount of threads used in this phase, further reducing the time spent in this phase.

From the qualitative results obtained on the previous Section, we conclude that our choice of the use of Quadric error metric over the edge length metric was correct. We observed that this metric selected edges to be eliminated on areas of the mesh with low slope or low detail, which was one of our main objectives towards a simplification algorithm that works well under the terrain meshes scenario.

Also, From the charts of number of edge-collapses performed per iteration on Section 5.1, we observed that as the number of iteration increases, the percentage of successful edge-collapses decays, from an overall 50% to $20\% - 33\%$ rate of edge-collapses performed. Although it is expected, because as more simplification operations are done over the mesh, it is more likely to encounter candidate edges that do not pass all the consistency test. Regardless of the number of threads utilized, the overall rates of 50% to $20\% - 33\%$ successful edge-collapses prevails, could be explained for the strategy of sorting and then shuffling the pool of candidates (explained on Section 4.3), which generates a diverse selection of candidates that are mixed enough to not be very sensitive to the size of the thread pool. Additionally, we can conclude that this strategy is significantly better that the strategy of selecting contiguous candidates edges (which in early experimentation led to simplification rates lower that 10%).

Another contribution, is that this algorithm explicitly describes each of its steps, as shown on Chapter 4. Unlike other algorithms descriptions, where some of its steps are obfuscated or not so clear how they do it, principally on how they repair the inconsistencies on the data structures after a parallel edge-collapse step.

Finally, this works show how the use of redundant data structures and the reuse of unused allocated memory from deleted elements, greatly helps to efficiently implement the internal repair step shown on Sections 4.6.2, 4.6.3 and 4.6.4, where each vertex, triangle and edge can detect if it has inconsistencies and repair themselves independently of each other.

6.2 Future Work

Future work may consider porting the candidate sorting stage to the GPU which would make the entire algorithm be executed on the GPU, having the CPU only acting as a manager for scheduling the iterations of the parallel edge-collapse waves. Further research can explore how larger batches of threads behave on newer GPU architectures, get the optimal number of threads that should be launched in parallel given certain parameters (to minimize the number of idle threads), as well as explore the cost of the method and quality of the result on more complex or different kind of meshes, such as meshes with a high count of valley zones or meshes with a high change of attitude, like a zone full of mountains of varying height.

Complementing the previous idea, we would like to evaluate the use and integration of other edge-collapse cost metrics that we reviewed in Section 3.2. The code is writen in a way that it does not require a lot of effort to add a new cost metric to the algorithm. Depending of the kind of mesh that will be simplify, some metrics may behave better than others in particular scenarios, such as the mountain mesh stated earlier, or may lead to different run-time v/s quality of resultant mesh scenarios. Furthermore, we would like to evaluate the quality of the results if we do not perform the candidate edges sorting stage explained on section 4.3, like other algorithms do, for example, the Frank-Scala Algorithm[5].

Although the rate of successful edge collapses per iteration that our strategy of sorting and then shuffling the pool of candidates (Section 4.3) is way better that the strategy of selecting contiguous candidates, we would like to investigate if there exist other strategies for selecting the candidates that may lead to better successful rates, which impacts in reducing the number of idle threads and waste of computational resources.

One of the restrictions of the proposed algorithm is that the border-line edges (Section 4.5.2) cannot be candidates for simplification. However, there may be situations where we want to simplify the border-line edges, without affecting the geometry of the mesh. One approach that we want to evaluate, is the simplification of the border-line edges with a half-edge collapse technique, where the border-line edges can be candidates if and only if we restrict the target of the edge-collapse only to the border of the mesh (with a half-edge collapse), which theoretically should preserve the mesh properties.

Additionally, we would like to assess if it is possible to reevaluate some of the discarded candidate edges in future iterations, principally the ones discarded by the concurrency test, because they were not collapsed for "bad luck" reasons, instead of geometric reasons (like exceeding the number of elements in the vicinity or being a border-edge), and they may still be suitable candidates in the next iterations.

Given that this algorithm is integrated in the Cleap library, we would like to evaluate if performing a Delaunay triangulation refinement step between parallel iterations of the simplification algorithm, or before the final repair step to generate the output mesh, would lead to a better quality of resultant meshes. This is supported by the fact that a Delaunay triangulated mesh maximizes the minimum angle of all triangles in the triangulation, and so they tend to avoid sliver triangles, which are not desirable for geometric applications. One thing to consider is, if this Delaunay refinement step is performed between parallel iterations, it can enable or disable some edges to be suitable candidates for edge-collapse, so it may be necessary to implement a candidate reevaluation step at some point before the simplification is performed again after a Delaunay Triangulation phase.

Because the implementations of the algorithms reviewed in Section 3.3 were not available for public use, we could not do a proper comparison between our algorithms and the algorithms from other authors, and we had to do the comparison with and ad-hoc implementation. In future iterations of this work we would like to do a comparison with those algorithms, either a implementation made by themselves, or an implementation made by ourselves based on the description of the algorithms made in their respective works.

We also observed in the results chapter, that the final vertices rearrange phase (Section 4.7.1) takes a lot of time in comparison with the other stages of the algorithm, and it is a necessary step in order to generate the output mesh. We would like to evaluate if it is possible to do this final rearrange step more efficiently, either increasing the number of threads, or use a more clever way to rearrange the vertices, or finding alternatives to this final step that let us to generate the same output mesh more efficiently.

Finally, the implementation of the proposed method will be made available for the community as part of the Cleap library with a simple API to access the functionalities in applied scenarios.

Bibliography

[1] Nvidia CUDA. `http://www.nvidia.com/object/cuda_home_new.html`, 2014. [Online; ultimo acceso 10-Noviembre-2017].

[2] Fabien Cellier, Pierre-Marie Gandoin, Raphaëlle Chaine, Aurélien Barbier-Accary, and Samir Akkouche. Simplification and streaming of gis terrain for web clients. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 73–81, New York, NY, USA, 2012. ACM.

[3] Frank Fisher, Ronald A.; Yates. *Statistical tables for biological, agricultural and medical research (6rd ed.)*. Edinburgh: Oliver and Boyd, 1963.

[4] Martin Franc and Václav Skala. Parallel triangular mesh reduction. In *Proceedings of ALGORITMY 2000 Conference on Scientific Computing*, pages 357–367, 2000.

[5] Martin Franc and Václav Skala. Parallel triangular mesh decimation without sorting. In *Proceedings of the 17th Spring Conference on Computer Graphics*, SCCG '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Michael Garland. *Quadric-based Polygonal Surface Simplification*. PhD thesis, Pittsburgh, PA, USA, 1999. AAI9950005.

[7] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[8] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 263–269, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[9] Nico Grund, Evgenij Derzapf, and Michael Guthe. Instant level-of-detail. In Peter Eisert, Joachim Hornegger, and Konrad Polthier, editors, *VMV*, pages 293–299. Eurographics Association, 2011.

[10] Jon Hjelmervik and Jean-Claude Leon. Gpu-accelerated shape simplification for mechanical-based applications. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, SMI '07, pages 91–102, Washington, DC,

USA, 2007. IEEE Computer Society.

[11] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[12] Donald E Knuth. *Seminumerical algorithms*. Addison–Wesley. pp. 139–140, 1969.

[13] Martin Kraus and Thomas Ertl. Simplification of nonconvex tetrahedral meshes. In *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 185–196. Springer-Verlag, 2002.

[14] Hyunho Lee and Min-Ho Kyung. Parallel mesh simplification using embedded tree collapsing. *Vis. Comput.*, 32(6-8):967–976, June 2016.

[15] S. Melax. A simple, fast, and effective polygon reduction algorithm. `http://dev.gameres.com/program/visual/3d/PolygonReduction.pdf`, November 1998. [Online; ultimo acceso 10-Noviembre-2016].

[16] Valentín Muñoz. Paralelización de algoritmos de mallas geométricas en gpu. `http://repositorio.uchile.cl/handle/2250/131809`, December 2014.

[17] Cristobal Navarro. Librería Cleap. `http://users.dcc.uchile.cl/~crinavar/doc/cleap/`, 2011. [Online; ultimo acceso 10-Noviembre-2017].

[18] Cristobal A. Navarro, Nancy Hitschfeld-Kahler, and Eliana Scheihing. A gpu-based method for generating quasi-delaunay triangulations based on edge-flips. In Sabine Coquillart, Carlos Andújar, Robert S. Laramee, Andreas Kerren, and José Braz, editors, *GRAPP/IVAPP*, pages 27–34. SciTePress, 2013.

[19] Alexandros Papageorgiou and Nikos Platis. Triangular mesh simplification on the gpu. *Vis. Comput.*, 31(2):235–244, February 2015.

[20] Remi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 1996.

[21] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falcidieno and Tosiyasu L. Kunii, editors, *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pages 455–465. Springer, 1993.

[22] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, July 1992.

[23] S. Shontz and D. Nistor. Cpu-gpu algorithms for triangular surface mesh simplification. In *Proceedings of the 21st International Meshing Roundtable*, pages 475–492. Springer Berlin Heidelberg, 2013.

[24] Robert F. Tobler and Stefan Maierhofer. A mesh data structure for rendering and

subdivision.

[25] Jiri Vad'ura. Parallel mesh decimation with gpu. http://eeict.feec.vutbr.cz/2011/sbornik/03-Doktorske%20projekty/09-Grafika%20a%20multimedia/07-ivadura.pdf, 2011.

Appendices

Appendix A

Parallel Lock-down Mechanism Example

In the following Figures, we show a sequence of steps performed by the Parallel Lockdown step of the algorithm, described in Section 4.4. This series of examples show two threads handling an edge competing for the lock-down of its respective neighborhoods.

A.1 Example 1: Two threads compete for a shared area

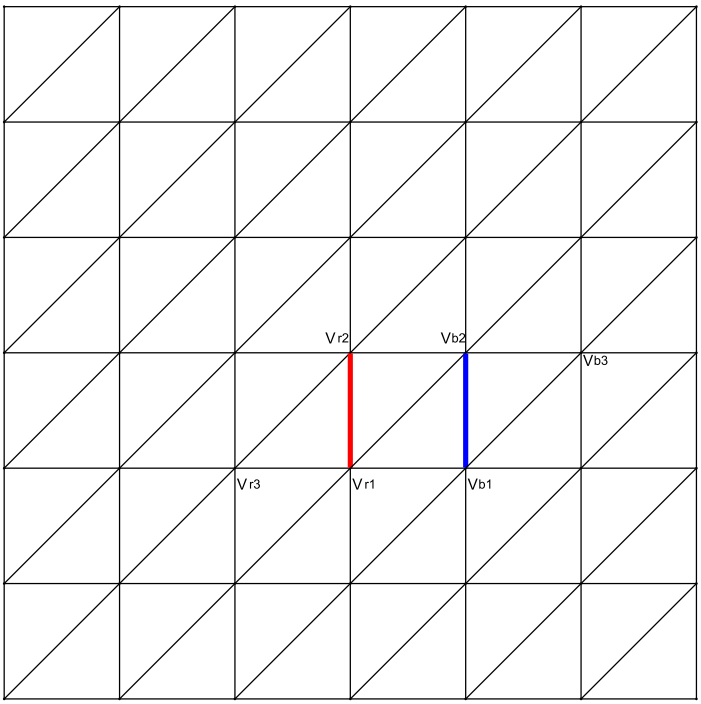


Figure A.1: Edges $\overline{V_{r1}V_{r2}}$ (red) and $\overline{V_{b1}V_{b2}}$ (blue) need to compete to gain control over a common shared vicinity.

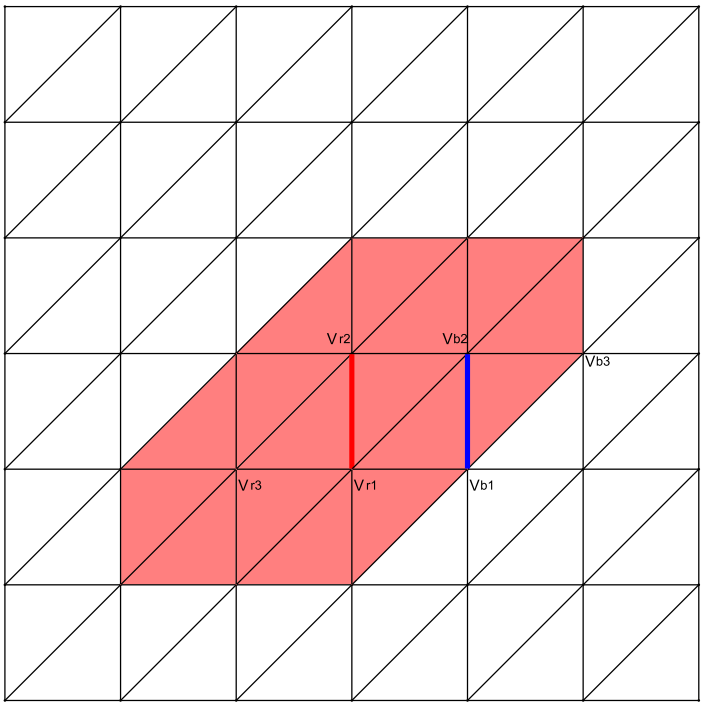


Figure A.2: Vicinity that the red edge needs to lock-down.

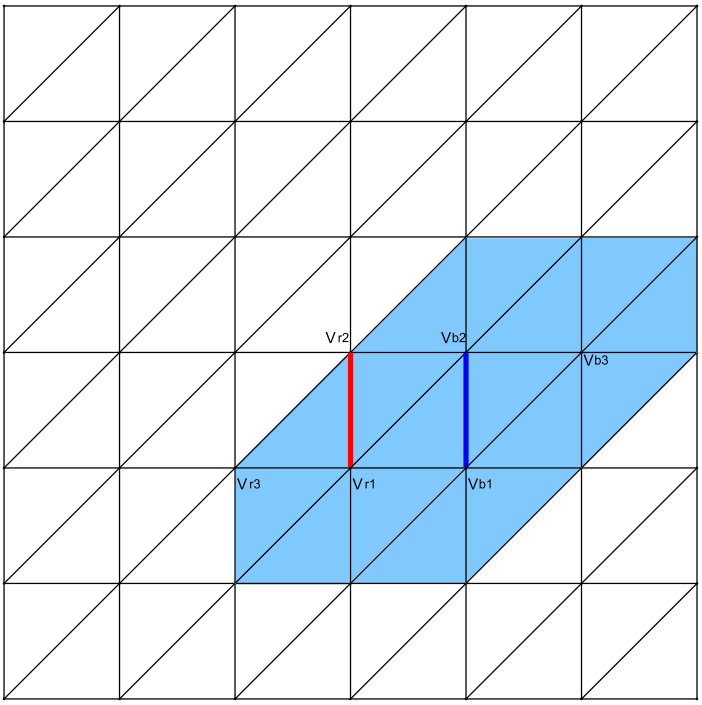


Figure A.3: Vicinity that the blue edge needs to lock-down.

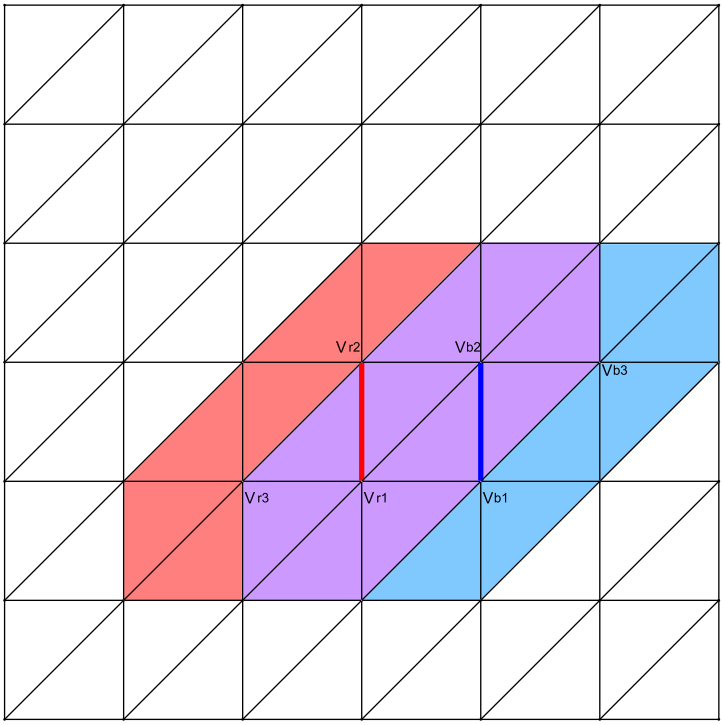


Figure A.4: Vicinity shared by both edges (in purple), and only one of them may have control over it at a given time.

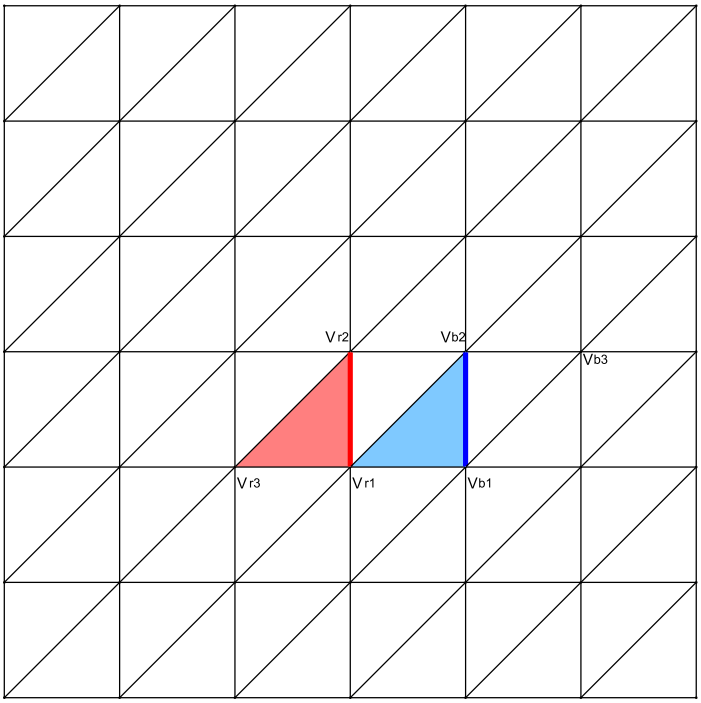


Figure A.5: First step of the mechanism, in which each edge locks down a triangle of their respective vicinity (V_{r1} and V_{b1}).

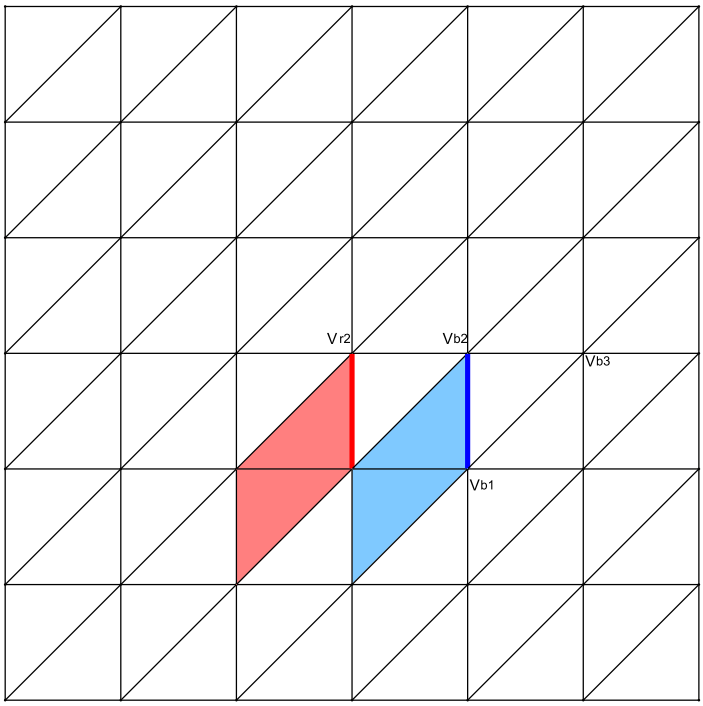


Figure A.6: Second step of the mechanism, in which each edge locks down another triangle of their respective vicinity.

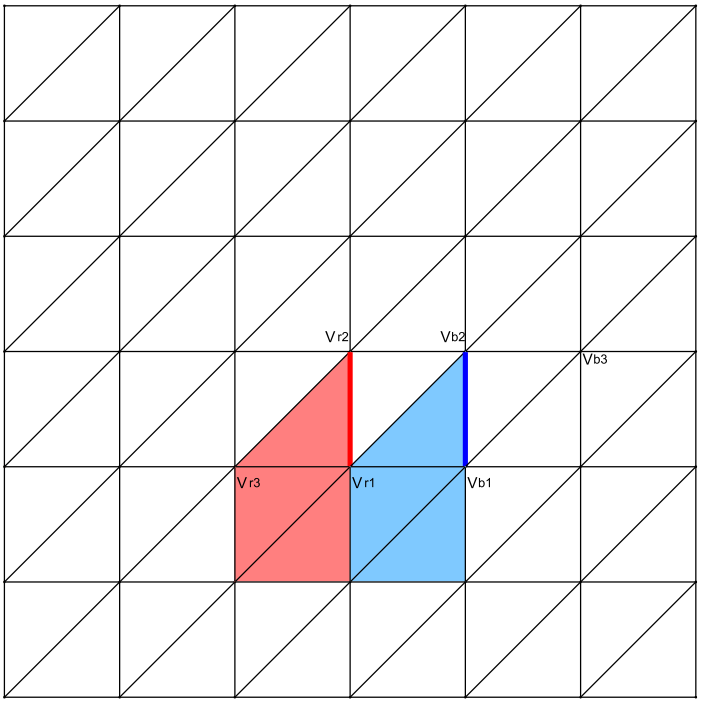


Figure A.7: Third step, in which each edge locks down another triangle of their respective vicinity.

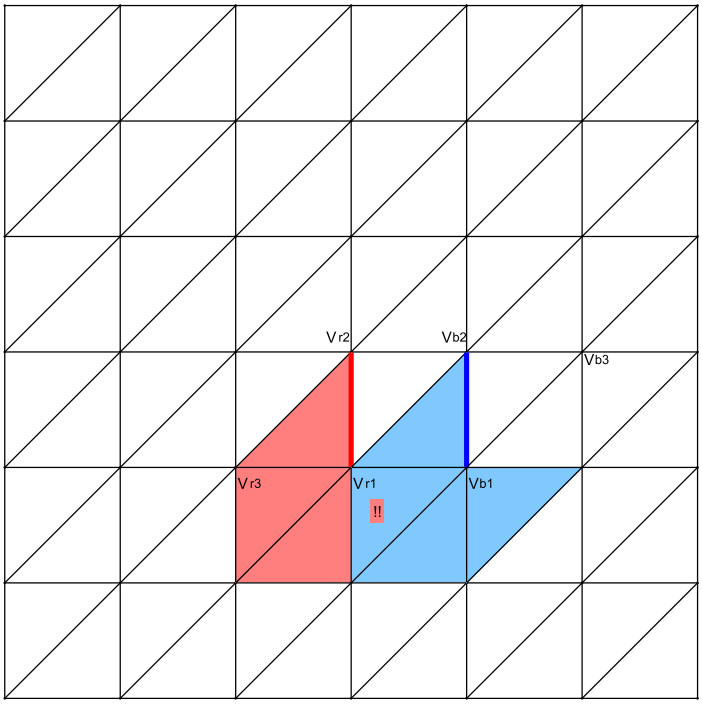


Figure A.8: Fourth step, in which the red edge tries to lock-down a triangle previously taken by the blue edge.

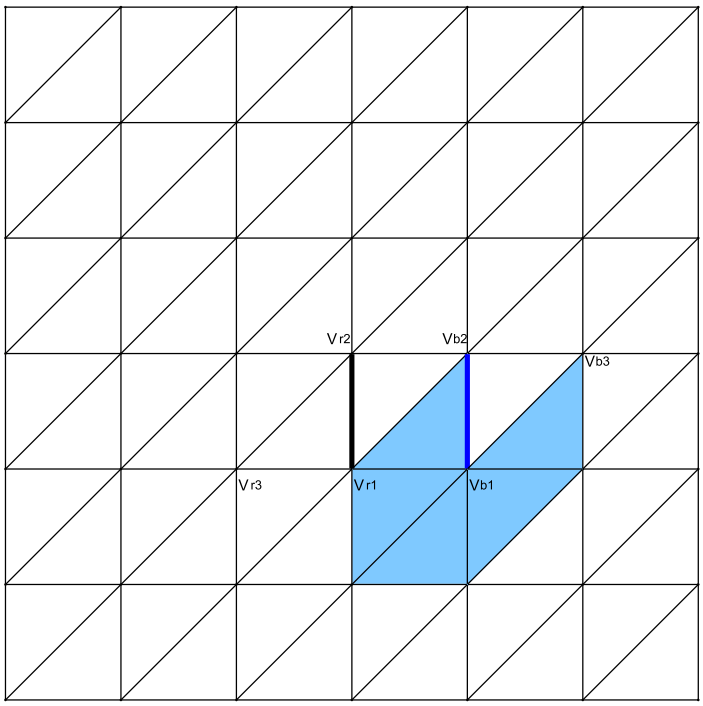


Figure A.9: Fifth step, in which the red edge had to release the lock-down over its triangles, because it can not aspire to obtain the lock-down over all the needed triangles.

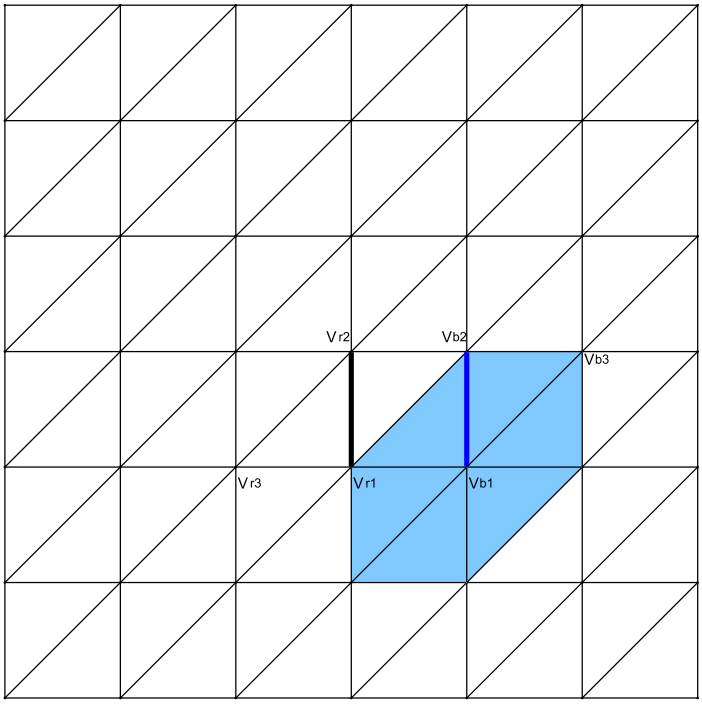


Figure A.10: Sixth step, in which the blue edge has lock-down of all triangles surrounding one of its vertices.

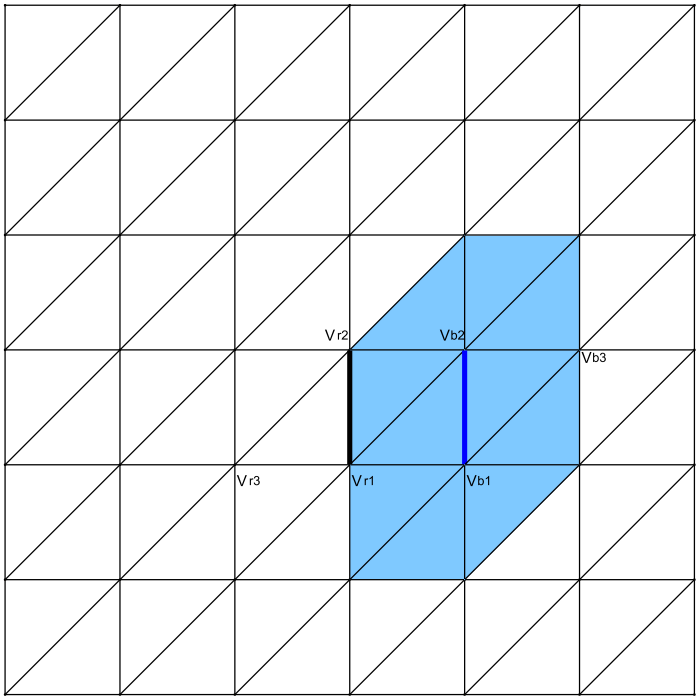


Figure A.11: Twelfth step, in which the blue edge has lock-down of all triangles surrounding both of its vertices (V_{b1} and V_{b2}).

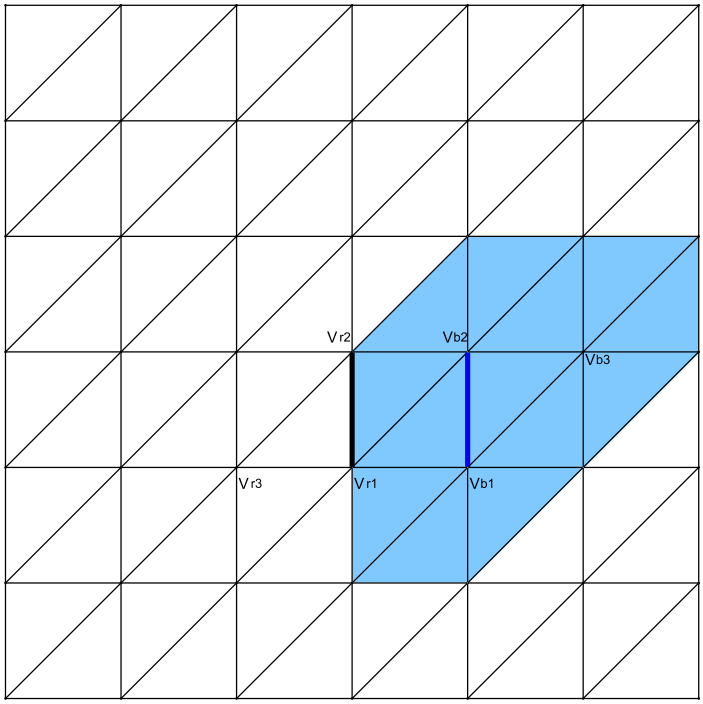


Figure A.12: Eighteenth step, in which the blue edge has gained lock-down of all triangles surrounding one of its opposite vertices (V_{b3}) .

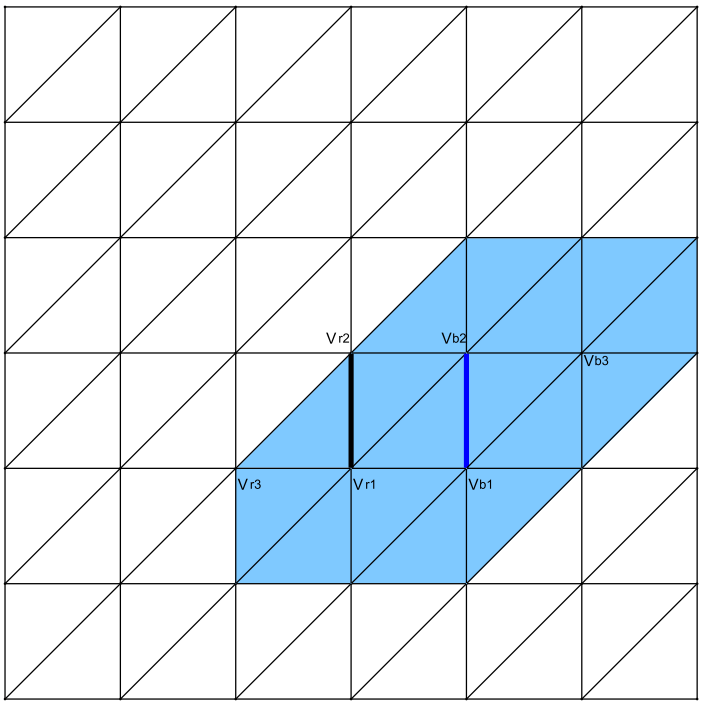


Figure A.13: Twenty-fourth step, in which the blue edge has gained lock-down of all triangles surrounding both of its opposite vertices (V_{b3} and V_{r1}).

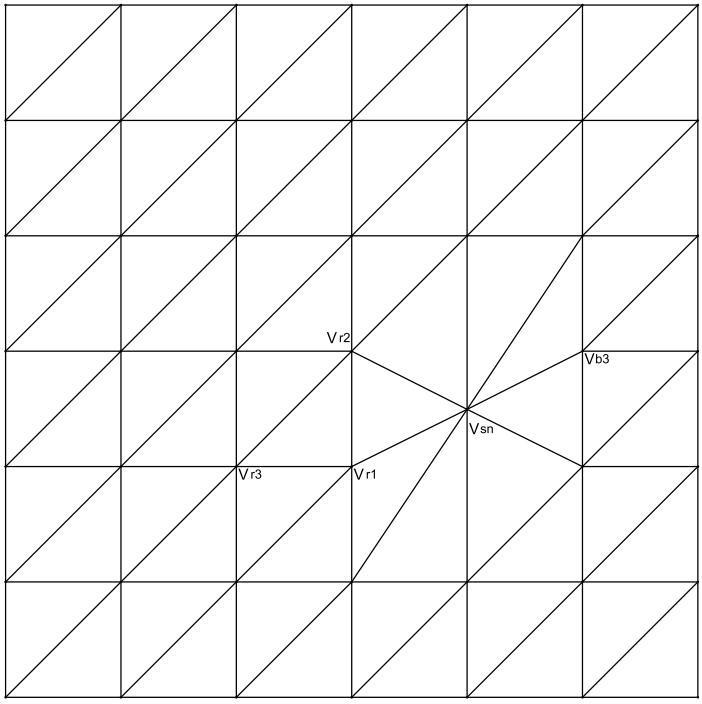


Figure A.14: When the blue edge has lock-down of all triangles in their vicinity, it can proceed to the simplification phase described in Section 4.6. This Figure shows the expected result.

A.2 Example 2: Two threads that do not compete for a shared area

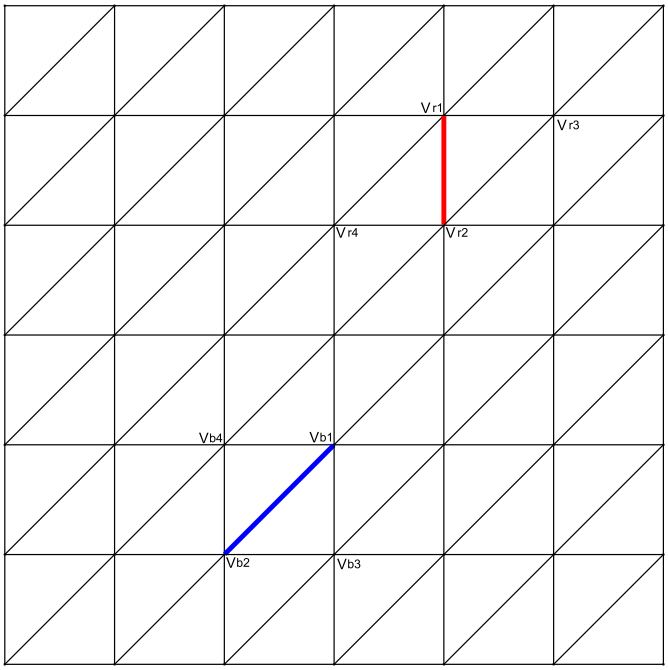


Figure A.15: Edges $\overline{V_{r1}V_{r2}}$ (red) and $\overline{V_{b1}V_{b2}}$ (blue) need to compete to gain control over its respective vicinity.

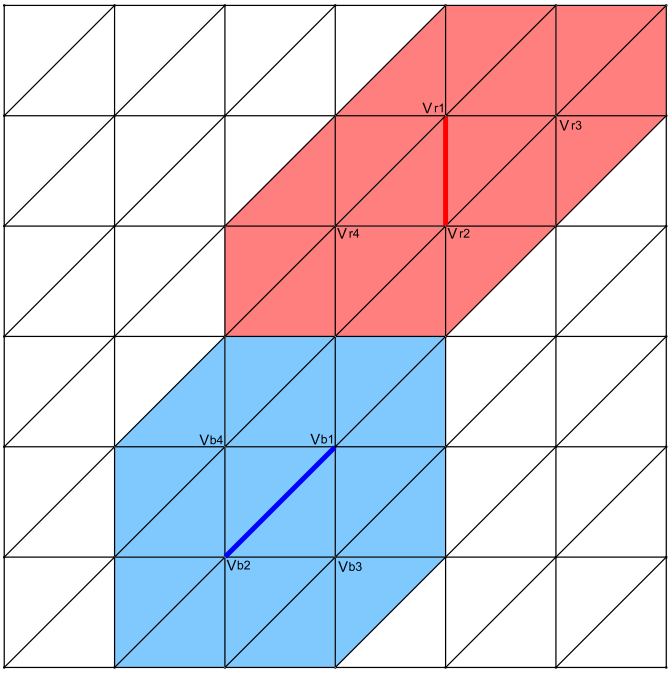


Figure A.16: Vicinity that each edge needs to lock-down. In this case, they do not share common areas.

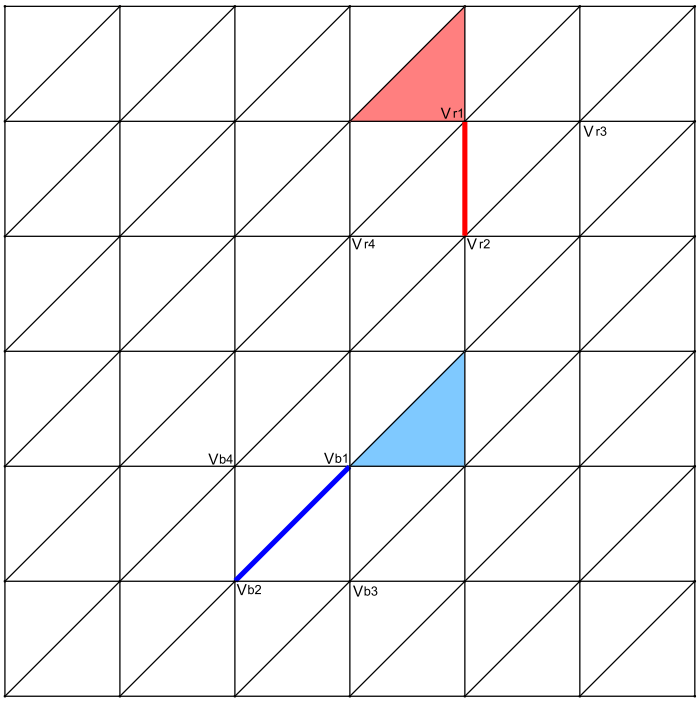


Figure A.17: First step, in which each edge locks down a triangle in their vicinity (V_{b1} and V_{r1}).

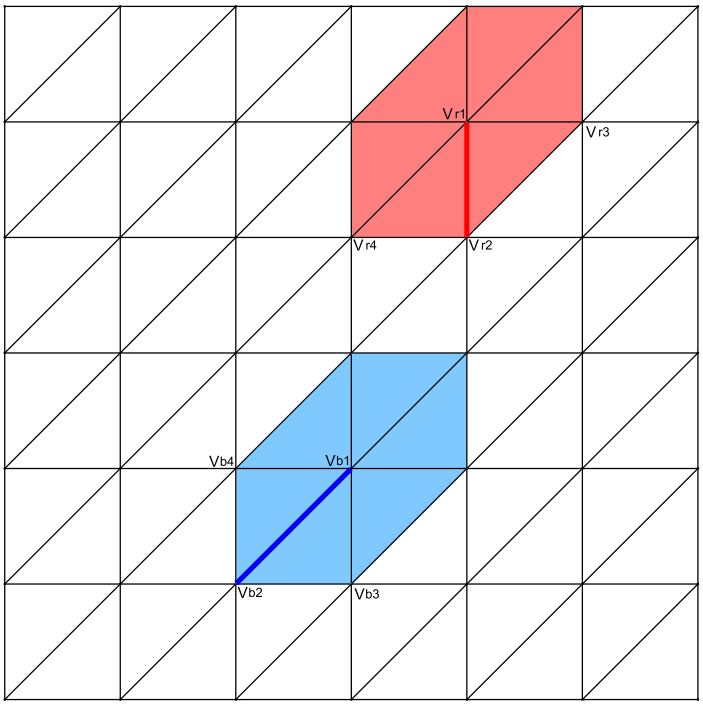


Figure A.18: Sixth step, in which each edge locks down the triangles surrounding one of its vertices (V_{b1} and V_{r1}).

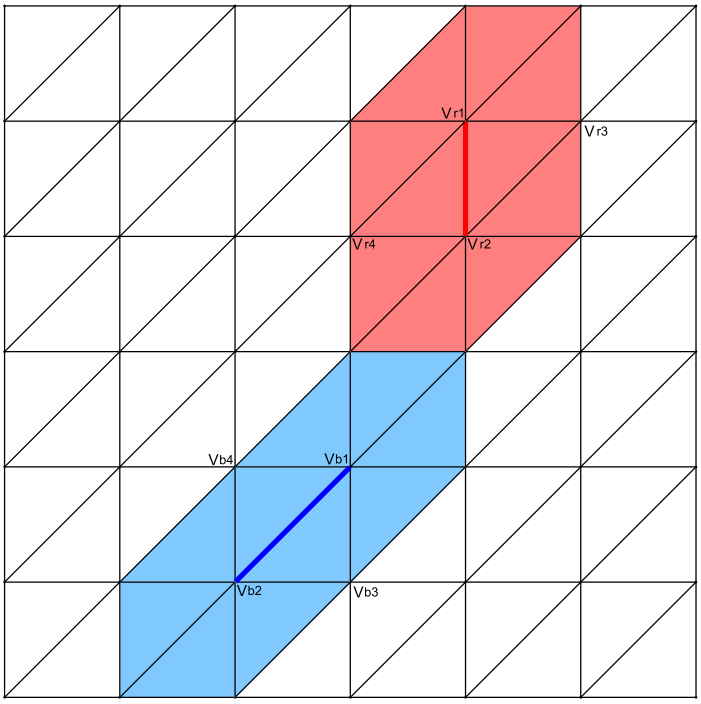


Figure A.19: Twelfth step, in which each edge locks down the triangles surrounding both of its vertices (V_{b1}, V_{b2}, V_{r1} and V_{r2}).

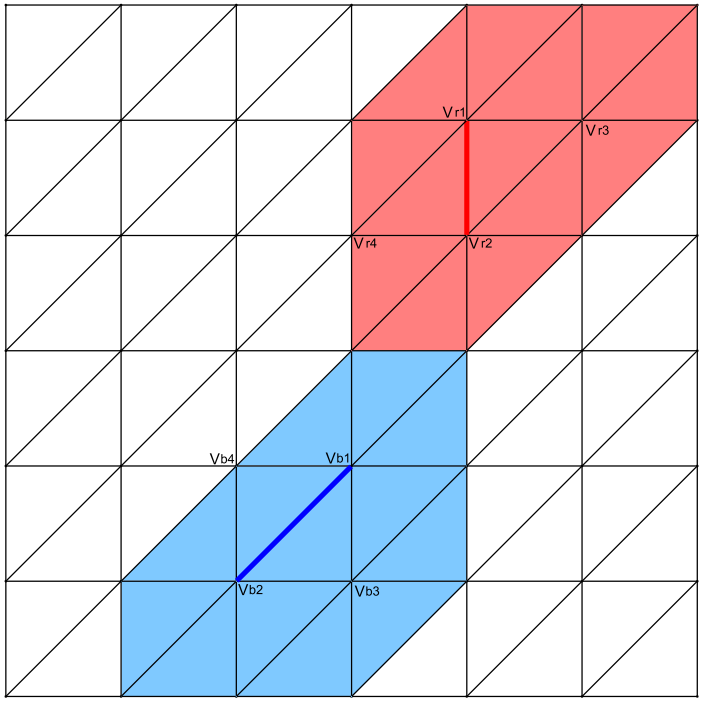


Figure A.20: Eighteenth step, in which each edge locks down the triangles surrounding one of its opposite vertices (V_{b3} and V_{r3}).

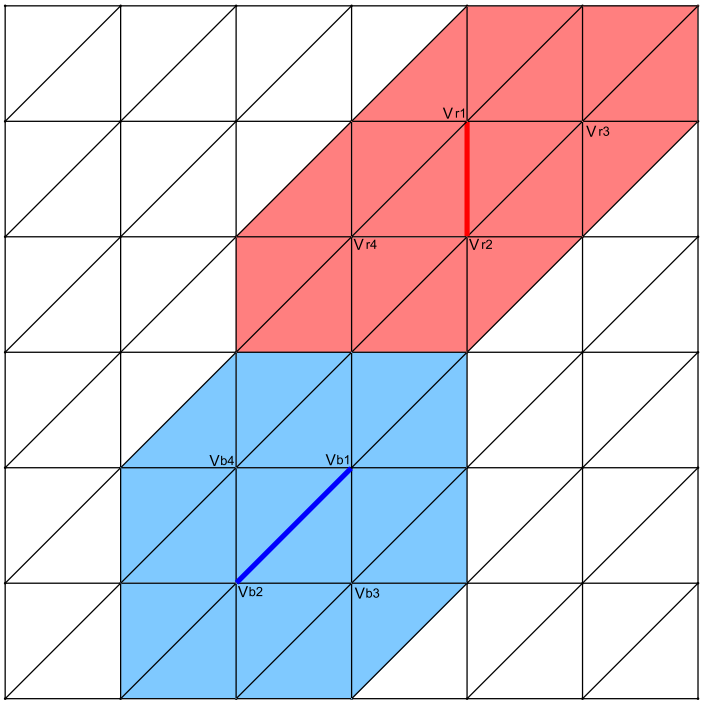


Figure A.21: Twenty-fourth step, in which each edge locks down the triangles surrounding both of its opposite vertices (V_{b3}, V_{b4}, V_{r3} and V_{r4}).

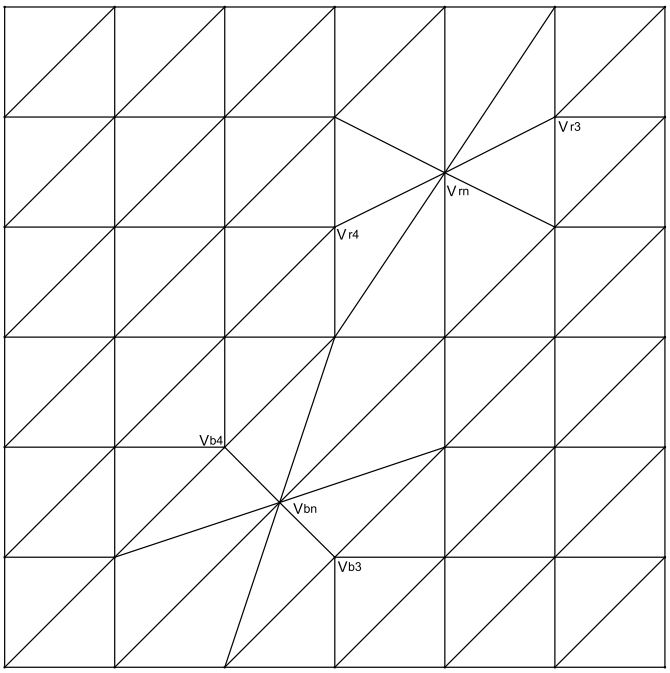


Figure A.22: When each edge has lockdown over all of the triangles in their vicinity, they can proceed to the simplification phase, without interfering each other.

Appendix B

Adjacency Table Example

In the following Figures, we show how the adjacency table shown in Section 4.2 is modified at the first step of the main part of the simplification algorithm described in Section 4.6, to reflect the changes on the vicinity of each vertex of the mesh. The left side of each Figure shows the status of the triangle incidency table, while the right side shows the status of the connectivity of the mesh.

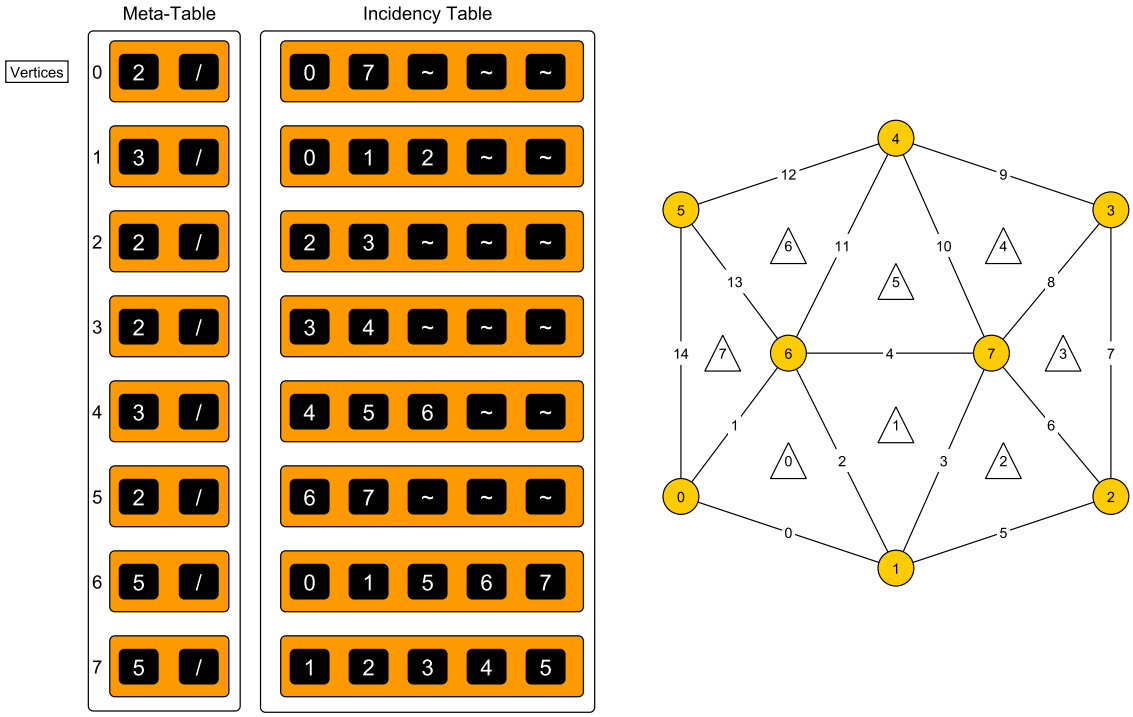


Figure B.1: Initial State before the simplification algorithm starts. The edge number 4 is the simplification target.

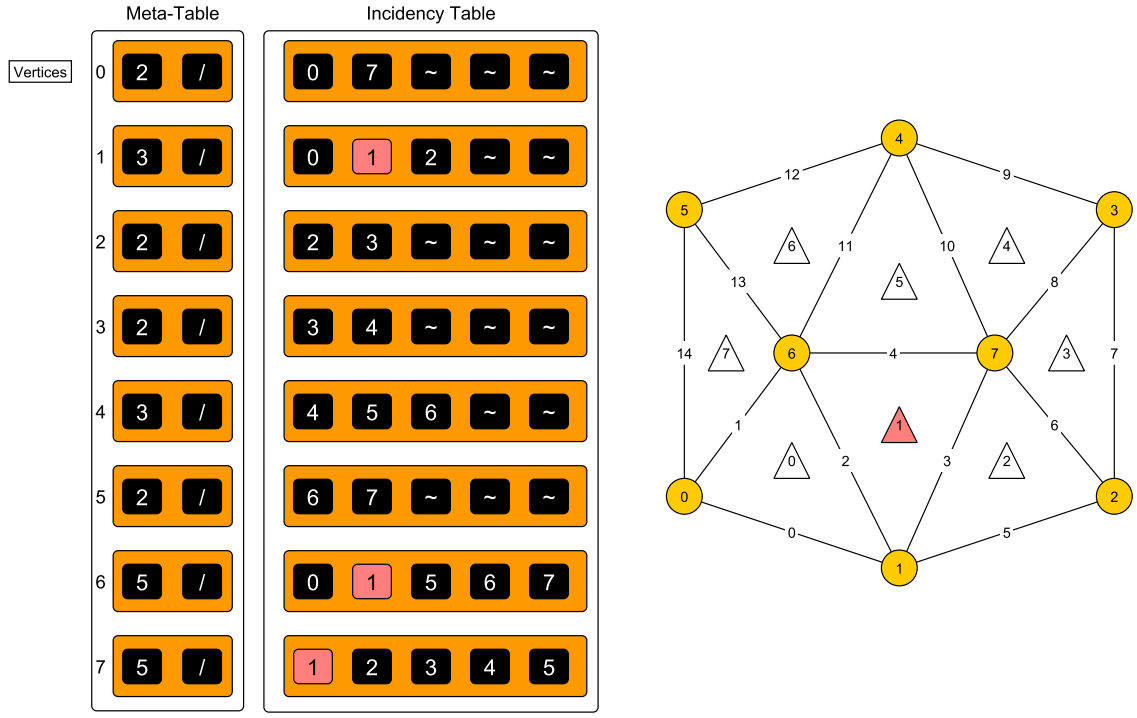


Figure B.2: We identify one of the triangles that will be deleted after the collapse (Triangle number 1), and detect which vertices have a reference to it.

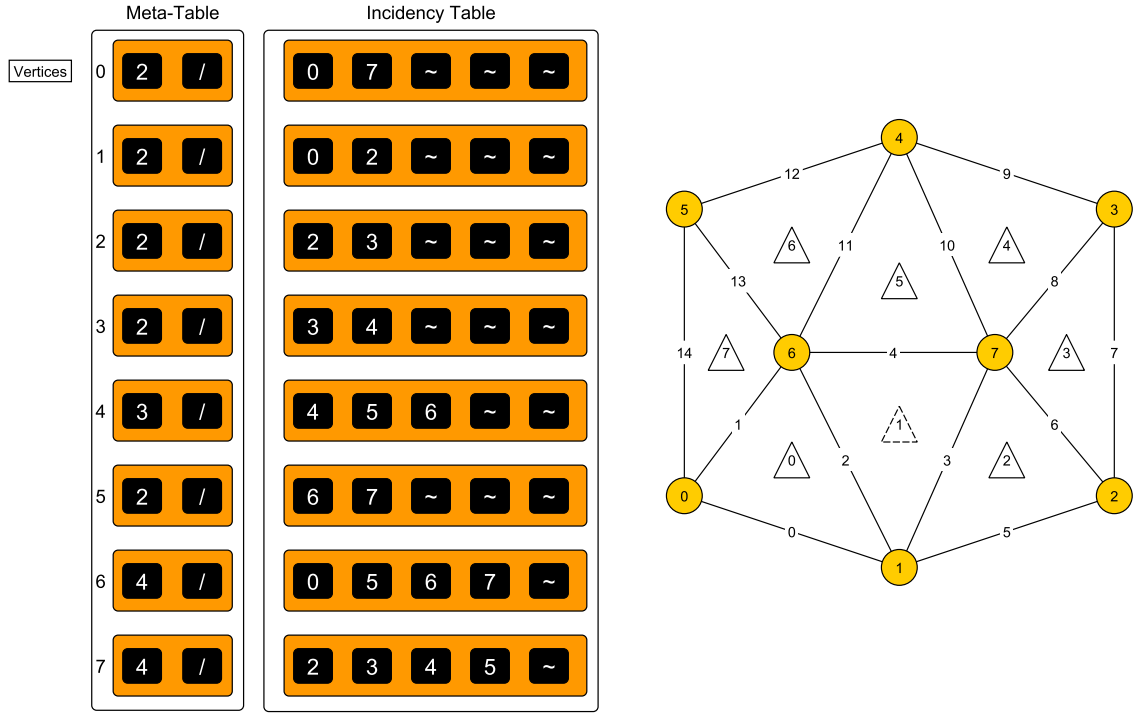


Figure B.3: We delete the references detected in the previous step, compact the adjacency table and update the number of incident triangles in the meta-table.

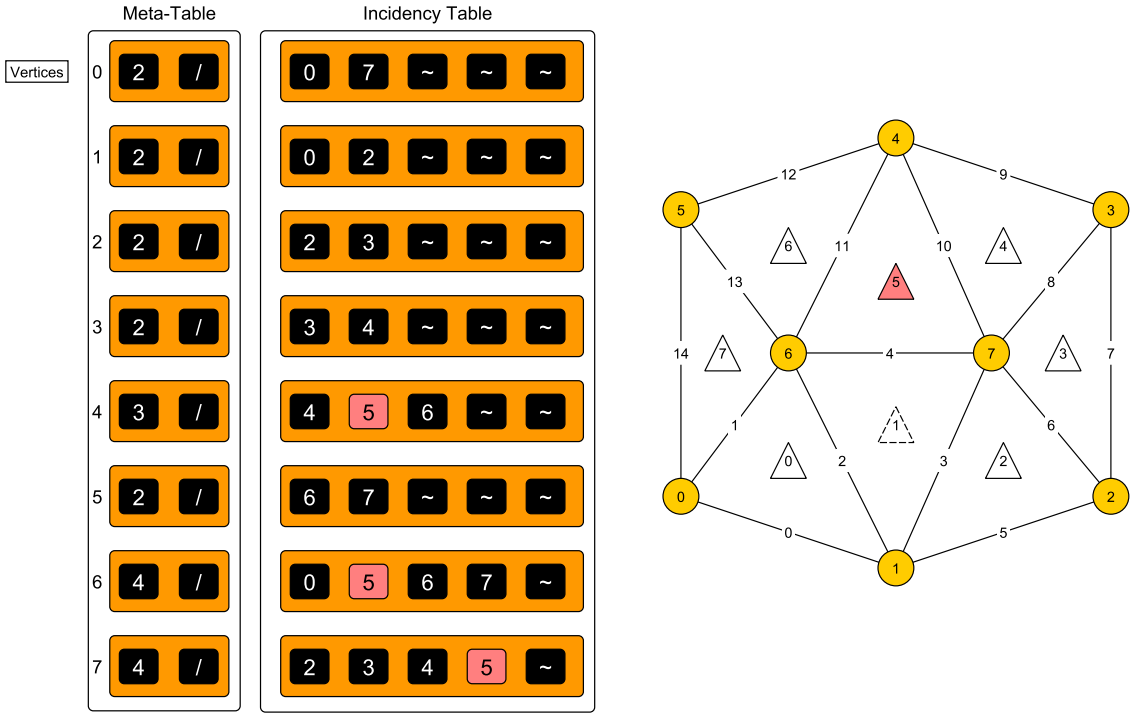


Figure B.4: We identify the other triangle that will be deleted after the collapse (Triangle number 5), and detect which vertices have a reference to it.

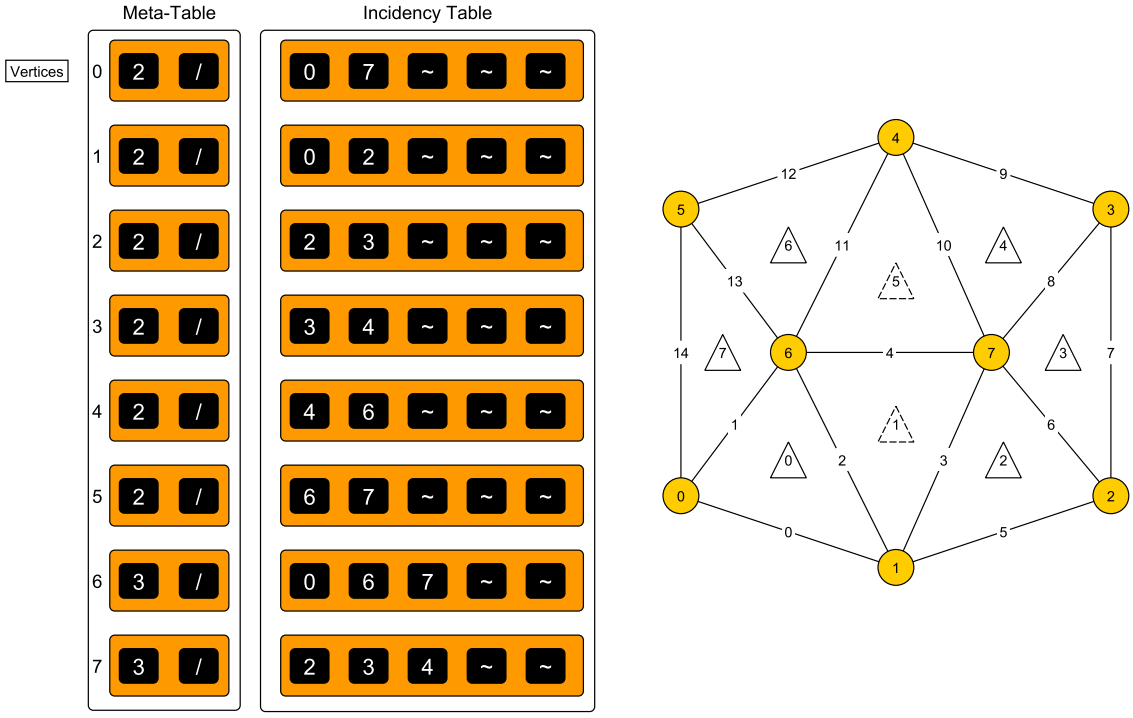


Figure B.5: We delete the references detected in the previous step, compact the adjacency table and update the number of incident triangles in the meta-table.

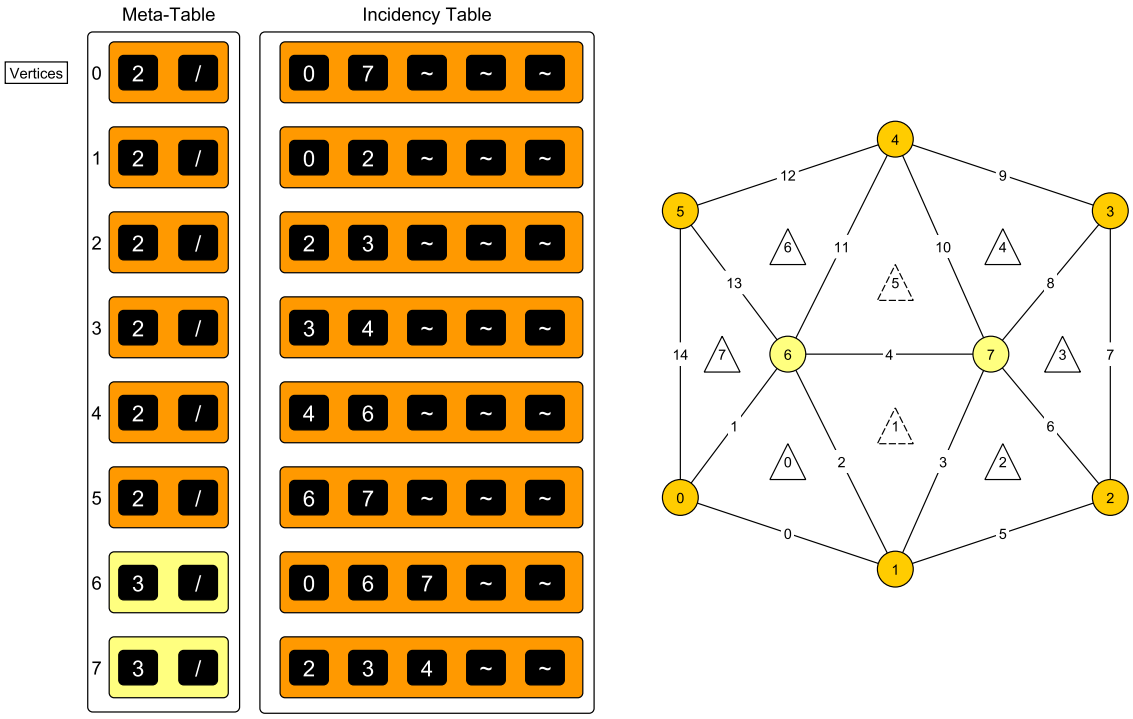


Figure B.6: We detect the vertices that will merge its data because of the edge-collapse.

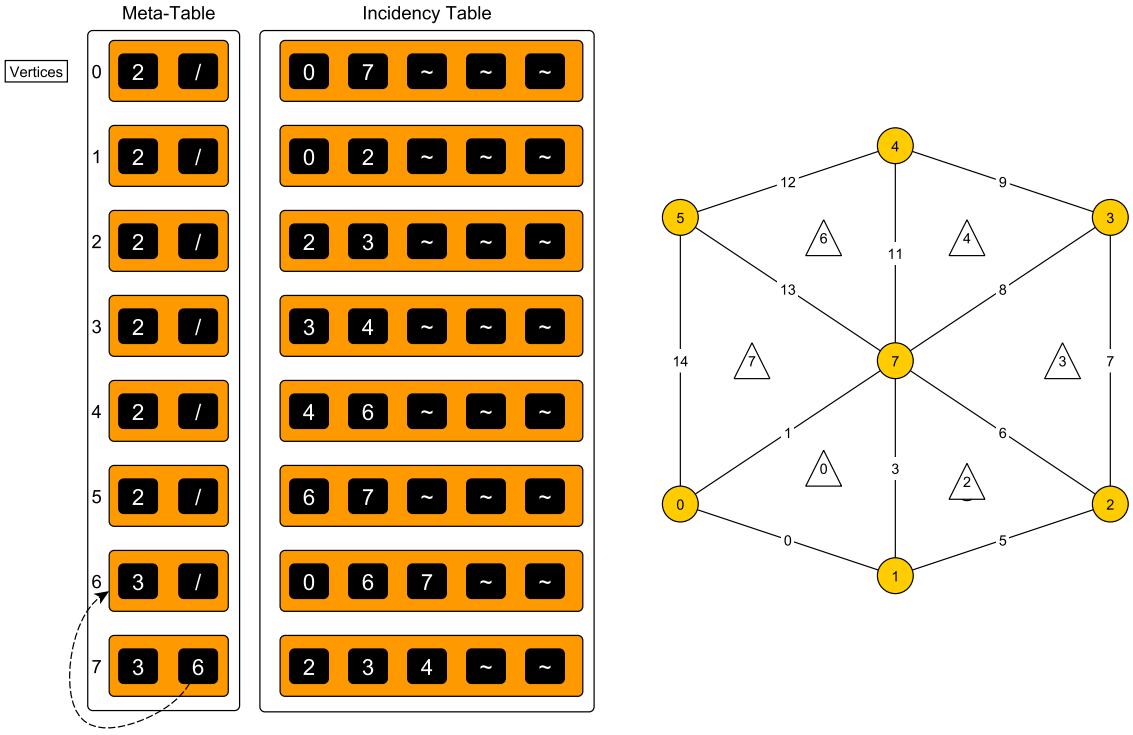


Figure B.7: We change the field *continues* in the meta-table from the vertex with higher ID, to reference the other one, and thus formalizing that the incident triangle data of vertex 7 continues in the memory space of vertex 6. This concludes the incident triangle repair step.

Appendix C

Edge-Collapse Algorithm Example

In the following Figures, we show how the Cleap data structure described in Sections 2.3 and 4.2 is modified at each significant step of the parallel simplification algorithm described in Section 4.6. The left side of each Figure shows the status of the Data structure that holds the mesh data (vertices, triangles and edges), while the right side graphically shows the status of the connectivity of the mesh.

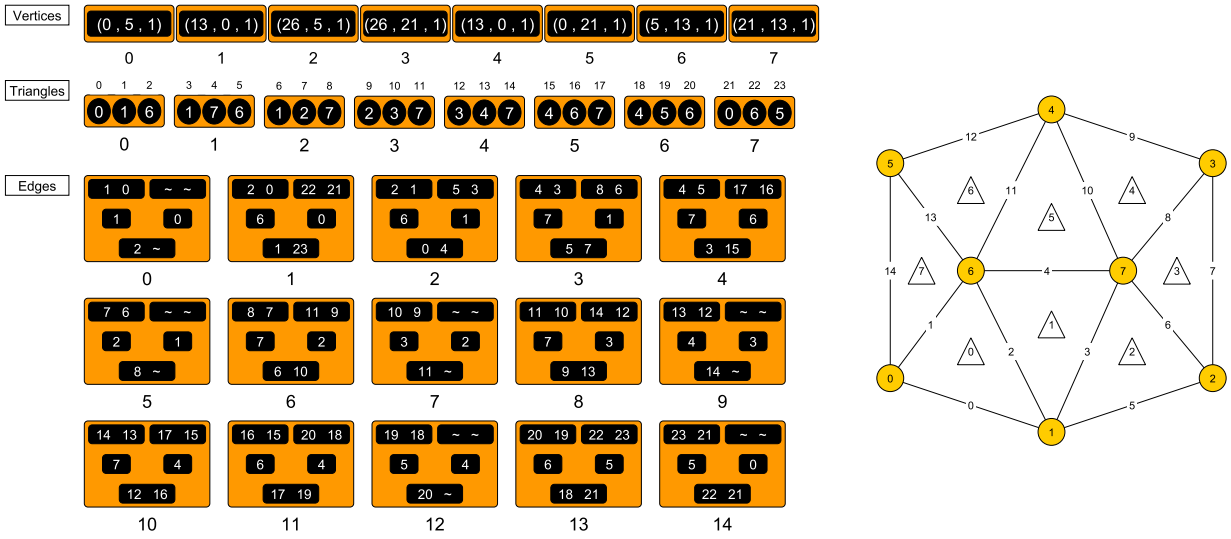


Figure C.1: Initial state of the data structure before the edge-collapse procedure.

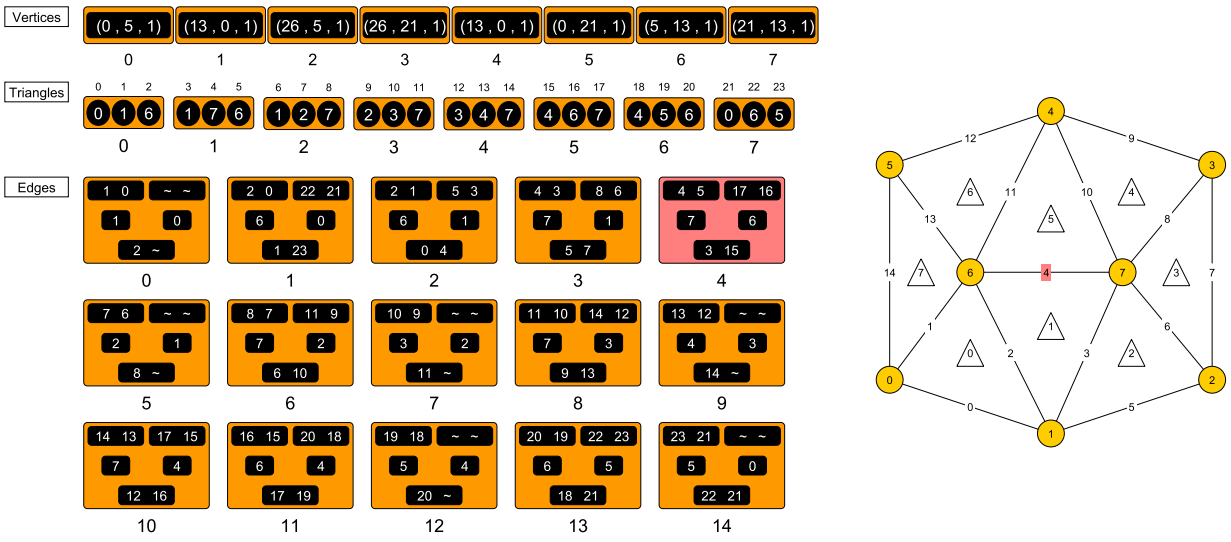


Figure C.2: Travelling all edges in parallel, each thread identifies its assigned edge that will be deleted in the data structure (for this example, edge number 4, marked in red). This corresponds to the step explained on Section 4.6.1.

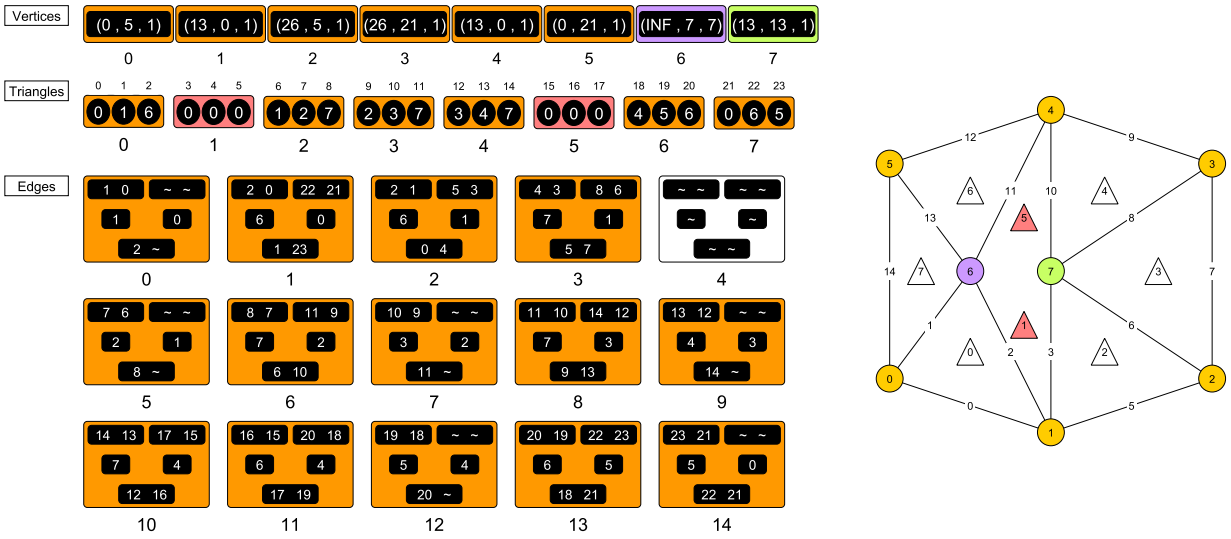


Figure C.3: Using the references on that edge, we identify the triangles and vertex that will be deleted. The data stored in the triangles is deleted and in the vertex space we write the ID of the other vertex of the edge that survived deletion and has the coordinates of the new position resulting from the edge-collapse. Finally, we delete the data stored on this edge.

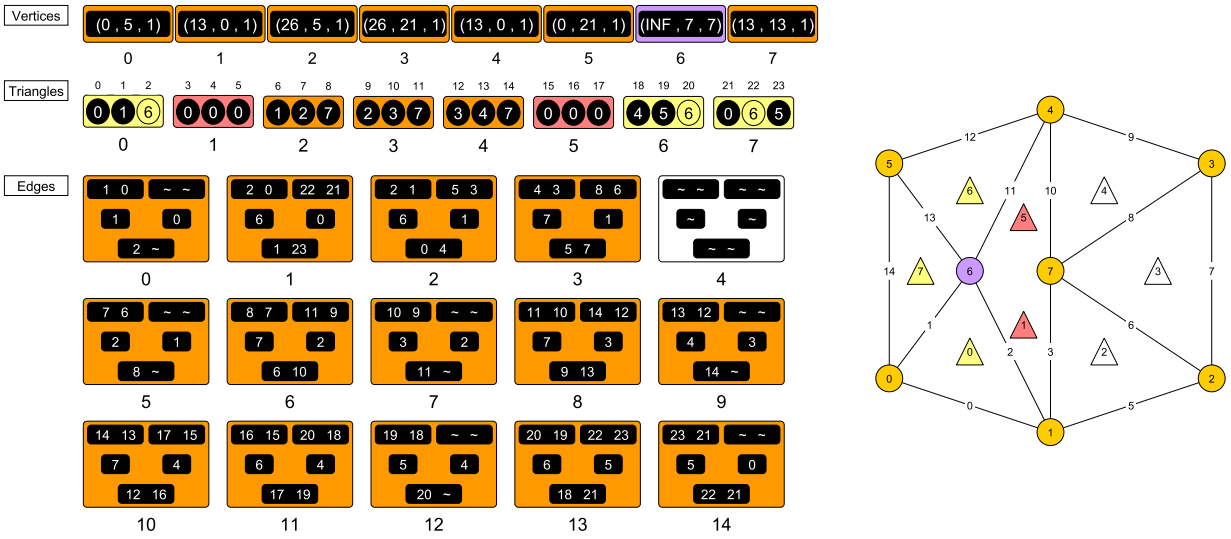


Figure C.4: Traveling all triangles in parallel, we identify the ones that have a reference to a deleted vertex (triangles 0, 6 and 7 have a reference to vertex 6). They detect if a vertex is deleted if its first coordinate has an INF value. This corresponds to the step explained in Section 4.6.2.

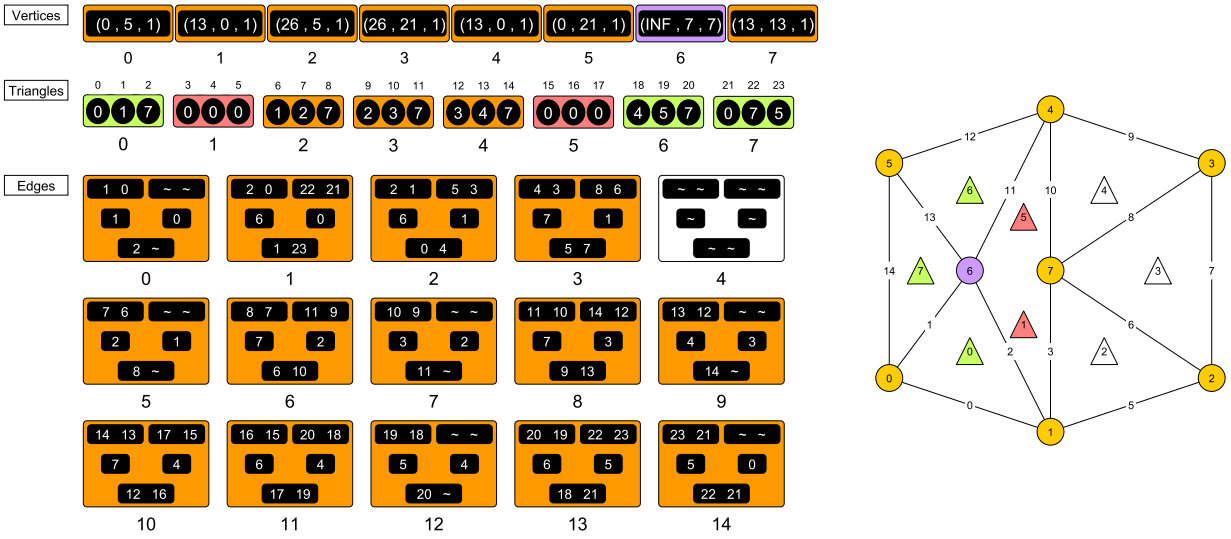


Figure C.5: Reading the vertex ID stored on the deleted vertex, each triangle repairs its respective inconsistency, and now they refer to vertex 7.

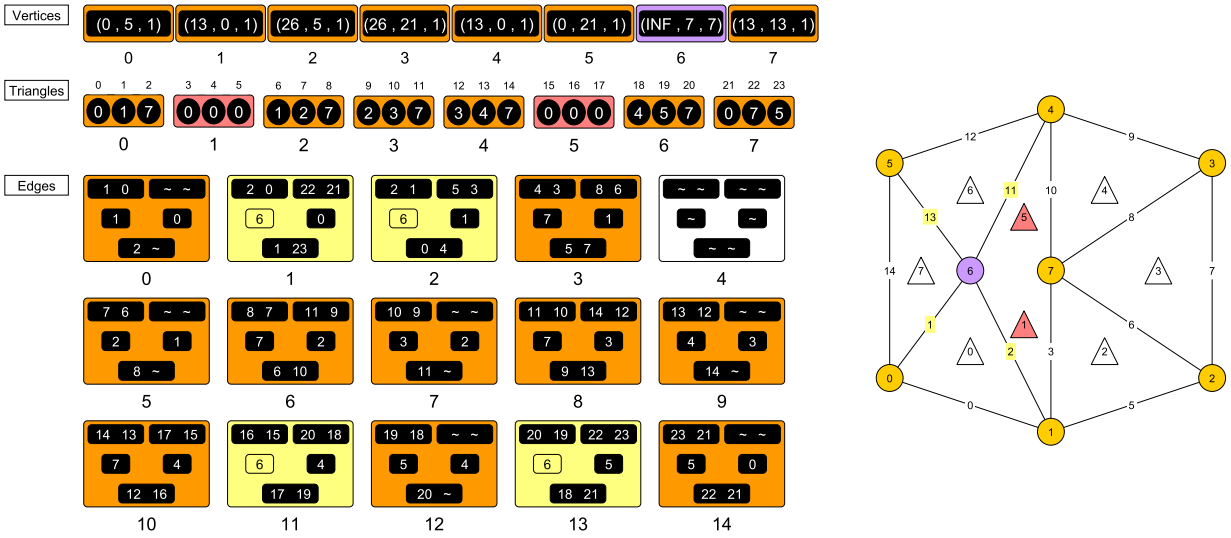


Figure C.6: Traveling all edges in parallel, we identify the ones that have a reference to a deleted vertex (edges 1, 2, 11 and 13 have a reference to vertex 6). This corresponds to the third step explained on Section 4.6.3.

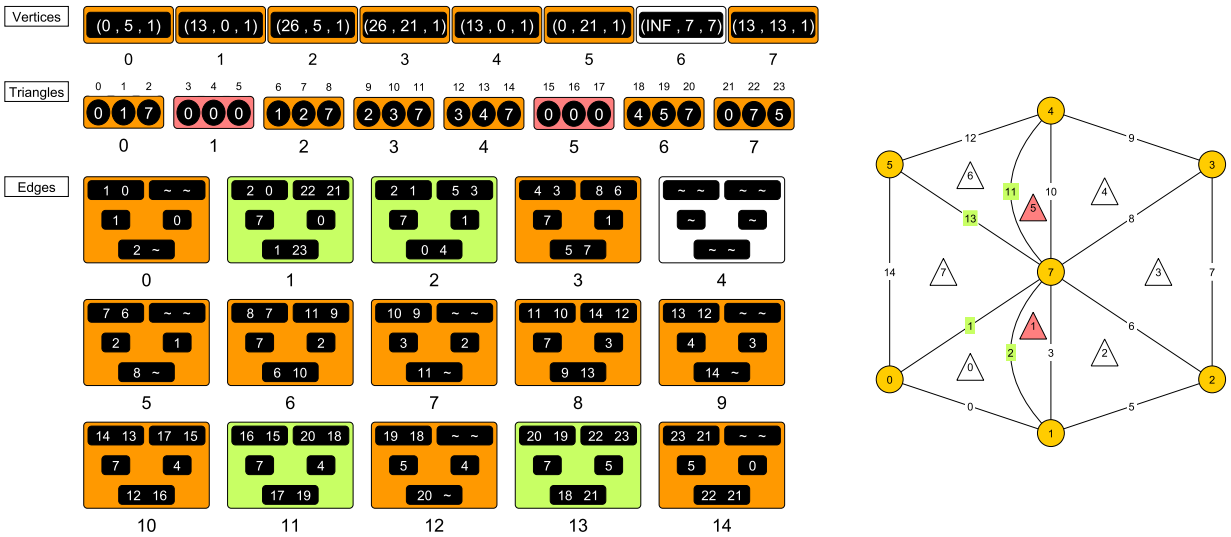


Figure C.7: Reading the vertex ID stored on the deleted vertex, each edge repairs its respective inconsistency, and now they refer to vertex 7.

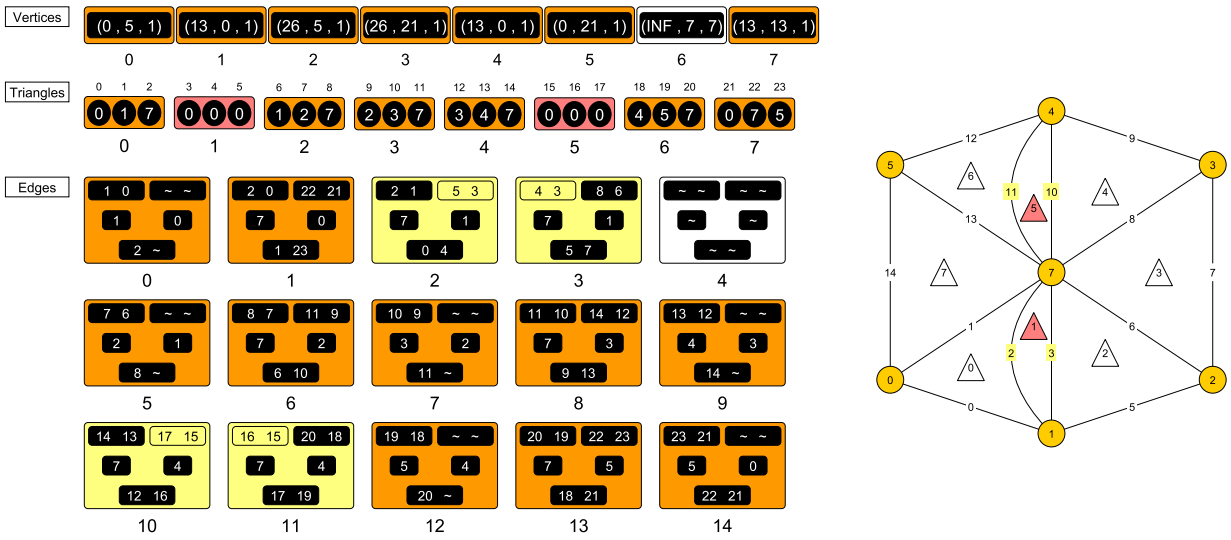


Figure C.8: After that, we identify the edges that have references to a deleted triangle (edges 2 and 3 have a reference to triangle 1 and edges 10 and 11 have a reference to triangle 5). This means that edges that refers to the same triangle, virtually holds the same data and have to be merged because of the edge-collapse.

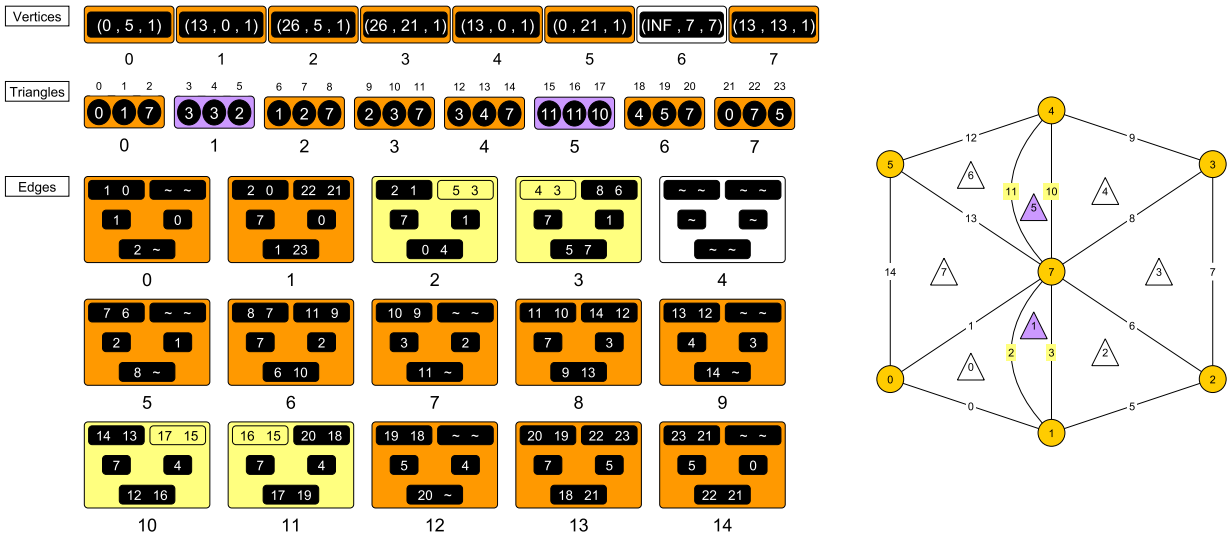


Figure C.9: Each of these edges writes its ID in the space of the deleted triangle that they are referencing (edges 2 and 3 writes they ID on triangle 1, and edges 10 and 11 writes they ID on triangle 5), to perform a consistency repair step in the future.

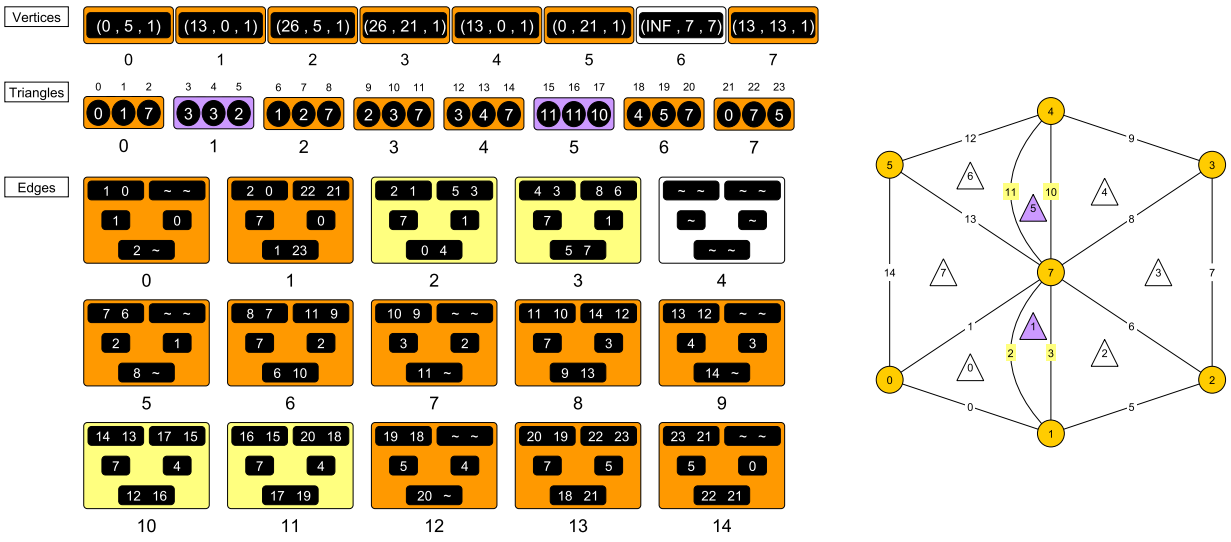


Figure C.10: Traveling all triangles in parallel, we identify the triangles that have 2 identical vertex IDs that are distinct from zero (triangles 1 and 5). This means that the triangle holds the IDs of the edges that have to be merged. This corresponds to the step explained in Section 4.6.4.

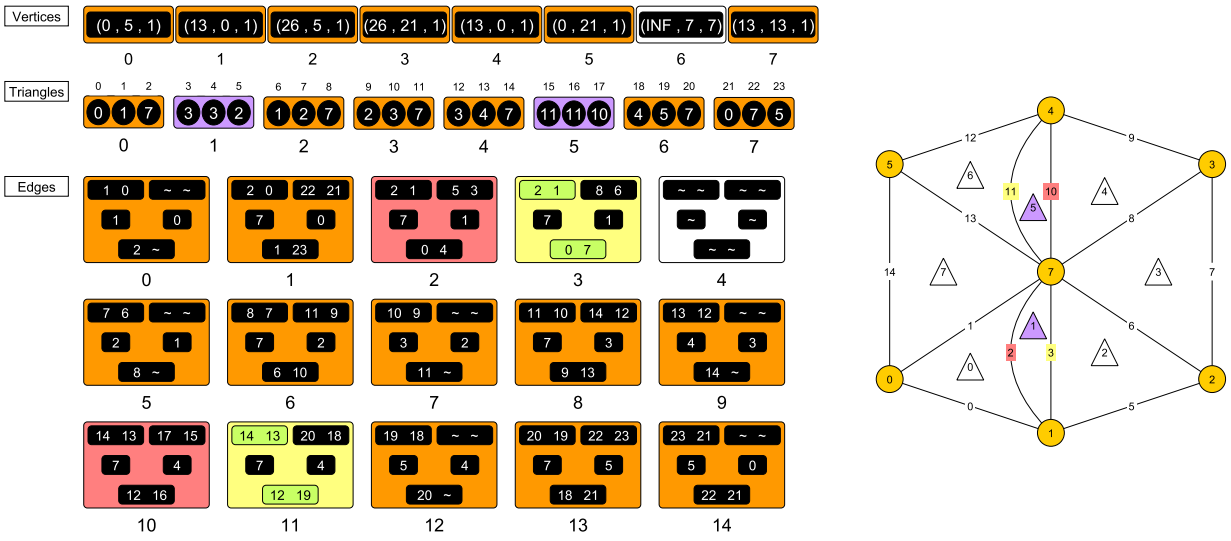


Figure C.11: The data of the edge with the lowest ID is transferred to the other edge in conflict. It transfers the references to the triangle an opposite vertex that they do not have in common.

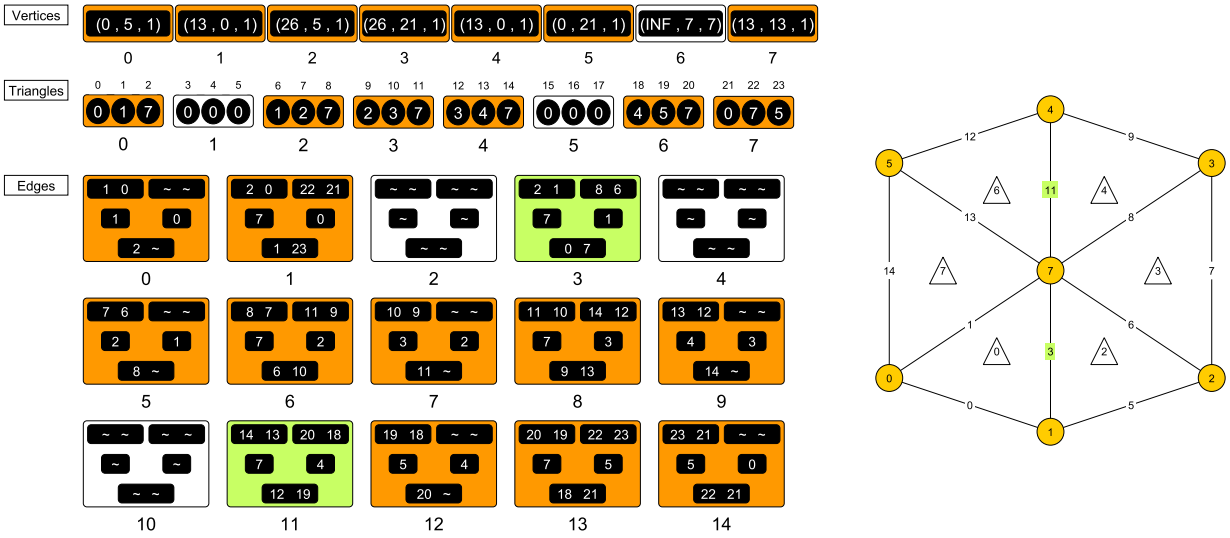


Figure C.12: The triangles previously identified are deleted. The edges that transferred its data are deleted as well.

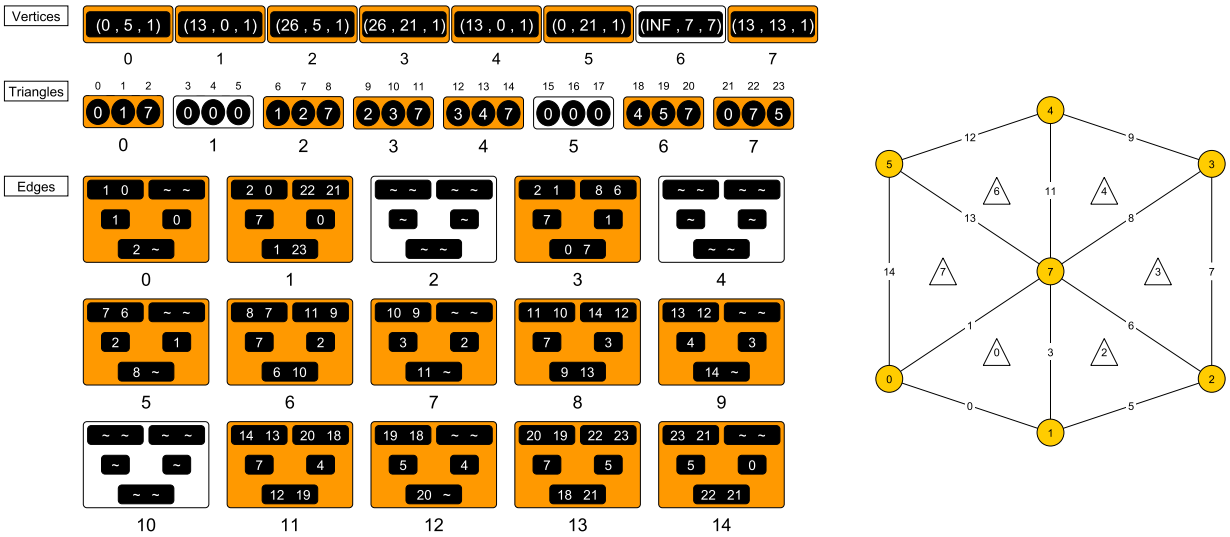


Figure C.13: This concludes the parallel edge-collapse algorithm phase. The elements in white correspond to the deleted elements of this procedure and are no longer referenced by other elements on the mesh.

Appendix D

Vertices Array Rearrange Example

In the following Figures, we show an example of how the vertices rearrange step, explained in Section 4.7.1, is done. Additionally, we show how we manage to repair the inconsistent references to vertices in the triangles and edges array, due to the *vertex-swap* procedure.

For this example, we will use the mesh shown in Figure D.1, where two edge-collapses have been performed, and the current status of the data structure can be seen in Figure D.2. For simplicity, in the subsequent Figures we show a simplified version of the Cleap data structure, that only shows the vertices references. The other components are not affected or modified by this procedure, and thus are omitted.

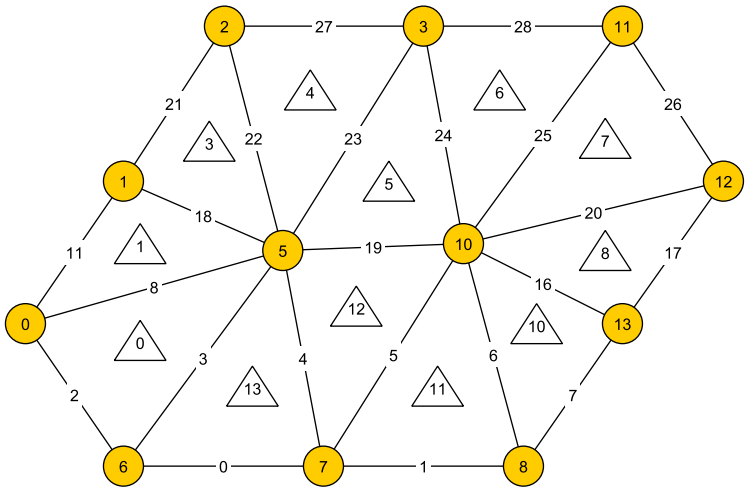


Figure D.1: A mesh where two edge-collapses have been performed. Originally it had 14 Vertices, 16 Triangles and 29 edges. Currently has 12 Vertices, 12 Triangles and 23 edges.

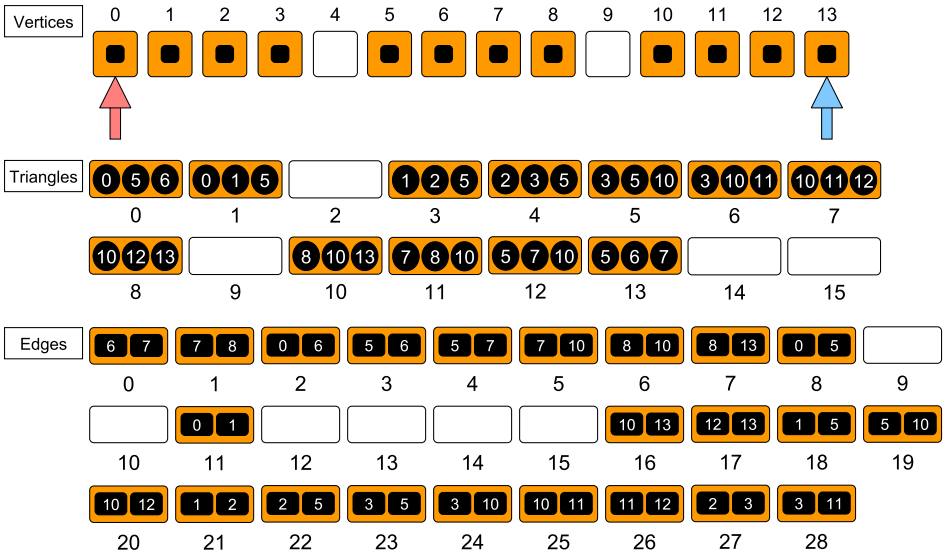


Figure D.2: Current state of the mesh shown on Figure D.1. To begin the rearrange process, the red pointer is initialized on the left side of the array, and the blue pointer on the right side.

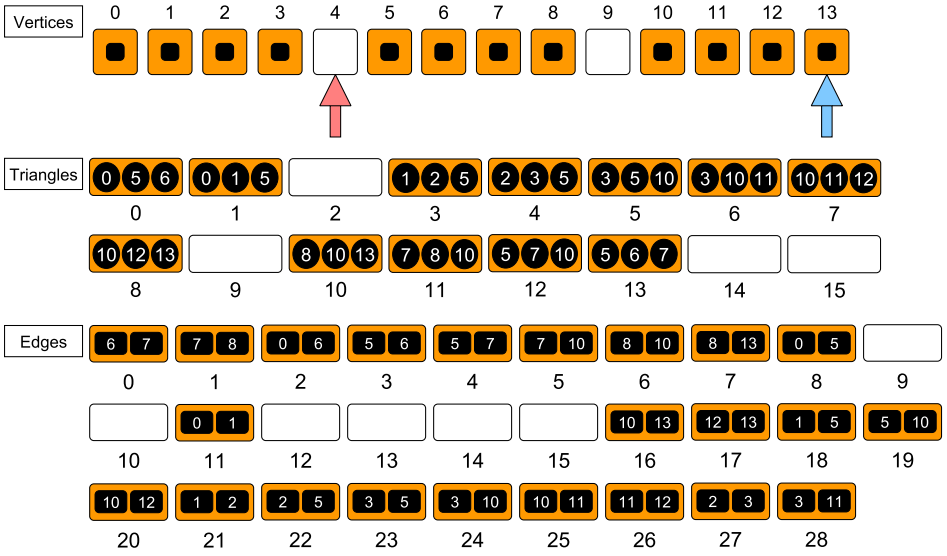


Figure D.3: The red pointer moves to the right until it finds a empty space. The blue pointer moves to the left until it finds a occupied space.

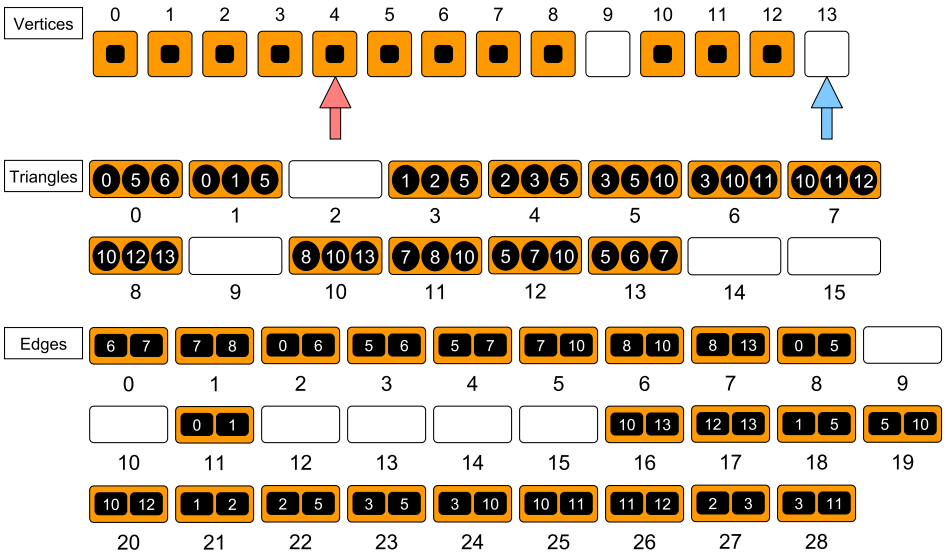


Figure D.4: The data referenced by the blue pointer moves to the empty space referenced by the red pointer.

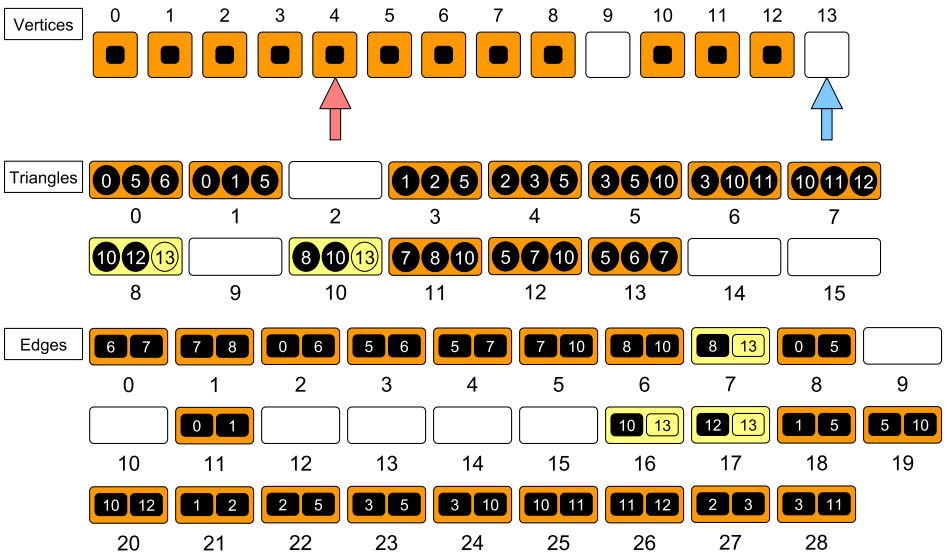


Figure D.5: In parallel, we identify the triangles and edges who hold a reference to the space previously occupied by the swapped vertex (triangles 8 and 10 and edges 7, 16 and 17 holds a reference to vertex 13).

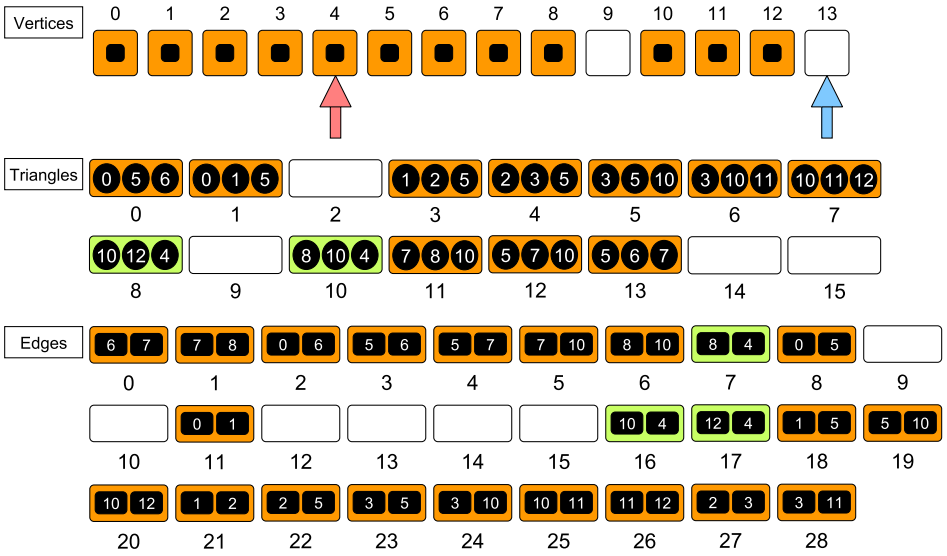


Figure D.6: Then we replace this incorrect reference for the correct one (index 13 to index 4).

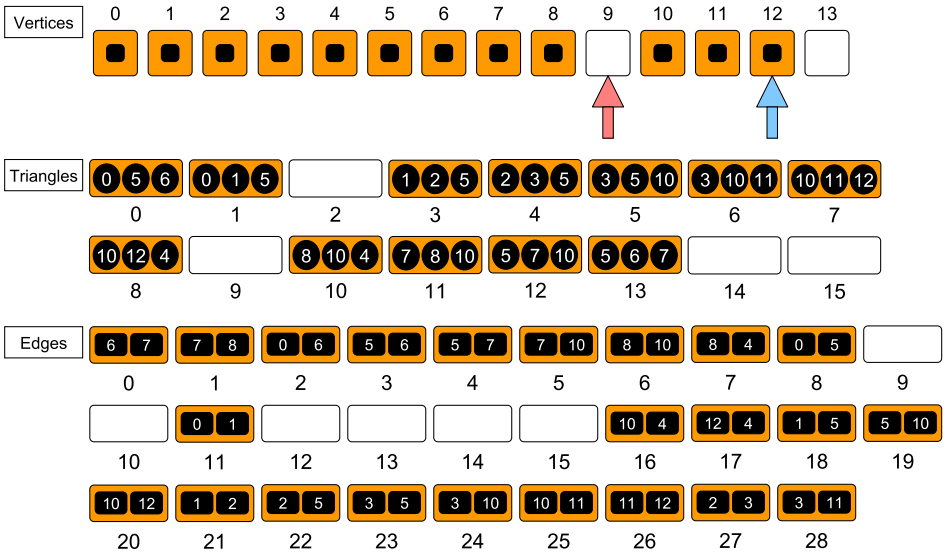


Figure D.7: The red pointer moves to the next empty space. The blue pointer moves to the next occupied space.

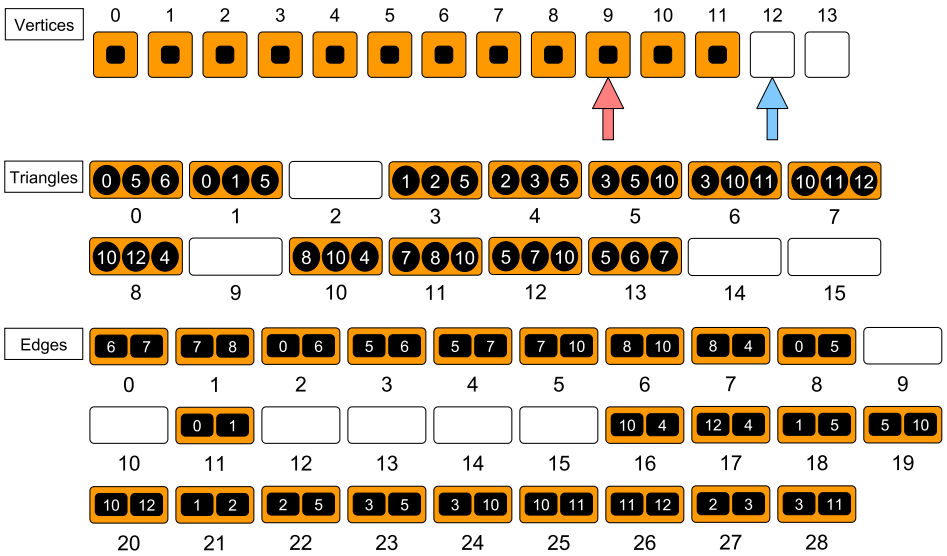


Figure D.8: The data referenced by the blue pointer moves to the empty space referenced by the red pointer.

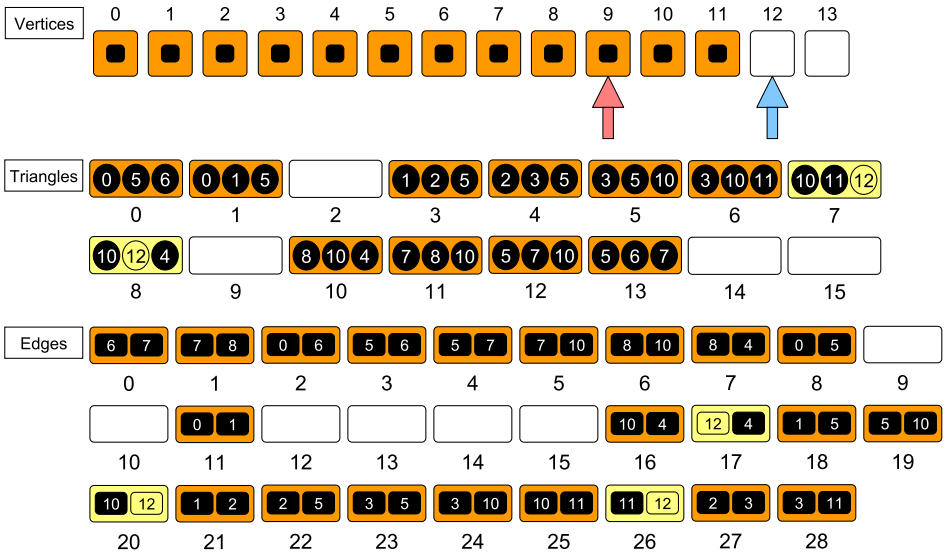


Figure D.9: In parallel, we identify the triangles and edges who hold a reference to the space previously occupied by the swapped vertex (triangles 7 and 8 and edges 17, 20 and 26 holds a reference to vertex 12).

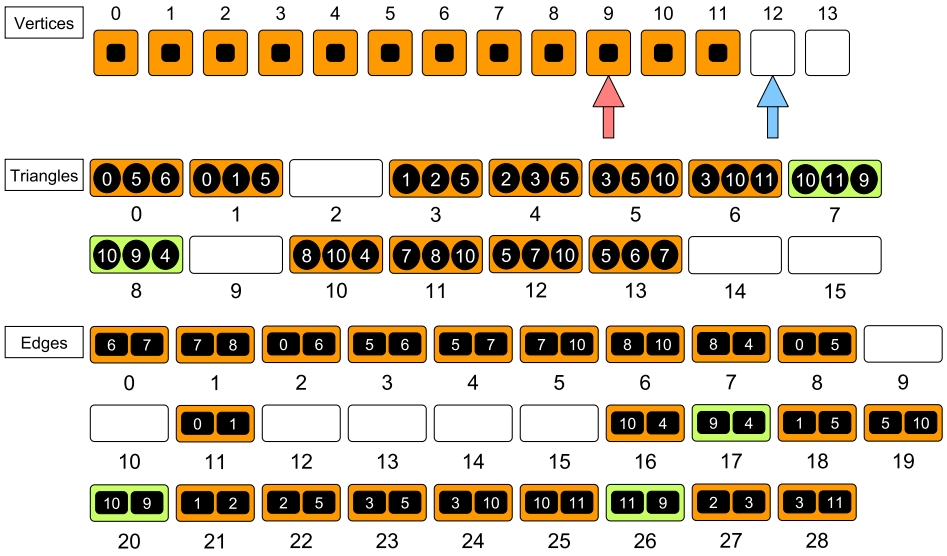


Figure D.10: Then we replace this incorrect reference for the correct one (index 12 to index 9).

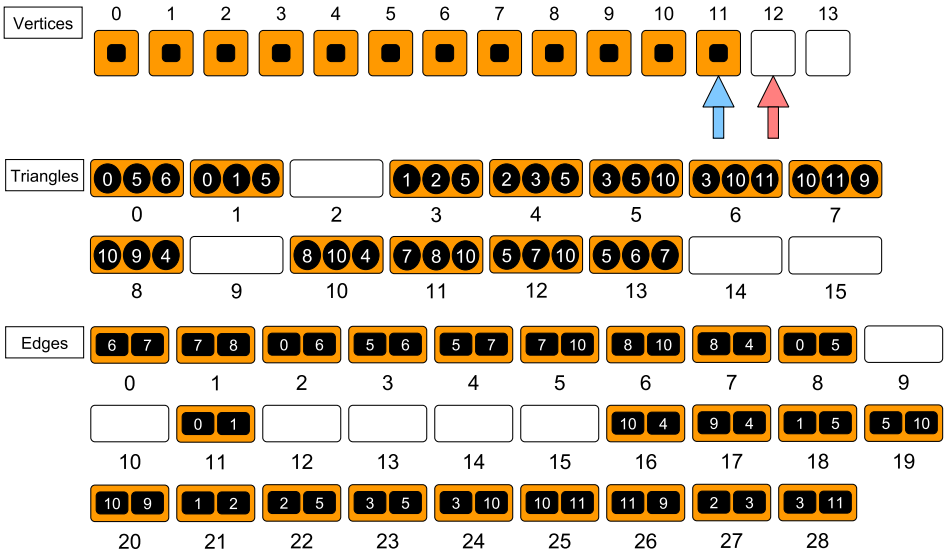


Figure D.11: When the pointers cross, the vertices array is rearranged and this procedure is finished.