MAURÍCIO GOUVÊA GRUPPI

Methods for Avoiding Round-off Errors on 2D and 3D Geometric Simplification

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*

VIÇOSA
MINAS GERAIS - BRASIL
2016

MAURÍCIO GOUVÊA GRUPPI

METHODS FOR AVOIDING ROUND-OFF ERRORS ON 2D AND 3D GEOMETRIC SIMPLIFICATION

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 23 de novembro de 2016.

Levi Henrique Santana de Lélis

Marcelo Bernardes Vieira

Marcus Vinicius Alvim Andrade
(Orientador)

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
G892m
2016

Gruppi, Maurício Gouvêa, 1991-
Methods for avoiding round-off errors on 2D and 3D geometric simplification / Maurício Gouvêa Gruppi. – Viçosa, MG, 2016.
xi, 63f. : il. (algumas color.) ; 29 cm.

Inclui apêndices.
Orientador: Marcus Vinícius Alvim Andrade.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.55-58.

1. Algoritmos. 2. Programação paralela (Computação). 3. Geometria Computacional. 4. Números racionais. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.1

"One day I will find the right
words, and they will be simple."

Jack Kerouac

Agradecimentos

Gostaria de agradecer, primeiramente, ao professor Marcus Vinícius por toda orientação concedida a mim desde a primeira vez em que estive envolvido com pesquisa, ainda na graduação, e por todo o apoio durante o mestrado. Agradeço também ao professor Salles por sua indispensável contribuição a este trabalho.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro a meu projeto de mestrado.

Acima de tudo, sou grato à minha família por ser meu ponto de apoio nos momentos mais conturbados. Em especial ao meu pai, José Virgilio, sem o qual eu jamais teria chegado aqui.

Contents

List of Tables

List of Figures

Resumo

GRUPPI, Maurício Gouvêa, M.Sc., Universidade Federal de Viçosa, novembro de 2016. **Methods for Avoiding Round-off Errors on 2D and 3D Geometric Simplification**. Orientador: Marcus Vinícius Alvim Andrade.

Neste trabalho avaliou-se a ocorrência de erros de arredondamento decorrentes do uso de aritmética de ponto-flutuante em algoritmos de simplificação geométrica 2D e 3D. Erros de arredondamento podem fazer com que algoritmos produzam resultados topologicamente inconsistentes, isto é, resultados que violam alguma característica crucial do modelo original. Foram mostradas situações em que inconsistências ocorrem, mesmo em algoritmos projetados para evitá-las. Visando contornar este problema, dois métodos foram propostos: *EPLSimp*, para simplificação de linhas, e *UGSimp* para simplificação de malhas triangulares. Em ambos os métodos, testes preemptivos para verificação de inconsistência topológica foram realizados utilizando-se números racionais de precisão múltipla, em vez de números de ponto-flutuante. O uso de números racionais não gera erros de arredondamento, entretanto o uso de precisão múltipla implica em um aumento no tempo de execução dos algoritmos. Para compensar esta redução de desempenho, os algoritmos foram implementados com o paradigma de computação paralela. Como resultado, temos dois métodos de simplificação isentos de erros de arredondamento por ponto-flutuante. Testes mostraram um ganho considerável no tempo de execução com as implementações paralelas.

Abstract

GRUPPI, Maurício Gouvêa, M.Sc., Universidade Federal de Viçosa, November, 2016. **Methods for Avoiding Round-off Errors on 2D and 3D Geometric Simplification**. Adviser: Marcus Vinícius Alvim Andrade.

In this work, we evaluated the occurrence of round-off errors on floating-point arithmetic for the problem of 2D and 3D geometric simplification. Round-off errors may lead algorithms to produce topologically inconsistent results, that is, results that fail to preserve crucial features of the original model. Some algorithms are designed to avoid such inconsistencies, however, they are usually implemented with floating-point arithmetic. Even these algorithms may fail to output topologically consistent results due to round-off errors. In order to overcome this issue, two methods were proposed: *EPLSimp* for polyline simplification, and *UGSimp*, for triangular mesh simplification. On both methods, preemptive tests are carried out to detect and prevent topological inconsistencies. Such tests use multiple precision rational numbers instead of floating-point numbers. The use of rational numbers does not present round-off errors. Nevertheless, it causes an increase on the execution time of the algorithms. To compensate for this performance loss, both algorithms were implemented using a parallel computing paradigm. As result, the methods presented do not output topologically inconsistent models. Tests have shown a considerable performance gain with parallel implementations of the proposed approaches.

1 Introduction

Often in computer graphics, practitioners are faced with a trade-off between data complexity and processing performance. Due to the advances in spatial and geometric data collection technologies, the volume of data tends to grow into large sets of information [2, 3]. Having bigger data sets adds complexity to the input of geometric algorithms, increasing the time needed for processing. Sometimes this excess of information is redundant, affecting little, or not affecting at all, the outcome of an algorithm. For that reason, datasets can be subjected to reduction of level of detail. Simplification plays an important role on reducing level of detail for the goals of performance and visualization.

At different levels of scale, a map can present different amount of information, so that the information shown is always clear and consistent to the viewer. A map of a city may show roads, intersections, and buildings with precision. However, if the same map is scaled up (i.e. *zoomed out*) to show an entire county or province, presenting all this information with the same level of detail and clarity may not be possible. Features may degenerate, such as polygons being presented as lines or points, and points that are close to each other may become clustered into a single point on the scaled map. On a 3D scene, objects far from the viewer usually need to be drawn with less detail than objects that are close, if the viewer moves away from an object, its level of detail can be reduced. This can reduce the computational cost of drawing the scene. An automatic method can work on such situations *on demand*, that is, simplifying features when necessary.

It is still difficult to quantitatively determine what is the best simplification for an object. Consider an input model map that consists of vertices and edges, if one desires to simplify it in order to reduce the number of vertices by half, there are many possible solutions. For instance, vertices for removal can be chosen at random, or there may be a certain metric to rank vertices before removing them. The latter is certainly the most commonly used strategy. Even though multiple ranking metrics exist, there is no absolute consensus of which one is the best. Each metric is designed for a specific purpose, such as to preserve the original shape of a map, or by preserving visibility on a 3D terrain. Therefore, simplification is solved by heuristic techniques that estimate reasonably good results. Whatever method is selected, simplification always introduce error to the model, then heuristics are used in order to evaluate this error according to a certain metric, such as displacement of effective area, Hausdorff distances or deviation of normal vectors [4].

Although the concept of best solution for simplification is not well defined, there are constraints that can be applied in order to determine whether a solution is valid or not. During simplification, changes on the model can introduce self-intersections,

or intersections with other elements. As a consequence, the simplified model presents inconsistencies: it does not correspond to its original representation. It is important that simplified models maintain consistency in order to be a satisfactory representation of the real world. Furthermore, geometric algorithms, such methods for *point location* may not work properly on objects containing self-intersection [5].

1.1 Polyline Simplification

A polyline, or polygonal chain, is a curve defined by a totally ordered set of vertices on which adjacent vertices are connected by edges. It can be used to represent maps and geometric features. Polyline simplification is a form of map generalization which consists of reducing the number of vertices on the input curve. Given a polyline P with vertices $V(P)$, a simplification of P is P_s with $V(P_s) \subset V(P)$.

The process of creating $V(P_s)$ can be constructive, creating the simplified line by adding vertices of the original line to it. Ramer-Douglas-Peucker's Algorithm (RDP) [6] [7] is one of the classic algorithms for map generalization that performs simplification in a constructive manner. The algorithm starts with a straight line containing only the endpoints (i.e., the first and last vertices) of $V(P)$. Then, it starts adding vertices to $V(P_s)$ if they are, at least, at a distance ϵ from the current curve. The value ϵ is called *distance dimension* and is defined by the user. The intuition about distance dimension is that points that lie far from the curve tend to contribute more to the curve's shape. Whereas points that are close to it, or points that are collinear, contribute little to the curve's model. The algorithm constructs the simplified line by adding vertices until a stop condition is met, which is usually when there are no more vertices farther than ϵ from the line.

Alternatively, $V(P_s)$ can be obtained by an elimination algorithm, such as Visvalingam-Whyatt's (VW) [8]. The simplified curve is determined by removing vertices from $V(P)$. Vertices are ranked by their *effective area* and the vertex with lowest effective area is removed. VW's method is discussed further on Chapters 2 and 3.

One can say that any subset of $V(P)$ is a valid simplification of the curve P. However, when topological relationships between elements are modified, the resulting model may contain topological inconsistencies. Such changes may be the creation of intersections or change of *sidedness* between features. Consider a polyline t with starting point a and endpoint b, and a rectangle that is placed on the right side of the polyline. If t is subjected to a simplification, the changes on t's topology can cause the rectangle to now be on the left side. As a practical example, t can represent a road and the rectangle can represent a building on a map. In this case, the relationship between t and the building was changed, and the simplified map is no longer consistent with the real world. Preventive verification can be used to secure topological consistency.

1.2 Triangular Mesh Simplification

Three-dimensional objects are usually represented as a collection of polygons. In most cases such polygons are triangles. A vector representation of a triangular mesh contains, at least, a list of vertex coordinates and a list of face incidences. The edges are not necessarily given, but they can be deduced from the first two lists.

Reducing the level of detail of a triangular mesh requires the removal of one or more of its simplices, such as vertices, edges and faces. Removing an entire face requires a post-processing step to fill up the hole created. Removing a vertex will make its faces degenerate, since they will have only two vertices. In [9] this problem was solved by removing degenerate faces and applying a re-triangulation process to the affected region. The removal of edges also require an extra step for fixing the faces that become degenerate. The edge collapse, or edge contraction, approaches consist of removing edges by merging two vertices together [10]. This process is illustrated in Figure 1.1 and described as follows: an edge $e = (v_1, v_2)$ (in red) is removed by merging its endpoints v_1 and v_2 into a resulting vertex v_3. The set of faces adjacent to v_3 is $F(v_3) = F(v_1) \cup F(v_2)$. The placement of v_3 can be simply determined by one of e's endpoints, or by the midpoint between v_1 and v_2. Alternatively, an optimization function can be used to determine the point on which the collapse would introduce the lowest error to the overall mesh.

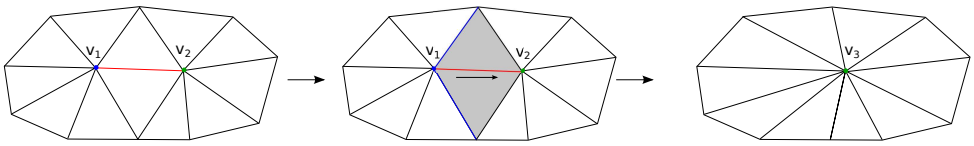


Figure 1.1 – Example of edge collapse. Faces in gray color become degenerate and are deleted. Vertex v_1 and v_2 are merged into v_3.

In addition to the reduction operation, it is essential to define a metric to select the simplices to be removed. In the case of edge collapse operations, this metric will define edges that are going to be contracted first. A naive approach would be to select the edge with shortest length, which seems reasonable since short edges tend to induce less change in the overall topology. A more efficient metric takes into consideration the displacement caused to faces when removing an edge. It was presented by Garland [11] and it is discussed on Chapter 4.

The edge collapse process can lead to topological inconsistencies on the mesh. It can cause faces to intersect or reverse their orientation by rotating their normal vector by 180°. Therefore, an extra step is required to guarantee topologically consistent results, that is, preemptive tests or mesh repairing processes need to be performed so that no intersection is created as a result of simplification.

1.3 Round-off Errors on Floating-Point Arithmetic

In order to represent a non-integer number on a computer system, it is necessary to convert its infinite representation into a finite precision number. This conversion process is known as discretization. It consists of shortening a number's length by rounding or discarding the least relevant digits in order to fit it into a finite amount of *bits*. A real number is transformed into a floating-point number by a *rounding function*, which is defined by the following mapping [12]:

$$\square : \mathbb{R} \to \mathbb{F}$$

Where $\mathbb{R}$ and $\mathbb{F}$ are, respectively, the real numbers and the floating-point numbers sets. The *rounding function* $\square$ maps a real number x to a number $\square x \in \mathbb{F}$. Common rounding functions are: *Round to nearest*, which rounds x to the nearest value in $\mathbb{F}$; and *Truncation*, which discards the least significant digits from x. In order to be practicable, rounding functions must present two properties:

- *Projectivity*: $\square x = x$ for all $x \in \mathbb{F}$;
- *Monotonicity*: $x \leq y \implies \square x \leq \square y$ for all $x, y \in \mathbb{R}$.

The *round-off error* of x is defined as the difference between the rounded value $\square x$ and the exact number x. This error is innate to the use of finite precision representation such as floating-point numbers. It may cause numerous problems to computations. One of the problems caused by round-off errors is the breaking of associative property of the addition. It is known that the following equivalence holds true for $a, b, c \in \mathbb{R}$:

$$a + b + c = (a + b) + c = a + (b + c)$$

However, there are values in $\mathbb{F}$ for which this property is not guaranteed. Taking $a = 10^{30}$, $b = -10^{30}$ and $c = 1$, using finite precision representation we have that:

$$(a + b) + c = 1$$

$$a + (b + c) = 0$$

That is, the associative property of the addition was not kept. This occurs because $(a + b) = 0$ on the first equation. However, $(b + c)$ is a *large negative* number that gets rounded to -10^{30}. Thus, due to a rounding error, the second equation is incorrect.

Round-off errors could have been the cause of several disasters involving highly precise computations. Some historical examples are: the Patriot Missile Failure, which caused the death of 28 people; the Ariane 5 Rocket explosion; and the sinking of the Sleipner offshore platform [13].

Geometric algorithms are directly affected by rounding errors, producing incorrect responses. In a study presented by Kettner [1], the effects of round-off errors are discussed for many geometric problems.

The first example of round-off errors presented by Kettner is the failure of the planar orientation problem. This problem consists of computing whether three points on a plane are collinear, form a left-turn, or a right-turn. This can be determined by the sign of the determinant:

$$orientation(p, q, r) = sign\left(\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}\right)$$

Positive, negative or zero indicate left-turn, right-turn, and collinear, respectively.

To verify the consistency of the method, Kettner executed the following experiment: first, three collinear points p, q and r are defined using floating point representation, e.g. $(0, 0)$, $(0.5, 0.5)$ and $(12, 12)$; then, p's coordinate is changed to an adjacent floating point coordinate; the planar orientation is calculated for the new setup. The process is repeated until all points on a predefined area are calculated (say all points on a 256×256 square area).

For the aforementioned example, where $q = (0.5, 0.5)$ and $r = (12, 12)$, it is expected that for every point p above line (q, r), the orientation of (p, q, r) is a left-turn. Similarly, for every p under (q, r), the orientation of (p, q, r) is a right-turn. Moreover, for p sufficiently close to (q, r), (p, q, r) is collinear.

Experiments have shown, however, that this does not work as one would expect. Figure 1.2 illustrates the result for the example mentioned above, the black line is an approximation of the real line through (q, r), the blue area contain points that were considered to make a left-turn with (q, r). Points in red were considered to make a right-turn with the line. Finally, yellow points were considered to be collinear with (q, r). There are yellow points surrounded by blue and red points. In addition, there are red (blue) points above (below) the line. These situations do not correspond to the expected behavior of the algorithm. Thus, in this example, the planar orientation algorithm failed to give the correct answer because the determinant computation resulted in on of the following errors:

- *perturbed zero*: a 0 result is misclassified as $-$ or $+$.
- *sign-inversion*: a $-$ result is misclassified as $+$ (and vice-versa).
- *rounding to zero*: a $+$ or $-$ result is misclassified as 0.

As a consequence, any algorithm that relies on planar orientation will also fail to provide correct outputs. Other examples of failure are given by Kettner: intersection detection and computation of the convex hull.

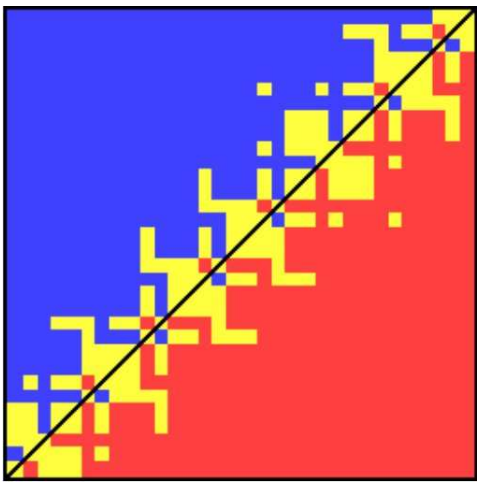


Figure 1.2 – Geometry of the planar orientation predicate for double precision floating point arithmetic. Blue points are considered to form a left-turn with line (q, r). Red points were considered to form a right-turn. Yellow points were collinear. The black line is an approximation to line (q, r).

On both polyline simplification and triangular mesh simplification, topological consistency is guaranteed by preemptive tests or repairing processes. Preemptive tests may include intersection and point location tests. An algorithm may fail to detect the occurrence of intersection between two lines if subjected to round-off errors. Consequently, the simplification algorithm may present results with topological inconsistencies.

Several techniques to overcome the problems of round-off errors have been presented . The simplest and most common approach is the *epsilon tolerance* method. Given a value ϵ defined by the user, two values x and y are considered equal if $|x - y| < \epsilon$. The biggest issue of this approach is that it is difficult to choose a value for ϵ that works properly for every input. Thus, the use of epsilon tolerance may fail in many situations [1].

Another commonly used strategy is the *snap rounding* [14], used to convert arbitrary precision arrangement of segments into finite precision representation. It consists of splitting the plane as a grid and rounding the position of every vertex to the center of the cell it lies inside. Vertices can be segment endpoints or segment intersections. Snap rounding can create inconsistencies and change the original topology [14]. Figure 1.3 shows how snap rounding can change the topology of segments.

A method for evaluating exact predicates was proposed by Shewchuk [15]. The Adaptive Precision Floating-Point technique consists of using the minimum level of precision needed to achieve exactness. Geometric predicates are usually obtained by the sign of a determinant, therefore, the actual value of determinant does not need to be exactly computed, as long as the sign of the approximated result is correct. To determine whether the sign of a determinant is correct, an approximation and

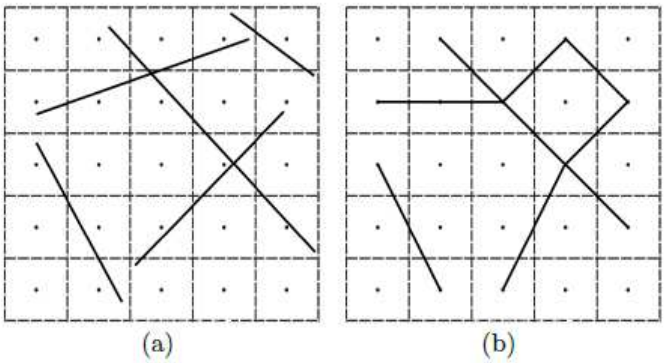


Figure 1.3 – (a) Original segments; (b) Segments after the process of snap rounding: endpoints and intersection points were rounded to the center of their cells.

error estimate are calculated. If the error is large enough to result in an incorrect sign, the values are recalculated using a higher level of precision. Although it is an efficient approach, this technique is limited to calculating geometric predicates and cannot solve every geometric problem.

The Exact Geometric Computation (EGC) model proposed by Li [16] uses algebraic models to represent mathematical objects and perform computations without errors. An algebraic number is the root of a univariate polynomial with integer coefficients. For example, the number $\sqrt{2}$ does not have a finite representation, but it can be represented exactly by the pair $(x^2 - 2, [1, 2])$, interpreted as the root of the polynomial $x^2 - 2$ that lies in the interval $[1, 2]$. This strategy is able to eliminate round-off errors but its performance is significantly low in terms of execution time.

A robust way of eliminating round-off errors is by using exact computations based on rational numbers with arbitrary precision [17]. The strategy is to represent values by pairs of integers that can grow as much as needed, which can be used perform arithmetic operations without round-off errors. However, managing arbitrary precision integer requires substantial computational effort which causes a considerable performance decline. In this work we intend to use rational numbers to get rid of round-off errors, combining it with high performance computing to make up for the performance loss.

1.4 Objectives

The general objective of this work are: to investigate the inconsistencies introduced by floating-point arithmetic on simplification algorithms; to develop methods to avoid such errors; to compare the quality of the results; finally to introduce strategies to compensate for the loss of performance. More specifically, we intended to:

- Find situations in which simplification algorithms create results with topological inconsistencies. Including those inconsistencies occurring due to floating-

point round-off errors.

- Develop and implement strategies to efficiently avoid such errors.
- Measure the performance of these methods, comparing the execution time against floating-point algorithms.
- Use parallel computing to improve performance of exact methods.
- Compare the quality of the output and execution time of the improved method against other known methods.

1.5 Obtained Results

Chapters 2, 3 and 4 describe the results obtained in this work. More specifically, Chapter 2 describes *TopoVW*, a method for fast and consistent line simplification. The results of this stage were published as a conference paper on the 17^{th} International Conference on Enterprise Information Systems (ICEIS) [18]. Chapter 3 shows how round-off errors affect *TopoVW*, and presents *EPLSimp*, a method for line simplification that uses multiple precision rational numbers to avoid round-off errors and parallel computing for performance gain. This phase resulted in another conference paper published and presented on the XVI Brazilian Symposium on Geoinformatics (Geoinfo) [19]. Chapter 4 investigates the occurrence of round-off errors on triangular mesh simplification which lead to topological inconsistent results. It also uses parallelism to increase performance. A conference article showing these results was submitted to the 19^{th} International Conference on Enterprise Information Systems (ICEIS). Finally, Chapter 5 presents a general conclusion of this work.

The problem introduced in Chapter 2 is the polyline simplification problem with some additional constraints. There are multiple polylines and control points on a map. The goal is to simplify the polylines to a certain level of detail without changing topological relationships between lines and control points. That is, the algorithm may not introduce inappropriate intersections and the sidedness of all features must be preserved. To achieve that, an extension of the VW algorithm, called *TopoVW*, was proposed. The contributions of this method are the use of *point-in-triangle* tests to determine whether or not the removal of a vertex would cause self-intersection or change of sidedness. Additionally, the proposed algorithm uses a uniform-grid [20] in order to restrict the number of points to be tested. The algorithm was able to perform simplification on multiple datasets efficiently and producing no topological inconsistencies. In addition, it performed faster than other algorithms due to the use of a uniform grid to reduce the number of *point-in-triangle* tests.

Chapter 3 presents specific situations on which preventive strategies, such as the one used by *TopoVW*, fail due to round-off errors. The *point-in-triangle* tests can

give false-negative results causing *TopoVW* to produce simplified lines with self-intersections and sidedness change. The contributions of this method are the use of a strategy to overcome the round-off error problem, we proposed *EPLSimp* which is presented as an extension to *TopoVW* and introduces the use of multiple precision rational numbers to avoid round-off errors on *point-in-triangle* tests. Additionally we developed a strategy to split the simplification problem into sub problems that can be processed in parallel, reducing the algorithm's execution time, since the use of multiple precision rational numbers introduces a considerable overhead to the processing.

Chapter 4 extends the idea of avoiding topological inconsistencies to the 3D perspective. A method for topologically consistent simplification of triangular meshes, *UGSimp* is presented. It uses the edge collapse approach to simplify a triangular mesh. The contribution of *UGSimp* are: performing preemptive tests to avoid self-intersection of simplices; instead of using a global criterion for edge selection, *UGSimp* divides the mesh using a uniform-grid and each cell of the grid has a queue of edges to be removed. This allows the parallel simplification of the mesh and increases the performance of the algorithm. Results have shown that the use of local criterion does not imply lower quality results.

Some authors do not consider topological inconsistencies caused by simplification as an issue. In [21], Rossignac states that some topology alterations do not affect the results visually. For example, simplifying an object to a point where it becomes disconnected. However, intersection is not mentioned in the report and may introduce problems in applications. Models used for 3D-printing should not have self-intersections. The existence of self-intersections on objects may affect point location algorithms. Moreover, changing orientation of faces can affect algorithms for hidden surface determination, therefore "hiding" parts of the mesh that are actually visible.

Additional tests were executed to further evaluate the performance of *UGSimp*. The results show comparisons between *UGSimp* and the GPU method when simplifying the datasets to 10%, 25%, 50% and 90% level of detail. The quality of such results was also measured and color-scale models were used to show the comparison between *UGSimp* and the GPU method [22]. Furthermore, tests were performed using different resolutions for the uniform-grid. The choice for a good resolution relies on the balance between the amount of parallelism and the amount of edges entirely inside a cell. A higher grid resolution contains a higher number of independent cells, but the number of edges that lie entirely inside a single cell is lower than a low-resolution grid. The best performance of *UGSimp* was using a resolution of 512 cells.

2 An Efficient and Topologically Correct Map Generalization Heuristic [1]

Abstract

We present TopoVW, an efficient heuristic for map simplification that deals with a variation of the generalization problem where the idea is to simplify the polylines of a map without changing the topological relationships between these polylines or between the lines and control points. This process is important for maintaining clarity of cartographic data, avoiding situations such as high density of map features, and inappropriate intersections. In practice, high density of features may be represented by cities condensed into a small space on the map, inappropriate intersections may produce non-existing intersections between roads, rivers, and buildings. TopoVW is a strategy based on the Visvalingam-Whyatt algorithm to create simplified geometries with shapes similar to the original map, preserving topological consistency between features in the output. It uses a point ranking strategy, in which line points are ranked by their effective area, a metric that determines the impact a point will cause to the geometry if removed from the line. Points with inferior effective area are eliminated from the original line. The method was able to process a map with 4 million line points and 10 million control points in less than 2 minutes on a Intel® Core™ 2 Duo processor.

2.1 Introduction

Map generalization, or cartographic generalization, is a process that aims to simplify the way cartographic data is represented by removing information that is not relevant to the viewer, while preserving crucial features on the map. Generalization is an innate trait of every geographical data, maps are generalized representations of reality, and the more generalized is a map, the more remote it becomes from the real world [23]. The output of this process is a map with less complex data, focusing on desirable properties of the original map. The biggest concern in this process is to find the balance between simplification and reality .

[1] In this chapter we presented a method for 2D line simplification that avoids topological inconsistency and uses floating-point arithmetic. *An Efficient and Topologically Correct Map Generalization Heuristic*, published on ICEIS 2015 (17th International Conference on Enterprise Information Systems). [18]

Although generalization is a task that has been effortlessly done by humans, it is still hard to automate. One of the main reasons is that information may change during varying of scale, and relationships between features in the map have to be maintained. Another reason is that "a map is a complex mix of metric and topological patterns reflecting individual and interdependent gestaltic properties. Understanding these forms, and conveying salient characteristics requires both cartographic and geographic knowledge" [24, p.7].

Situations requiring generalization ideally occur due to failure of the map to meet its goals, that is, during the simplification process, the map is unsuccessful in preserving clarity of the content, at a given scale, for the intended viewer [25]. Some conditions that occur due to scale reduction may determine the need of generalization on a map, such as high feature density and imperceptibility, when a feature may be invisible, enlarged, or converted to the appearance of a different feature, for instance, when a polygon is condensed so much it acquires the shape of a line or a point.

2.1.1 Topological Consistency

Preserving consistency of data is crucial on both digital and analogue cartography. As result of generalization processes, both geometries and topological relationships of map features are expected to be changed [26]. This is undesirable since it may result in inappropriate maps, containing inconsistent relationship between features, such as roads (lines) intersecting buildings (polygons, for instance), buildings intersecting other buildings, or overlap between rivers and buildings. Therefore, simplifications algorithms must handle such situations, if they occur. Sidedness is another aspect to be preserved. Maintaining sidedness means that, after the process of generalization, features which were simplified will not be on a different side of other features (such as control points).

2.1.2 Polyline Simplification

One way to perform cartographic generalization is by applying polyline simplification to reduce the complexity of such features in a map. A polyline is a series of line segments defined by a sequence of n points (v_1, v_2, ..., v_n) called vertices, each two neighbouring segments in the series share a common endpoint, namely, a polyline describes a curve starting on a point v_1, passing through every vertex on the sequence until v_n. Figure 2.1 shows an example input containing three polylines L_1, L_2 and L_3, and also a feature point P. To apply generalization to a polyline implies simplifying its representation by removing points from its sequence, that is, to attempt to represent the same curve with fewer vertices, the process of simplification is illustrated by Figure 2.2. This, however, can cause inconsistencies of topological

relationships between polylines and points. For instance, applying simplification on a county dataset may change its boundaries in such a way that a point representing a city may fall in the wrong county, i.e., during the process of simplification it was not taken into consideration that it was mandatory for the point representing the city to be inside the county polygon by the end of the generalization, that is, sidedness was not kept. Moreover, polyline simplifications can create inappropriate intersections between polylines as shown by Figure 2.3. Among the most well-known algorithms for performing polyline simplification are Douglas-Peucker, also known as Ramer-Douglas-Peucker (or RDP) [6, 7] and Visvalingam-Whyatt [8].

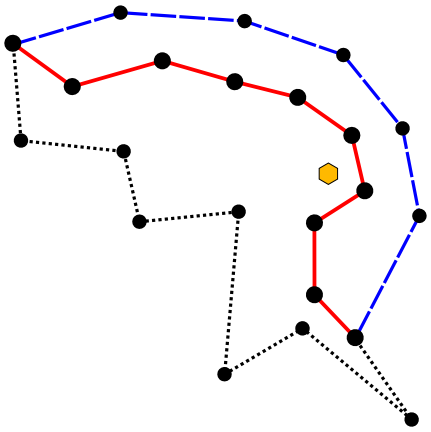


Figure 2.1 – Example input of three polylines and a control point.

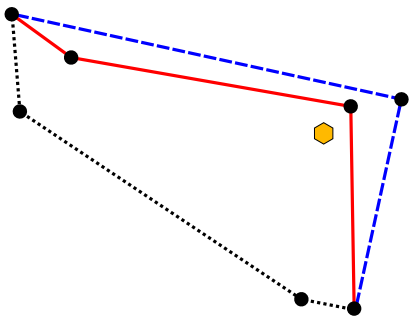


Figure 2.2 – Example of valid simplification output, note that topology consistency is preserved, no intersections were created and sidedness is maintained.

In this paper is presented a solution for the following variation of the generalization problem: given a set of polylines and control points, the goal is to simplify these polylines by removing some of their vertices (except endpoints) such that topological relationships between pairs of polylines and between polylines and control points are kept. In practice, polylines may represent boundaries of counties or states, and control points may represent cities within these states. As indicated by Estkowski et al. [27], this problem is difficult to solve, even approximately. This paper presents an extension of the line simplification method *Grid-Gen* [28] , in which a new heuristic based on Visvalingam-Whyatt algorithm [8] is included. The method uses a strategy of point ranking to determine the best points to be kept in the output line. Polyline

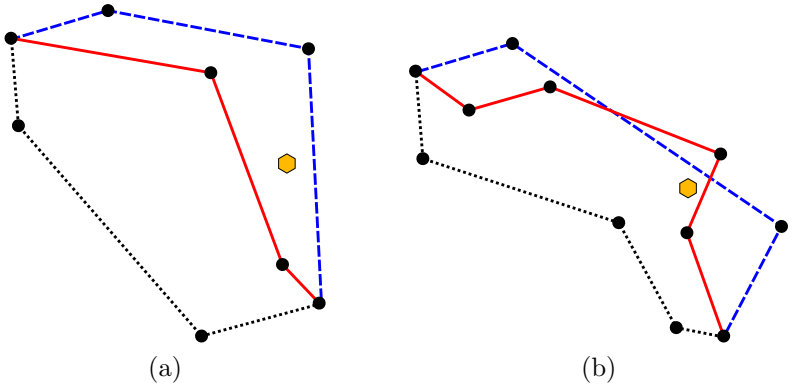


Figure 2.3 – (a) Invalid simplification output, feature point P at placed in the wrong side of line L_2 after the process; (b) Intersection created between lines L_1 and L_2. Note that this intersection did not exist prior to the simplification, therefore it is inappropriate.

points are ranked by its effective area, namely the area generated by a point and its neighbours, then a cut-off factor is applied for eliminating points from the ranked list, points with lower effective area values will produce less area displacement when eliminated from the line, thus causing less impact to the topology of the output [8].

2.2 Related Works

One alternative for performing map generalization uses a stochastic optimization technique [29]. It carries out operations such as displacement, size exaggeration, deletion and size reduction in order to resolve conflicts that result from map scale reduction. It uses a trial position approach, for each of n discrete polygonal objects is assigned k candidate trial positions that represent the original, displaced, size exaggerated, deleted and size reduced states of the object, which results in a total of k^n distinct map configurations. It is expected that some of these configurations will retain decreased level of map conflict. The goal is then to find the configuration with the lowest level of graphic conflict, which could be done by an exhaustive search, however, in practical situations this is not viable, i.e., for realistic values of n and k, an exhaustive search cannot provide a solution in efficient time. The method utilizes a simulated annealing strategy [30] to overcome the exhaustive search issues. The algorithm has been successful in reducing map conflict within reasonable time. Nevertheless, since it makes use of a limited number of k pre-defined trial positions, it may provide inappropriate solutions [29].

Ramer-Douglas-Peucker's algorithm has the issue of outputting polylines with inconsistent or incorrect topological relationships. An approach proposed by Saalfeld to polyline simplification using the Douglas-Peucker algorithm attempts to solve this problem by adding tests to the stopping conditions of the original method which guarantee the consistency and correctness of the resulting features [31]. Its

main idea is to generate polylines with Douglas-Peucker and try to refine it by adding points to the resulting curve (polyline) such that the curve does not present any topological inconsistency. However, that adding points to the curve may erase previous topological inconsistency and create new ones. It is also proved that the search for points to be added to the polyline can be restricted to just a part of the segment [31].

Although the previous method handles topological inconsistencies generated by the Douglas-Peucker algorithm it does not consider two topological configurations that may be undesirable: coincidence topology, which consists of the overlapping of two polylines or the overlapping of a feature point and a polyline, and the incidence topology, which concerns the incidence of a polyline point on another polyline p, but not on a vertex of p. Coincidence and incidence topologies are unwanted because they can generate redundancies on a map, such as slivers and gaps [32]. A method to handle these configurations work as an expansion of Saalfeld's method, by determining *essential vertices* and adding a proximity criterion to the simplification process, the method eliminates coincidence and incidence from the results.

Another Douglas-Peucker based approach proposed by Li et al. [33] intends to avoid topological inconsistencies as well as cracks on polygon shapes. Its strategy is based on detection-point identification, which are points lying within a minimum boundary rectangle (MBR) of the bounded face formed by a subpolyline and its corresponding simplifying segment. These detection-points are used for consistency verification of the simplification process. The detection-point identification process starts by getting the MBR $M_{i,j}$, of the bounded face formed by point $P_{i,j}$ and its corresponding simplifying line segment $e_{i,j}$. The next step is to identify features whose MBRs intersect $M_{i,j}$, then, from vertices of the exterior and interior boundaries of such features, identify those lying within $M_{i,j}$ and add such vertices to an output detection-point set R. An improvement to this process is made to avoid unnecessary points arising due to shared edges, that is, edges that belong to more than one line. However, the problem of polyline simplification does not generate cracks, since it works with multiple polygonal chains, and not with polygons which contain shared edges, therefore, the proposed method does not have to handle these particular situations.

A method proposed by Estkowski and Mitchell [27] performs map simplification of polygonal subdivisions by using a detour heuristic to prevent intersections of lines, which can result from another simplification method, such as Douglas-Peucker. It first calculates the simplification of an input S using the Douglas-Peucker algorithm, then it calculates all pairs of intersecting lines placing them in a separate list L. Thereafter, given two intersecting lines s and s' in the list L, each described by a pair of endpoints v_1, v_2 and v'_1, v'_2 respectively, the idea is to search a graph, called detour graph, for a shortest path connecting the endpoints of s (or s'). If this

path exists, the vertices are added to the original line in order to eliminate the intersection. Otherwise, it carries out a splitting process which consists of choosing a random vertex u and separating line s into two segments v_1u and uv_2, this is repeated until a solution is found within a certain error bound.

2.3 The Proposed Method

2.3.1 The Grid-Gen Heuristic

Given a set of control points C and an input map M composed of a set P of polylines, the heuristic simplifies M by iteractively processing each polyline independently. When a polyline is processed, *Grid-Gen* [28] iterates through all its interior points v_i (that is, the points excluding the endpoints) and removes v_i if this deletion would not change the topological relationships between the map's elements. This process is repeated until the number of points in the simplified map reaches a target value defined by the user or until the map cannot be further simplified without changing its topology.

To determine if the deletion of a polyline point v_i would change the map topology, *Grid-Gen* verifies if there is any control point or polyline point inside the triangle t whose vertices are v_i and its two adjacent points (i.e., v_{i-1} and v_{i+1}).

Figure 2.4 presents an example of the possible topological changes that may happen during the deletion of points. Notice that there is a control point x inside the triangle formed by polyline point a and its two adjacent points. If a polyline is simplified by removing a, then the topological relationship between the curve and x will change. Point b also cannot be removed since polyline point y is inside the triangle containing b as vertex and, thus, the deletion of b would change the topological relation between b's polyline and y's polyline (the two polylines would cross if b was removed). Therefore, neither a nor b could be removed from the current map. On the other hand, point c can be removed from the map without changing its topology since there is no control or polyline point inside the triangle generated by vertex c.

Algorithm 1 shows a pseudo-code for the method. Notice that, if no point is removed during one iteration, in the successive iterations no point will be removed either, because the map in the next iteration will not differ from the map in the previous one, therefore, the method can be terminated.

In some situations, Algorithm 1 may create maps with invalid topology. If a polyline p has coincident endpoints and the polygon (or *island*) defined by this polyline does not contain any control points or polylines in its interior, then applying Algorithm 1 may remove all points from p, thus, creating an invalid polygon.

In addition, if two polylines p_1 and p_2 have coincident endpoints, and the polygon generated by them does not contain control points or polylines in its interior,

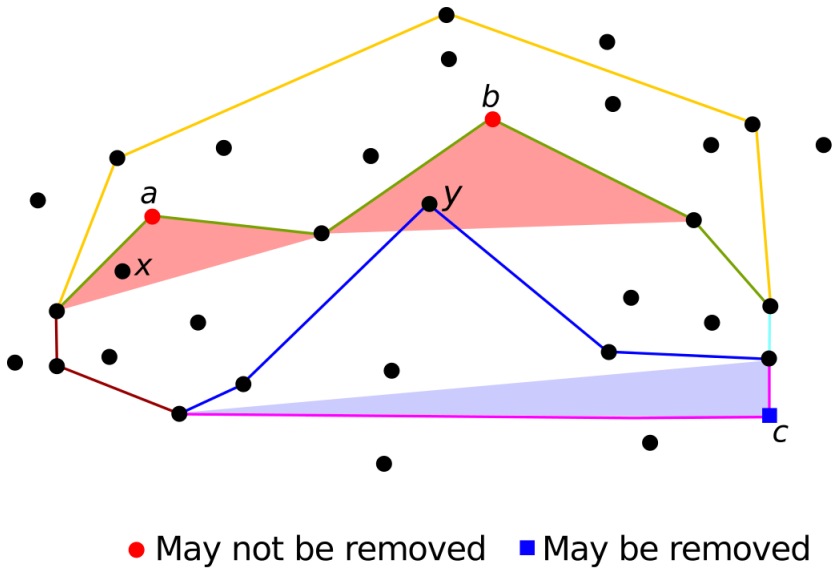


Figure 2.4 – Determining if the deletion of some points would change the map topology.

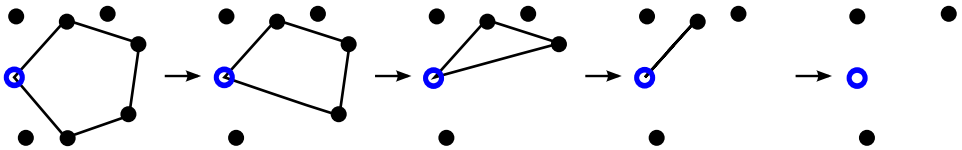


Figure 2.5 – Simplification of a polyline with coincident endpoints and no control points or other polyline in the interior of the polygon that it defines.

Algorithm 1 may remove all interior points from p_1 and p_2, creating two coincident line segments, as shown by Figure 2.6.

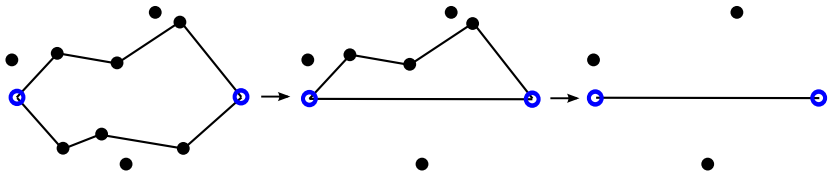


Figure 2.6 – Simplification of two polylines with coincident endpoints and no control points or polylines in the interior of the polygon defined by them.

To solve these two problems, *Grid-Gen* preprocess the input adding *dummy* control points that ensure that the heuristic would never simplify the polylines to an invalid state. If a polyline p has coincident endpoints, two dummy control points are added at an infinitesimal distance around one of the line segments that forms p. See an example in Figure 2.7. This ensures that one of these control points will be always in the interior of the polygon defined by p and, therefore, the heuristic will never remove all interior points of p. By using this strategy, Algorithm 1 will never generate invalid simplifications as it would in changes in the topological relationships between the polylines and the *dummy* points.

If an input polyline p has only two points, *Grid-Gen* also adds two *dummy* control points in an infinitesimal distance around p. Furthermore, if during the

simplification all the internal points of a polyline are removed, the dummy points are also added around the resulting polyline. This ensures that no simplification would create a polyline coincident to p. Figure 2.8 presents an example where all interior points of a polyline p are removed and, then, two *dummy* points are added to the map.

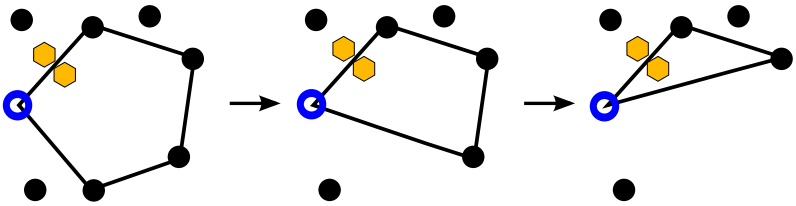


Figure 2.7 – Use of dummy points (hexagon in orange) to avoid invalid simplifications: note that, in this dataset, the process terminates after removing the second interior point of the polyline since a dummy control point is inside the triangle defined by the resulting polyline.

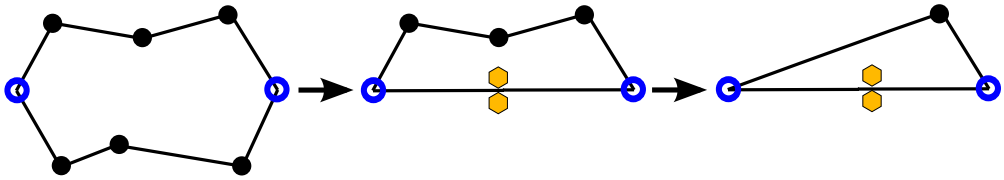


Figure 2.8 – Use of dummy control points (hexagon in orange) to avoid invalid simplifications.

Another concern is about the special case in which some line segments are collinear to the endpoints of a polyline. For instance, in Figure 2.9 there are no control points or polylines within the triangle v_{i-1}, v_i, v_{i+1} and, therefore, Algorithm 1 could remove vertex v_i, creating coinciding segments. This special case is treated by considering every point along the edges of the triangle v_{i-1}, v_i, v_{i+1} are inside triangle, thus not removing v_i.

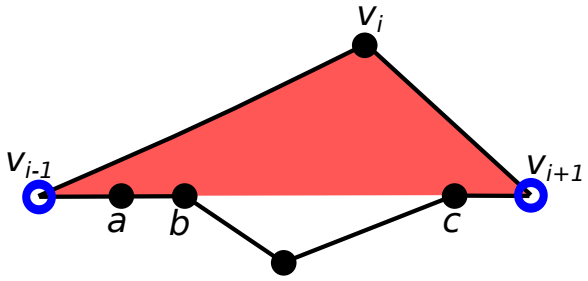


Figure 2.9 – Example of polylines with the same endpoints: if v_i is removed, the resulting polyline will have segments coinciding with segments of the other polyline. To solve that, it is considered that points on the edges of the triangle v_{i-1}, v_i, v_{i+1} (points a, b and c) are effectively inside the triangle.

Algorithm 1 Pseudo-code for the Grid-Gen algorithm.

1: $np2rem \leftarrow$ Number of points to remove
2: $npar \leftarrow 0$ //Number of points already removed from the map
3: **while** npar $<$ np2rem **do**
4: **for** each polyline L in the input map **do**
5: **for** each interior point v_i of L **do**
6: **if** colinear(v_{i-1}, v_i, v_{i+1}) OR $\nexists$ polyline point/control point in a triangle v_{i-1}, v_i, v_{i+1} **then**
7: $npar \leftarrow npar + 1$
8: Remove v_i from L
9: **if** npar $>$ np2rem **then** Stop the algorithm

The bottleneck of *Grid-Gen* is to detect if a polyline or control point lies inside a triangle. To speedup this step, a uniform grid [20] is used. More specifically, the idea is to create a $N \times M$ grid (where N and M are parameters defined by the user), superimposed over the map being simplified. Each cell c of the grid contains a list of all points (polyline and control points) inside it. Given a triangle t, only the points in the cells that intersect t need to be checked in order to verify if there is any point in t. If a polyline is simplified, the point removed from the polyline is also removed from the uniform grid. Figure 2.10 presents an example of an uniform grid superimposed on a map.

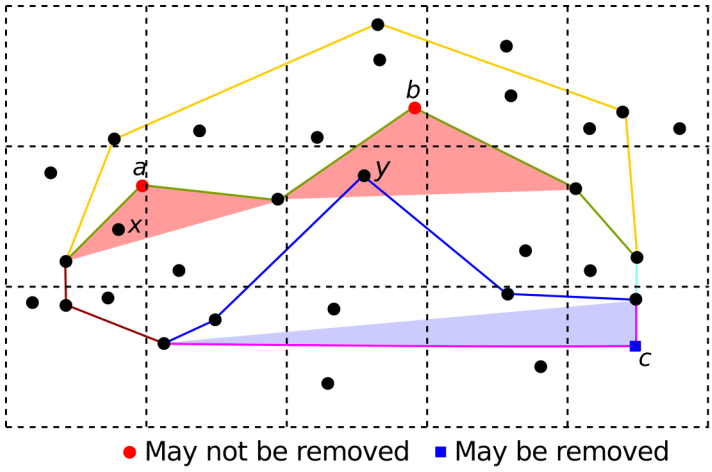


Figure 2.10 – Example of a 3×5 uniform grid superimposed over a map.

2.3.2 Extending Grid-Gen

As explained in section 2.3.1, *Grid-Gen* iteratively processes the input map removing interior points from the polylines that would not cause a topology inconsistency. Although this strategy can efficiently simplifies a map avoiding topological inconsistency, it does not try to keep the simplified map geometry similar to the original one.

The *TopoVW* is an extension of *Grid-Gen* that ranks the interior polyline points based on their "relevance" to the map shape and performs the map simplification iteratively removing the least "relevant" point. More precisely, the points are ranked using a strategy similar to Visvalingam-Whyatt's [8] algorithm, and a simplification process similar to *Grid-Gen* is performed on the map, processing the points in an order based on their rank.

2.3.2.1 Visvalingam-Whyatt Algorithm

A method for line generalization proposed by Visvalingam and Whyatt [8] uses the *effective area* of a point to define the point elimination sequence. This method is suitable for both minimal generalization and caricatural generalization. The *Effective area* of a point v_i is defined as the area of the triangle formed by v_i and its neighbors, that is, the triangle whose vertices are v_{i-1}, v_i and v_{i+1}. This represents the areal displacement created if point v_i is removed from the line. The algorithm for Visvalingam-Whyatt method is as follows (Algorithm 2):

Algorithm 2 Pseudo-code for the Visvalingam-Whyatt generalization method.

```
Data: Input line L as a list of points, separate list R of ranked points
Compute the effective area of each point on the line
Delete all points with zero area and store them in a separate list
loop
    Find the point with least effective area and call it current point.
    Delete the current point from the original list L and add it to the ranked list
  R with its effective area
    Recalculate the effective area of the two adjacent points
    if Size of L = 2 then Terminate
```

As seen in Algorithm 2, it removes points from an input line L and adds these points to a list R ranked by the effective area of each point. This can be used to only include the most important points in the resulting line, as determined by a cut-off factor. This method has shown better results than other simplification algorithms such as the ones proposed by Ramer-Douglas-Peucker [7], Roberge [34], and Dettori and Falcidieno [35].

2.3.2.2 The Method TopoVW

While the ranking point strategy tries to generate simplified maps similar to the original input data, the topological inconsistency detection strategy derived from *Grid-Gen* ensures that no topological error is introduced in the output map. The method *TopoVW* consists of integrating both strategies.

Given a polyline point v_i, the rank of v_i is defined based on the *effective area* of v_i. As shown by Visvalingam-Whyatt [8], points with higher effective areas are usually "more relevant" than points with smaller areas for preserving the original

shape and, thus, during the simplification step, points with smaller effective areas have higher priority for deletion. In order to preserve topological consistency, the simplifications step uses an approach similar to *Grid-Gen*'s for point deletion.

For efficiency purposes, *TopoVW* initially pre-processes the input data computing the *effective area* of the polyline points. These points are kept in a priority queue with priority based on the point's *effective areas*. Since the interior points of all polylines are kept in the same priority queue, *TopoVW* simplifies the polylines globally, that is, in different iterations points from different polylines may be removed. When a point v_i is deleted from the map, this may change only the *effective area* of its two adjacent points v_{i-1} and v_{i+1} (in L's polyline) and, thus, these two areas are recomputed and the new values are used to update the priority of v_{i-1} and v_{i+1} in the queue.

2.4 Experimental evaluation

TopoVW was tested on a computer with the following configuration: Intel® Core™ 2 Duo E7500 at 1596 MHz, 4GB of RAM, Seagate ST2000DM001-1CH1 7200 RPM SATA hard-drive, and Linux Mint 17 Qiana.

The tests were performed on the same datasets previously used by Magalhães et al. [28]: the first 5 datasets were used by the ACM GISCUP 2014 [36] organizers to evaluate the algorithms submitted to the GISCUP contest. Since datasets 1 and 2 were too small (they contained less than 2000 polyline points and less than 200 control points), test results for both datasets were not included in this paper.

The polygons in dataset 6 were obtained from *IBGE*'s (the Brazilian geography agency) website [37] and represents the Brazilian county divisions. Dataset 7 represents the United States continental county division and was obtained from the United States Census website [38]. The control points in datasets 6 and 7 were selected in random positions in the maps. Table 2.1 presents the number of control and polyline points in each map.

Comparisons between the processing times of *TopoVW* and *Grid-Gen* were performed and are shown in Table 2.2. Both methods were configured to simplify the maps by removing 50% of their points. The uniform grid size was chosen based on the configuration that led to the fastest performance in Magalhães et al's paper [28]. Table 2.1 presents the time that the two methods spent in the simplification process and, also, the I/O time to load the map (in the GISCUP GML format) and write the output (the I/O time is the same to the two implementations). Table 2.3 shows the number of points removed by simplification using *Grid-Gen* and *TopoVW*.

Observe that, in the worst case, the simplification step of *TopoVW* was 7 times slower than the same step in *Grid-Gen*. However, in all scenarios I/O used a considerable amount of the total processing time. Indeed, if the total running time of the

algorithms was considered, *TopoVW* was less than 3 times slower than *Grid-Gen* in the worst test case.

Table 2.1 – Processing-time (in milliseconds) for simplifying the datasets with control points.

Dataset	3	4	5	6	7
Polyline pts. ($\times 10^3$)	8	30	30	300	4000
Control points	151	256	1607	10^4	10^7
I/O	12	37	53	447	89187
Simp. (*TopoVW*)	9	52	46	1469	95732
Simp. (*Grid-Gen*)	4	16	15	196	19158

Table 2.2 – Processing-time (in milliseconds) for simplifying the datasets without using control points.

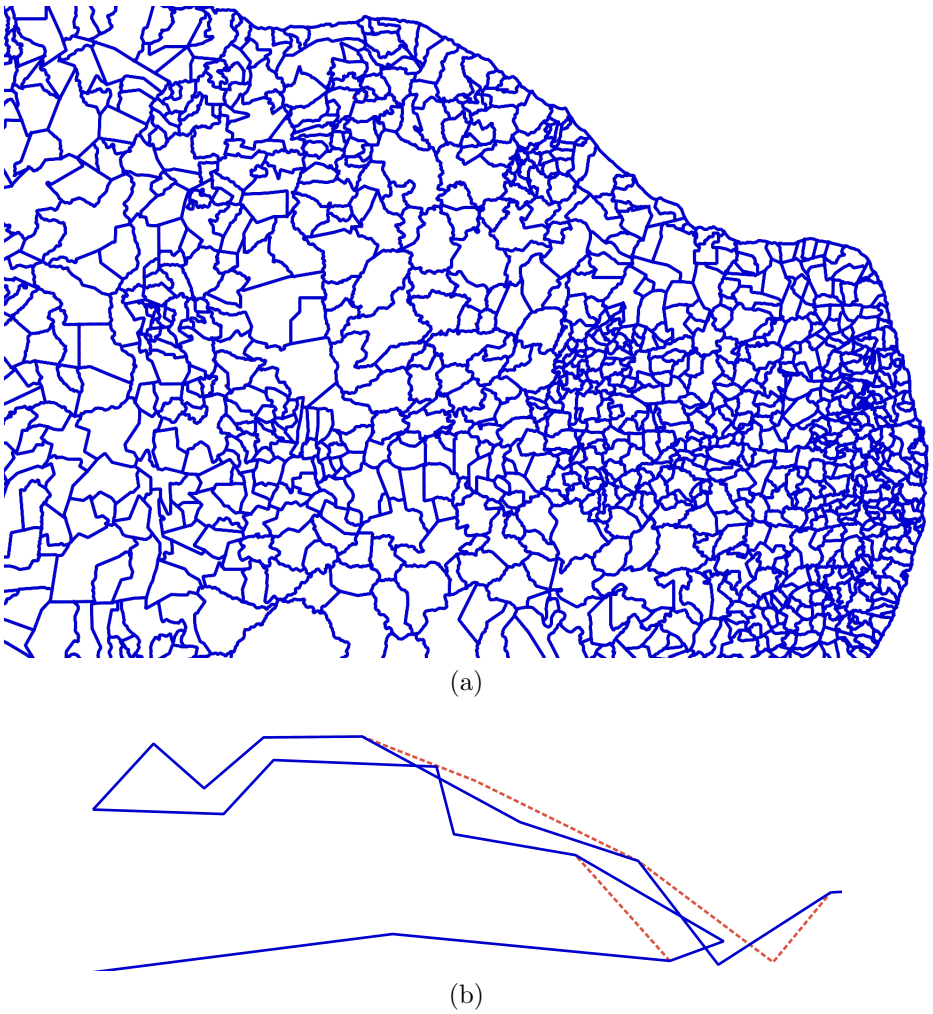
Dataset	3	4	5	6	7
Control points	151	256	1607	10^4	10^7
I/O	9	32	37	490	3856
Simp. (*TopoVW*)	8	49	42	1400	89439
Simp. (*Grid-Gen*)	3	14	12	195	16892
Simp. (VW)	6	24	23	900	21062

Table 2.3 – Maximum number of points removed by simplification.

Dataset	1	2	3	4	5	6	7
No. of points	992	1564	8531	28014	28323	342738	3645559
Visvalingam-Whyatt	938	1472	7579	25308	23661	309648	3627135
Grid-Gen	938	1463	7578	25298	23639	309445	3627008
TopoVW	938	1463	7578	25297	23640	309424	3626993
Grid-Gen w/ Control Pts.	928	1435	7545	25212	23411	308992	3609484
TopoVW w/ Control Pts.	927	1430	7545	25207	23390	308935	3627008

2.5 Conclusions and future works

It was presented *TopoVW*, a heuristic that uses techniques based on the Visvalingam-Whyatt (*VW*) [8] algorithm to perform map simplification generating maps that not only are topologically consistent but also tries to preserve the similarity with the original datasets. This means that, if applied to a map containing roads, buildings and rivers, *TopoVW*'s simplification will neither produce intersections of such features nor change the topological relationships (for example, if a road is in the left side of a building it will continue in the left side after the simplification). *TopoVW* also supports the use of control points in the generalization process, avoiding changing



(a)

(b)

Figure 2.11 – Comparison between the output generated by VW (represented by blue/solid lines) and TopoVW's output (represented by red/dashed lines) using dataset 6 as input: (a) Part of the Northeast of Brazil. (b) Zoom in a region where it is possible to see a small difference between the outputs.

the topological relationships between the polylines and these control points. Control points may be used in practical applications such as simplifying state/county maps, where lines represent county borders and cities may be represented by points (more specifically, control points). If *TopoVW* is applied to this situation, it is guaranteed that no point will be on the wrong side of the lines. Namely, after simplification, every city will stay inside the correct county or state.

During the experiments, we observed that *TopoVW*'s output is very similar to *VW*'s. In fact, the small differences that we observed between the methods happened in regions were *VW* created topological errors. Since *VW* is suitable for creating map generalizations and caricatures with good quality, this suggests that *TopoVW* also has this feature.

As the tests showed, *TopoVW* is fairly efficient. Furthermore, it adds little overhead to the *VW* algorithm (an algorithm that, despite keeping the similarity between the input and output, does not try to preserve the topological relationships),

being only 4 times slower than *VW*. Finally, *TopoVW* uses dummy control points to treat its special cases and, thus, it is very simple to implement.

Future work includes comparing *TopoVW*'s performance and output quality against methods based on the Ramer-Douglas-Peucker algorithm. Furthermore, another extension is to perform more tests in order to evaluate the quality of *TopoVW*'s output not only in maps, but also in vector illustrations. Finally, another interesting work is to adapt *TopoVW* to perform simplification of 3D vector objects.

3 Using Rational Numbers and Parallel Computing to Efficiently Avoid Round-off Errors on Map Simplification [1]

Abstract

This paper presents *EPLSimp*, an algorithm for map generalization that avoids the creation of topological inconsistencies. *EPLSimp* is based on Visvalingam-Whyatt's (VW) algorithm on which least "important" points are removed first. Unlike VW's algorithm, when a point is deleted a verification is performed in order to check if this deletion would create topological inconsistencies. This was done by using arbitrary precision rational numbers to completely avoid errors caused by floating-point arithmetic. *EPLSimp* was carefully implemented to be efficient, although using rational numbers adds an overhead to the computation. This efficiency was achieved by using a uniform grid for indexing the geometric data and parallel computing to speedup the process.

3.1 Introduction

The map simplification process, also known as map generalization, allows the production of maps with different levels of details [39]. It consists of removing information that is not relevant to the viewer, while preserving essential features on the map. Generalization is inherent to every geographical data since every map consists of generalized representations of reality, and the more generalized a map is, the more distant it becomes from the real world [23]. The output of this process is a map with more desirable properties than those from the input map. An example of generalization is scaling a map of a single town which contains detailed information about streets and buildings. When scaling this map to show nearby towns it may be necessary to simplify it so that it is not overburden by unimportant data.

A challenge in generalization is to find a balance between simplification and reality. Map simplification can produce inappropriate results as it may affect topological relationships. These results are said to be *topologically inconsistent* and they may present relationships that are conflicting with reality. For example, the simpli-

[1] The work presented in this chapter shows the occurrence of round-off errors and the problems that arise from it on 2D simplification. It also presents a method to avoid such errors. *Using Rational Numbers and Parallel Computing to Efficiently Avoid Round-off Errors on Map Simplification*, presented at GeoInfo 2015 (XVI Brazilian Symposium on GeoInformatics). [18]

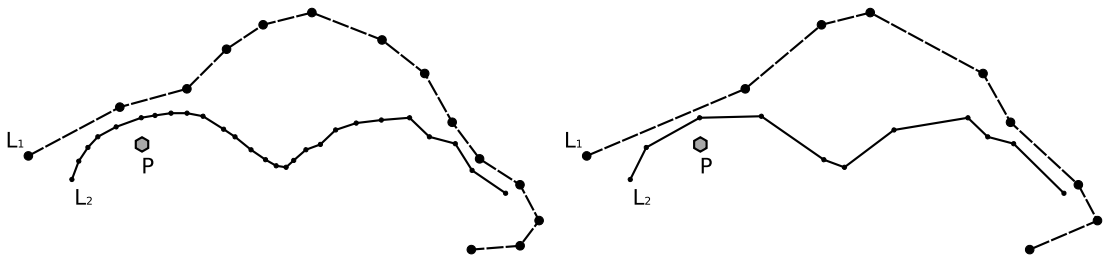
fication can create self-intersecting lines, improper intersections between lines and polygons, etc.

Another kind of topological inconsistency is the sidedness change, that is, after performing simplification, a feature can be on a different side regarding other feature on the map. For example, after the simplification of a line, a point which was originally on the right side of this line now can be on the left side. Thus when designing simplification algorithms it is important to guarantee topologically consistent results.

3.2 Polyline Simplification

An approach for performing map simplification is to reduce the complexity of its lines. That means making simpler representation of curves or polygon edges. Usually, lines are represented by polygonal chains or polylines. A polyline is a serie of segments defined by a sequence of n vertices (v_1, v_2, ..., v_n), where each segment consists of two endpoints and adjacent segments share a common endpoint. Figure 3.1(a) shows an example of two polygonal chains L_1 and L_2, and also a control point P (gray hexagon) that does not belong to a polyline but is considered relevant or meaningful.

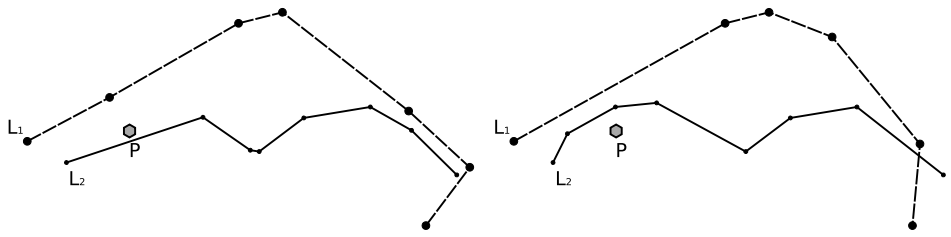
The basic idea of line simplification consists of removing points and representing the original curve using approximation with fewer vertices. Figure 3.1(b) presents an example of the simplification of the lines shown in Figure 3.1(a). Two famous and frequently used line simplification algorithms are the Ramer-Douglas-Peucker's algorithm (RDP) [6, 7] and Visvalingam-Whyatt (VW) [8] algorithm.



(a) (b)

Figure 3.1 – (a) Example of polylines L_1 and L_2 and a control point P. (b) Simplification of L_1 and L_2. Notice that topology consistency is preserved: no intersections were created and sidedness is maintained.

The line simplification process can bring inconsistencies to the output if some care is not taken. Figure 3.2 shows two examples where removing certain points from the polylines in Figure 3.1(a) would cause topological inconsistency: (a) after simplification, point P is on the other side of the simplified line L_2; (b) a "nonexistent" intersection between lines L_1 and L_2 is created.



(a) (b)

Figure 3.2 – Inconsistent simplification output: (a) P is on the wrong side of line L_2; (b) nonexisting ntersection between lines L_1 and L_2 is created.

Topological inconsistency may be created by some simplification algorithms such as the ones based on the RDP method. But there is another source of error that affects even algorithms that attempt to avoid inconsistencies: round-off errors resulting from floating point arithmetic. These errors occur because real numbers cannot be exactly represented in computational systems, instead, an approximation of the real number is used [40]. In order to overcome such problems, the best strategy is to make use of Exact Geometric Computation [17].

In this paper is presented a method that uses rational numbers and parallel computing to solve the following variation of the generalization problem: given a set of polylines and control points, the goal is to simplify these polylines by removing some of their vertices (except endpoints) such that topological relationships between pairs of polylines and between polylines and control points are maintained. In practice, polylines may represent boundaries of counties or states, and control points may represent cities within these states. The introduction of rational numbers was used to prevent errors introduced by rounding in floating point arithmetic. The use of arbitrary precision numbers is expected to increase the overall execution time of the algorithm since its operations are more complex. In order to compensate for this performance drop, parallel computing is used.

3.3 Related Works

In this section we describe algorithms for line simplification as well as problems that arise from floating-point arithmetic.

3.3.1 Algorithms for Line Simplification

Many algorithms for line simplification have been developed so far. One of the most famous is the Ramer-Douglas-Peucker's algorithm (RDP) [6, 7]. Its basic idea is to start with a very rough approximation of the original line (i.e. a straight line connecting the end vertices) and iteractively refine the approximation including, in each step, the vertex that is farthest from the current line. The method stops

when the distance between the farthest vertex and the line is greater than a given threshold (the smaller the threshold the less simplified the line is).

The RDP algorithm does not take topological consistency into consideration and may generate inconsistent results. An approach proposed by Saalfeld [31] attempts to avoid such inconsistencies. It uses Douglas-Peucker's algorithm to simplify lines and then starts a refining process by adding points to the output line so that the curve no longer presents any inconsistency. Noteworthy to mention that adding points to a curve may eliminate previous inconsistencies but may create new ones.

Another approach based on Douglas-Peucker was proposed by Li et al. [33]. It intends to avoid topological inconsistencies as well as cracks on polygon shapes using a strategy based on detection-point identification, which are points lying within a minimum boundary rectangle (MBR) of the bounded face formed by a subpolyline and its corresponding simplifying segment. These detection-points are used for consistency verification of the simplification process.

Visvalingam and Whyatt [8] proposed a method (called the *VW* algorithm) for line generalization that uses the concept of *effective area* of a point to define the priority of its removal. The *effective area* of a point v_i, for $1 < i < n$, in a polygonal chain $v_1, \cdots, v_n$, is defined as the area of the triangle formed by v_i and its two adjacent vertices, namely, v_{i-1}, v_i, v_{i+1}. The *VW* algorithm considers that the "importance" of the points are proportional to their *effective area* and, therefore, it ranks the points and simplifies the polylines by removing first the points with smaller areas.

Even though the *VW* algorithm performs simplification with good quality, it does not avoid topological problems in the map. To solve this problem, Gruppi et al.[18] developed *TopoVW*, a variation of the *VW* algorithm that avoids the creation of topological inconsistencies. Similarly to *VW*, *TopoVW* processes the points in an order based on their *effective area* but only removes a point v_i if its removal does not create inconsistencies in topology. When a point is removed the effective areas of its two neighbor points in the line are updated since the triangle associated with them change. *TopoVW* may be configured to stop when the number of points removed reaches a limit or when the smallest effective area of the points is greater than a given threshold.

Although some of the methods previously mentioned have mechanisms to detect and prevent topological inconsistencies created by the simplification process itself, these problems may still happen because of round-off errors related to the use of inexact arithmetic to process the points' coordinates.

3.3.2 Round-off Errors in Floating Point Arithmetic

The computational representation of a non-integer number is made by adjusting this number to a finite sequence of bits, this possibly causes the number to be an

approximation most of the time. Furthermore, even if some numbers can be exactly represented, arithmetic operations applied to these numbers may generate a result that is not exactly correct. In geometric algorithms this is a great issue since they may result in inconsistent outputs.

Kettner et al. [1] presented a study of how rounding in floating point arithmetic affects the planar orientation predicate and as consequence the planar convex hull problems. The planar orientation predicate is the problem of finding whether three points p, q, r are collinear, make a left turn, or make a right turn. This predicate is computed by verifying the sign of a determinant involving the points.

This determinant will be positive, negative or zero which means that points (p, q, r) form a left turn, right turn or are collinear, respectively. Due to round-off errors in floating point arithmetic the results can be classified incorrectly due to *rounding to zero*, *perturbed zero*, or *sign inversion*. Respectively, it means a non-zero result may be rounded to zero, a zero result may be mis-classified as positive or negative, and a positive result may be mis-classified as negative or vice-versa.

To observe the occurrence of issues caused by floating-point arithmetic, Kettner et al. [1] developed a program to apply planar *orientation* predicate on a point $p = (p_x + xu, p_y + yu)$ where u is the step between adjacent floating point numbers in the range of p and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either 1, -1 or 0 if the point corresponding to the matrix position is to the right, to the left or on the line that passes through q and r. Figure 3.3 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$ and $q = (12, 12)$ and $r = (24, 24)$. White cells represent correct output. The black diagonal line is an approximation of line (q, r). Black cells represent incorrect output, that is, black points above the diagonal were considered to form a right turn with the line (q, r), which is not true, it also applies to the points below the diagonal which were said to form a left turn with line (q, r). Gray cells contain points considered collinear to (q, r). According to Kettner et al., even using extended double arithmetic was not enough to overcome this issue.

As shown by [1], these inconsistent results in $orientation(p, q, r)$ predicate could make algorithms that use this predicate (such as the Incremental Convex Hull algorithm) to fail.

A well-known technique to get around round-off errors in floating point arithmetic is the Epsilon-tweaking, that consists in comparing numbers using a relatively small tolerance value epsilon (ϵ). In practice, epsilon-tweaking fails in several situations [1]. Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers [41]. However, Snap rounding can generate inconsistencies and deform the original topology if applied consecutively on a data set. Some variations of this technique attempt to get around these issues [42, 43].

One of the most robust ways for eliminating rounding errors in geometry is by

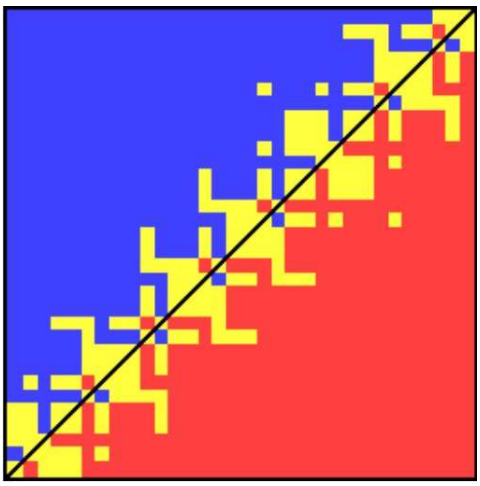


Figure 3.3 – Geometry of the planar orientation predicate for double precision floating point arithmetic. Blue represents points that form a left-turn with the black line. Red represents points that form a right turn with the black line. Yellow points are collinear with the line. The diagonal line (in black) is an approximation to line (q, r). Source: Kettner et al.[1].

using Exact Geometric Computation (EGC). According to Li [17], any problem handled by other approaches can also be solved by ECG. Additionally, EGC can do even more and the solutions may be of higher quality. This can be achieved by using arbitrary precision rational numbers [17], which eliminates rounding errors but considerably decreases performances as most operations are more computationally intensive.

3.4 Evaluation of Round-off Errors on Map Simplification

Similarly to other geometric problems, map simplification is also affected by round-off errors. As mentioned in section 3.3, *TopoVW* processes points in an order defined by their effective areas and only removes a point if its removal does not cause topological inconsistencies on the map. Given a polyline point v from a map, the removal of v causes a topological inconsistency if there is another point (that may be a polyline or a control point) inside the triangle formed by v and its two adjacent vertices in its polyline.

If the *point-in-triangle* test fails returning a false positive a point that could have been removed from the polyline will not be removed. If this test returns a false negative, on the other hand, topological inconsistencies may be created on the map.

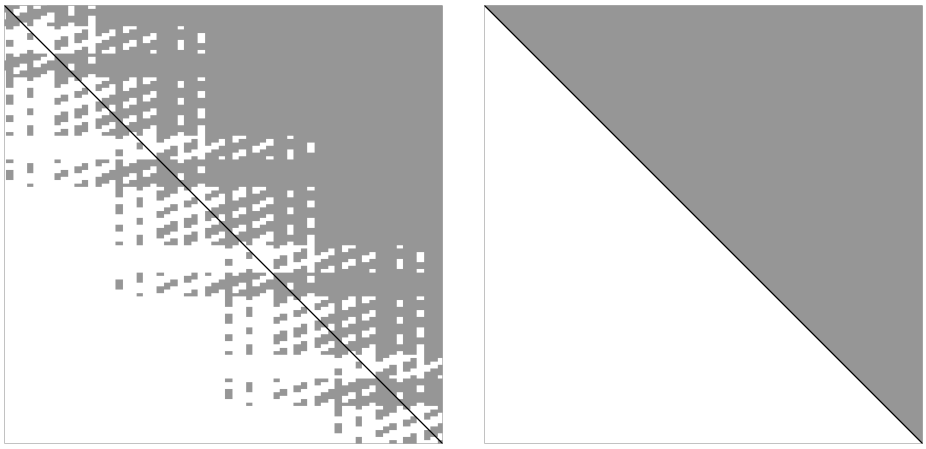
In *TopoVW*, the test to determine if a point p lies inside the triangle T formed by points r, s and t is performed by computing the barycentric coordinates of p in T, i.e., p is expressed in terms of three scalars a, b and c such that $p_x = ar_x + bs_x + ct_x$, $p_y = ar_y + bs_y + ct_y$, and $a + b + c = 1$. Point p lies in T if and only if $0 \leq a \leq 1$ and $0 \leq b \leq 1$ and $0 \leq c \leq 1$. A function $is\_inside(r, s, t, p)$ to perform the *point in triangle* test using the barycentric coordinates was implemented in C++. This

approach is similar to the one used by Kettner et al. shown in Section 3.3.2.

In a similar manner to the orientation test presented in the previous section, the function $is\_inside(r, s, t, p)$ may also return incorrect results in two situations:

- *false inside*: erroneously determine an outer point as inside;
- *false outside*: erroneously determine an inner point as outside;

To observe the occurrence of false inside and false outside results on the point-in-triangle test. We performed an experiment similar to Kettner's [1]. We start with three point as vertices of the triangle (r, s, t) and a point p. For each iteration, p varies in the neighborhood of its starting value, and we apply the function $is\_inside(r, s, t, p)$ to determine whether p is inside or outside triangle (r, s, t). This experiment was performed twice, initially the program was implemented using floating-point numbers. Then, we used GMP rational numbers to perform the tests the second time. The results can be seen in Figure 3.4(a) and (b). The gray areas correspond to points inside the triangle, whereas white areas are points considered outside. In both figures, the area in the upper right of the diagonal corresponds to the interior of the triangle. There are many wrong classifications in Figure 3.4(a). In Figure 3.4(b) no point is misclassified.



(a) (b)

Figure 3.4 – Result of the *is_inside* function experiment: (a) using floating-point numbers; (b) using GMP's rational numbers. The upper right portion of both figures corresponds to the interior of the triangle. Gray points were classified as inside, white points were classified as outside the triangle.

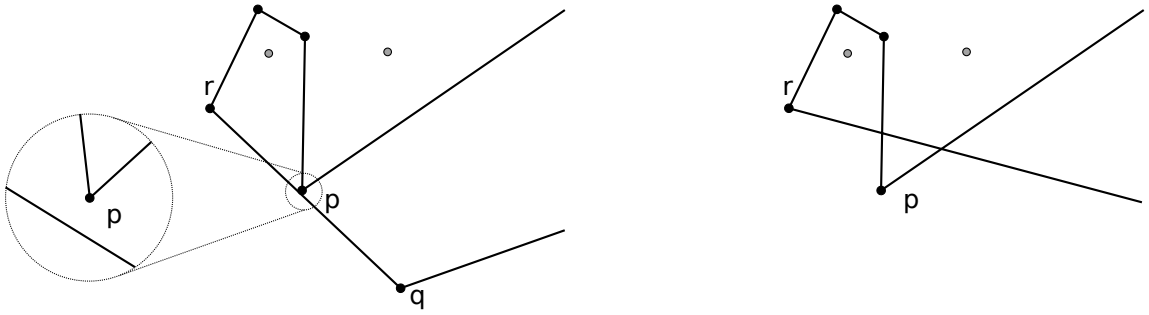
Since *is_inside* is *TopoVW*'s key operation, the method may avoid simplifying lines due to *false inside* appearance. Even more alarming, it may remove points on the presence of *false outsides*, what would change the topological relationships. Figure 3.5 shows an example of *false outside* simplification. In this example there are two non-intersecting lines (solid and dashed) as shown in Figure 3.5(a), the

zoomed area shows explicitly that both lines do not intersect. Point p is inside the triangle formed by points (r, q, w) with w not shown in the figure to preserve simplicity. However, due to a *false outside* failure point q is removed resulting in an intersection as seen in Figure 3.5(b).

Another instance of this problem is shown in Figure 3.6, where a single line is simplified. Similarly to the previous example, vertex p is inside the triangle formed by (r, q, w) but it is seen as a *false outside*. Vertex q is removed by the simplification process causing the line to self-intersect as seen in Figure 3.6(b).

(a) (b)

Figure 3.5 – (a) Example input on which false outside failure occur, two lines (solid and dashed) do not intersect. (b) Result of simplification with false outside, the removal of point q causes the lines intersect after simplification.



(a) (b)

Figure 3.6 – (a) Example input of a single line and the occurrence of a false outside. (b) Simplification with a false outside point. The removal of point q produces a self-intersecting line.

3.5 The *EPLSimp* Method

To avoid adding topological errors to the map in the situations described in section 3.4, we developed *EPLSimp*, a simplification algorithm based on *TopoVW* that uses exact arithmetic to completely avoid the round-off errors that may happen during

the point in triangle tests. In *EPLSimp*, all non-integers variables are represented using arbitrary-precision rational numbers. Since exact arithmetic is usually much slower than arithmetic with floating point numbers (that usually can be performed natively on the CPU), some optimizations were implemented in order to reduce the performance penalty that it introduces.

First, similarly to *TopoVW*, we used a uniform grid to index the polyline and control points from the map. The idea is to create a regular grid, superimpose it with the map and insert in each cell c the control points and polyline points that are inside c. Then, given a triangle T, only points in the uniform grid cells intersecting T need to be tested in order to verify if there is a point inside T.

One advantage of the uniform grid over more complex data structures such as quadtrees is that it is easier to be constructed and maintained. Given a set S of points, we compute the uniform grid by performing only one pass through the dataset: for each point p in S, the cell c from the grid where p should be is computed (by dividing p's coordinates by the dimensions of the grid cells) and p is inserted in c.

Since the slowest step during the construction of the grid is the computation of the cell in which each point p is (due to the division operations with arbitrary-precision rationals), we used parallel programming to accelerate this step. The idea is to pre-compute in parallel the cell in which each point is and, after that, insert the points in the cells (this insertion step is not done in parallel in order to avoid the cost of synchronizations).

After indexing the points, the next step consists in simplifying polylines. As mentioned in section 3.3, *TopoVW* sorts the points based on their effective areas and processes them by removing the ones whose removal would not create topological problems in the map. To accelerate the simplification process used in *TopoVW*, we subdivided the polylines into sets such that polylines from different sets may be simplified independently in parallel not requiring the synchronization of data structures accesses.

Algorithm 3 presents the simplification algorithm and the strategy used for subdividing the polylines into sets that can be simplified in parallel. This subdivision is also performed using a uniform grid (this grid may have a resolution different from the uniform grid used for indexing the points). We create this new uniform grid and, then, insert in each grid cell the polylines that are completely inside this cell. The polylines in different grid cells could be processed independently since the triangle formed by any polyline point never contains a point from another cell. On the other hand, polylines intersecting more than one cell cannot be processed in parallel without synchronization. For example, even though the polyline containing the vertex v in Figure 3.7b(a) does not intersect the cell containing the polygon P, before deleting v it is necessary to access the cell containing polygon P in order to

verify if the deletion of v causes a topological inconsistency. Therefore, if the two polylines in this figure are simplified in parallel the algorithm would need to perform synchronizations.

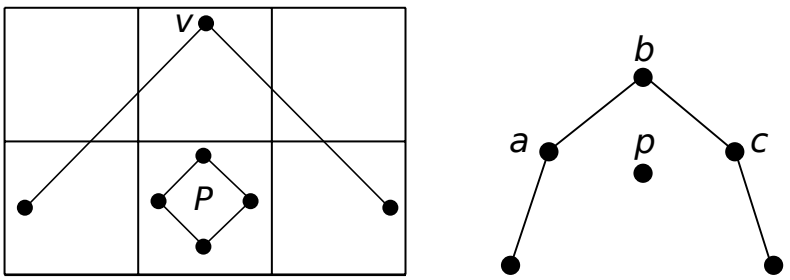
After processing all the polylines lying completely in single cells, we repeat the simplification process for the polylines intersecting more than one cell. In order to be able to do that in parallel, we reduce the uniform grid resolution, reclassify the remaining polylines and, then, simplify the ones that lie in single cells in this new uniform grid. This process is repeated until there is no more polyline to be simplified (eventually all the polylines will be processed since when the uniform grid is reduced to one cell all polylines that were not processed yet will lie in this unique cell).

To avoid the necessity of synchronizations between threads processing different sets of polylines, the simplification stopping criteria used in *EPLSimp* is the effective area of the points. That is, the thread simplifying a set of polylines stops the process whenever the point with smallest effective area in the set has an area greater than a given threshold. If the stopping criteria was the number of points removed, synchronizations would be necessary to ensure that all threads stop simplifying lines when the global number of removed points reaches the target number.

Algorithm 3 Parallel map simplification algorithm.

```
1: M: input map
2: MaxArea: maximum effective area of a point to be removed
3: GridSize: initial resolution of the uniform grid used to separate the polylines.
4: while GridSize > 0 do
5:     ug ← GridSize × GridSize uniform grid
6:     for each polyline p in M not simplified yet do
7:         if p is completely inside a cell c from ug then
8:             Insert p into c
9:     for each cell c in ug do //Parallel for loop
10:        //Iterate in an order based on effective areas
11:        for each point v_i in polylines from c | effectiveArea(v_i) < MaxArea
   do
12:            if ∄ point p | is_inside(v_{i-1}, v_i, v_{i+1}, p) then
13:                Remove the point v_i from its polyline
14:    GridSize ← GridSize/2
```

It is important to mention that we have considered other two parallelization strategies. First, we could pre-process the map verifying for each point if there is another point inside the triangle defined by it and its two neighbors. This pre-processing could be performed in parallel. After labeling the points that can safely be removed (that is, the ones without other points in their triangles), we could just remove the ones with smaller effective areas. This strategy would not work very well because when a point is removed the triangle of its two neighbors change. For example, in Figure 3.7b(b), any of the points a or b or c may be removed without changing the topological relationship between the polyline and the control point



(a) (b)

Figure 3.7 – (a) Example where a polyline intersecting multiple cells needs to access data in a cell it does not intersect. (b) Example where the deletion of a point makes the deletion of other points infeasible. Points a, b, and c are vertices of the polyline.

p. However, if a or c is removed the triangle associated with b will contain p and, therefore, b will not be a candidate to be removed anymore.

Another parallel strategy would be to perform the point inside triangle test in parallel. That is, given a triangle T, after using the uniform grid to select the points that are candidate to be in T we could perform the test to verify if each point is really inside T in parallel. However, preliminary experiments showed that, because of the uniform grid, the average number of points that need to be effectively tested in this step is usually small and, therefore, the performance gain obtained by processing them in parallel would not be good if compared with the overheads associated with the parallelism.

3.6 Experimental Evaluation

We evaluated *EPLSimp* by implementing it in C++ (the library GMPXX [44] was used to provide arbitrary precision arithmetic) and performing experiments in some small datasets artificially generated to contain polylines and control points that would introduce topological inconsistencies due to round-off errors in the simplification performed by *TopoVW*. Furthermore, experiments were performed in 3 real-world maps in order to evaluate the performance of *EPLSimp*. The computer used has a dual E5-2687 8-core/16-thread Intel Xeon CPU and 128 GB of RAM.

In the first set of experiments, we used the maps artificially generated to contain points in positions where the point-in-triangle tests would give a false negative answer (similar to the examples presented in section 3.4) and, therefore, methods such as *TopoVW* would create topological errors during the map simplification. As expected, because of the use of exact arithmetic, *EPLSimp* was able to simplify these maps without creating any topological inconsistency.

Next, we performed experiments in three datasets to verify the overhead added by the use of arbitrary precision rational numbers in *EPLSimp*. Dataset 1 was the largest dataset used in the ACM GISCUP competition 2014. It contains 30,000

polyline points and 1,607 control points. Dataset 2 represents the Brazilian county subdivision map available in the IBGE (the Brazilian geography agency) website and it contains 300,000 polyline points and 10,000 control points (the control points were positioned randomly in the map). Dataset 3 represents the United States county subdivision map available in the United States Census website and it has 4 million polyline points and 10 million control points (that were also positioned randomly in the map).

The choice of the uniform grid's dimension used by *TopoVW* and *EPLSimp* to index the points affects the performance of both methods and it can be performed using several strategies. For example, *TopoVW* automatically defines the grid size by computing the total number of polylines/control points in the map and chooses the grid dimension estimating the average number of points per cell close to a constant (this constant was determined experimentally). Since the best grid size for *TopoVW* may not be the best grid size for *EPLSimp* and since we want to compare the performance of these two methods, we chose experimentally, for each method and dataset, a configuration that presents the best performance (for example, in dataset 2, *TopoVW* and *EPLSimp*used grids with, respectively, 512^2 and 2048^2 cells).

The uniform grid that *EPLSimp* uses to classify the polylines that are processed in parallel was configured to have initially 256^2 cells and to iteratively reduce the resolution to half after completely processing each set of polylines that can be processed in parallel. As mentioned in section 3.5, this process is repeated until all polylines have been simplified, what happens, in the worst case, when the grid has only one cell.

Table 3.1 presents the wallclock-time (in milliseconds) of the two methods in two situations: in the first one they were configured to remove the maximum amount of points that they can remove without creating topological errors. In the second one, they were configured to remove 50% of the points. Row *initialize* contains the time for initializing the algorithm and includes the time for creating the data structures (such as the uniform grids). Row *simplify* contains the time spent in the simplification process. In all tests *EPLSimp* was tested using 16 *threads*.

EPLSimp was, on average, less than twice slower than *TopoVW*, even though we store and process all points coordinates using arbitrary precision rational numbers, that are much more computationally expensive to process than floating point numbers. This happens because *EPLSimp* was carefully implemented using techniques such as parallel computing and the uniform grid to accelerate the simplification process. It is worth mentioning that one of the advantages of the uniform grid over other indexing techniques (such as Quadtrees) is that it is easily parallelizable and can be created by performing a single pass over the data (this is particularly important for efficiency since the indexing is performed using coordinates represented by rational numbers).

Table 3.1 – Times (in ms) for the main steps of the map simplification algorithms. Rows *Max.* represents the time for removing the maximum amount of points from the map while rows *Half* represents the time to remove half of the points.

	Dataset	1		2		3	
	Method	TopoVW	*EPLSimp*	TopoVW	*EPLSimp*	TopoVW	*EPLSimp*
Max.	Initialize	4	22	28	190	1828	5353
	Simplify	39	60	626	445	46069	57095
	Total	43	82	654	635	47897	62448
Half	Initialize	4	22	28	186	1847	5447
	Simplify	25	41	357	331	23021	48090
	Total	29	63	384	517	24868	53537

Table 3.2 – Times (in ms) for initializing and simplifying maps from the 3 datasets considering different number of threads. The simplification was configured to remove the maximum amount of points from the maps.

		Initialization			Simplification		
	Dataset	1	2	3	1	2	3
Threads	1	71	655	26833	176	1574	250237
	2	91	568	15483	152	1150	131310
	4	54	422	9853	99	689	82641
	8	34	240	6552	61	483	62089
	16	22	190	5353	60	445	57095

Table 3.2 and Figure 3.8 show the evaluation of the scalability of *EPLSimp* considering 5 different number of threads. In these datasets, *EPLSimp* had a speedup of 2× when two threads were used and this speedup increased slowly for larger amounts of threads. For example, the running-time using 16 threads was not much different from the time using 8 threads. Some reasons for this behavior are: first, due to Amdahl's law [45], sequential parts of the algorithm limits its scalability; furthermore, some polylines sets may take more time to be simplified than others, what causes load imbalance in the threads; finally, when several threads run in parallel the memory accesses may saturate the memory bus. Anyway, it is worth mentioning that typical computers nowadays have 2 or 4 cores and, therefore, *EPLSimp* is able to present a good scalability in those computers.

3.7 Conclusion and Future Works

This paper presented *EPLSimp*, an algorithm for map simplification that does not produce topological inconsistencies. It uses arbitrary precision numbers to avoid round-off errors caused by floating-point arithmetic, which could lead to topological inconsistencies even in methods designed to avoid these problems, such as *TopoVW*.

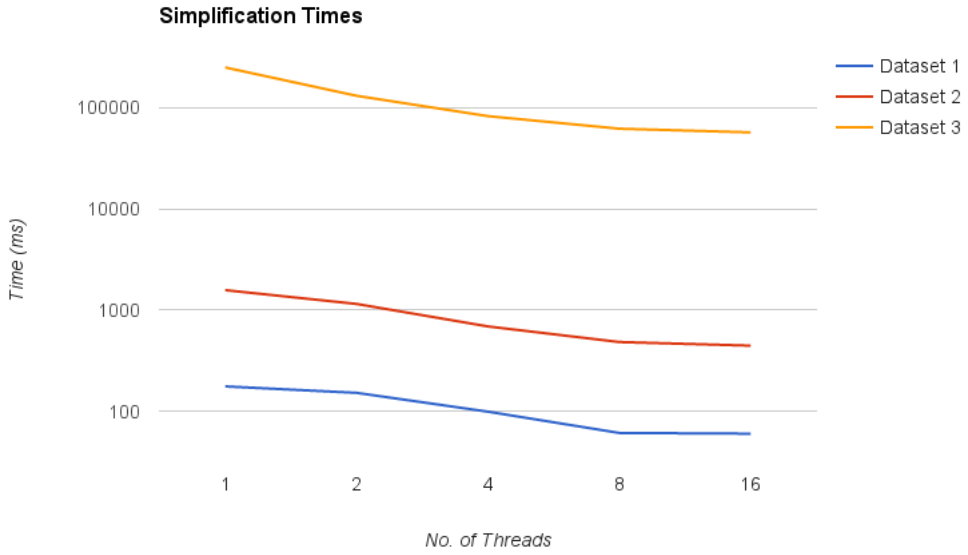


Figure 3.8 – Simplification times (in ms) for EPLSimp on three datasets using different numbers of threads.

EPLSimp was implemented to be efficient even though it uses arbitrary precision numbers, which are much slower to be processed than floating-point numbers. This efficiency improvement was achieved by using a uniform grid to index the geometric objects and, also, high performance computing. As a result, using 16 threads *EPLSimp* was, on average, less than twice slower than *TopoVW*, even though the latter performs all computation using inexact floating-point numbers (that are natively supported by the CPU) and then can generate "wrong" (or inconsistent) results.

For future work, we intend to develop other GIS algorithms using arbitrary precision arithmetic. Furthermore, adapting *EPLSimp* to simplify vector drawings and 3D objects is also an interesting future research topic.

4 Fast and Topologically Consistent Triangular Mesh Simplification Using Uniform-Grid [1]

Abstract

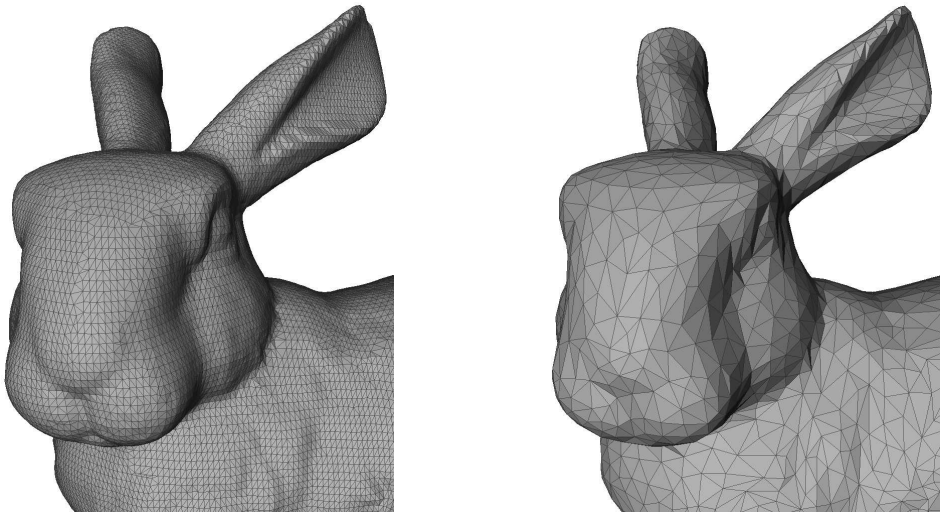
In this paper we present *UGSimp*, a triangular mesh simplification algorithm that avoids creating self-intersections on the mesh by performing preemptive tests before edge collapses. Due to the occurrence of round-off errors on floating-point arithmetic, *UGSimp* uses multiple precision rational numbers to carry out intersection tests. This operation introduces an overhead to the processing, which is compensated by the use of parallel computing. The mesh is divided using a uniform grid where each cell has a queue of edges ranked by the Quadric Error Metrics. This allows edge collapses to be performed in parallel, with each thread simplifying edges inside each cell, independently. Results have shown that this parallel strategy makes *UGSimp* fairly efficient when using floating-point representation, and avoids round-off errors when using multiple precision rational numbers.

4.1 Introduction

Techniques for reducing the level of detail of 3D objects have been intensively studied by computer graphics practitioners. In spite of the recent advances in graphical hardware, processing massive 3D data may be very time-consuming. In order to reduce the processing time of such meshes, one can make use of a simplified model. Simplification consists of reducing the amount of complexity of 3D objects while keeping them similar to the original models. It is widely used in computer games and computer visualization, among other areas [46]. A triangular mesh is essentially made of vertices that are connected to form faces (triangles), and can be reduced by removing its vertices or its edges, such as through the edge collapse operation [10]. Figure 4.1(a) shows a mesh and Figure 4.1(b) shows a simplified version of it, reduced to 10% the amount of faces.

Although it is a heuristic process, simplifying a model can disrupt its topological consistency. This means that the relationship between its features may be changed,

[1] In this chapter, the topological inconsistencies are evaluated on triangular mesh simplifications. It also presents a method to avoid such errors. *Fast and Topologically Consistent Triangular Mesh Simplification Using Uniform-Grid, submitted to ICEIS 2017 (*19^{th} *International Conference on Enterprise Information Systems). [18]*



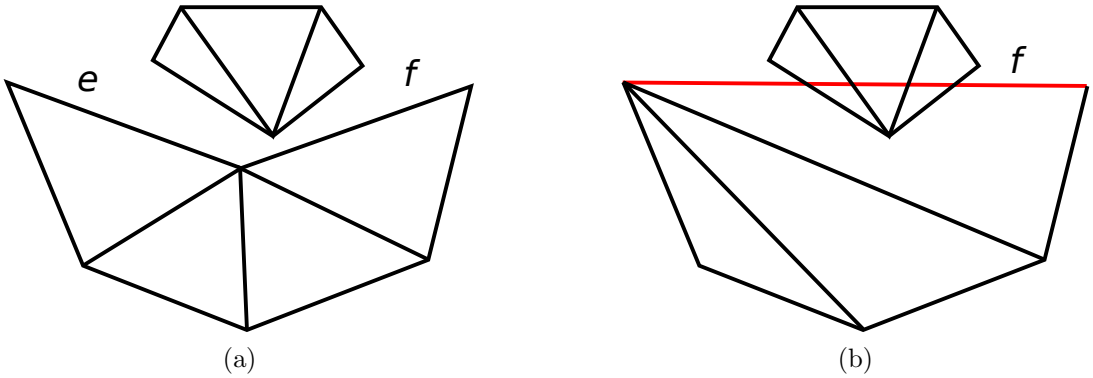
(a) (b)

Figure 4.1 – (a) Original bunny model with 69451 triangles; (b) A simplification of the bunny model with 6904 triangles.

such as two simplices (vertex, edge or face) intersecting each other improperly or a face flipping its orientation.

Thus, it is necessary to ensure that the simplification does not produce inconsistent results. But even when intersection prevention is present, using floating-point arithmetic may lead to failure due to round-off errors. An exact computing approach may be used to avoid this situation [17]. Figure 4.2 shows an example of inconsistent mesh as result of simplification. Assuming that edge e is removed, for instance, by an edge collapse, edge f (in red) is transformed and intersects other faces.

Guaranteeing topologically consistent results require an extra step on simplification algorithms, which may increase the execution time. In order to make up for this overhead, high performance computing (HPC) may be applied. Shared memory parallel architectures, such as OpenMP, are commonly used HPC techniques [47] [48].



(a) (b)

Figure 4.2 – Simplification resulting in an inconsistent mesh. (a) shows the original mesh, (b) is the simplified mesh, one edge intersects another mesh.

In this paper we present, *UGSimp*, a method for triangular mesh simplification that does not produce topologically inconsistent results by using exact computing predicates to perform preemptive tests, which may increase the execution time. In order to speed-up the algorithm, parallel computing was implemented using OpenMP. The mesh was divided using a uniform grid where each cell contains segments that could be simplified in parallel. Apart from the I/O operations and initialization of data structures, the program runs in parallel.

4.2 Related Works

Various methods for mesh simplification have been proposed in past decades. These can be organized according to the basic operation they use. Schroeder et al. [9] have proposed a strategy for vertex removal, called vertex decimation. This approach consists of choosing a vertex to be removed from the mesh, applying the removal of the vertex and its coincident faces, and repairing the mesh afterwards via retriangulation. Another approach for mesh reduction was proposed by Hoppe et al. [10] and consists of applying operations to edges, such as edge collapse and edge swapping, in order to reduce mesh complexity while keeping the error bound within a threshold. Cohen [49] introduced the Simplification Envelopes method, which produces a hierarchy of different level of details with both local and global scopes. A method proposed by Ronfard and Rossignac [50] uses the edge collapse operation to reduce a triangular mesh, using a geometric error cost function to rank the edges to be removed. Garland [11] proposed the Quadric Error Metrics for simplifying triangular mesh in a efficient manner. Each vertex has a 4×4 quadric matrix associated with it, the cost of collapsing an edge is given by the geometric displacement of faces it causes, which can be calculated by the multiplication of the placement vertex and its quadric matrix.

Edge collapse (or edge contraction) has been one of the most used techniques for triangular mesh simplification. Many recent methods use it as simplification operation while applying novel metrics and other techniques to guarantee a fast and high-quality approach. Campomanes [51] proposes a multi-objective method that targets accuracy and simplicity in the result of a simplification. Shontz et al. [52] proposed a triangular mesh simplification method combining CPU and GPU to accelerate it. Papageorgiou [22] presented an algorithm that runs mostly on the GPU and performs parallel edge collapses.

The aforementioned methods are able to efficiently simplify triangular meshes. However, they may present results with inconsistencies such as self-intersecting faces in the output. Staadt [53] proposed a method for preventing self-intersection during simplification by testing whether an edge collapse causes simplices to intersect. Gumhold [54] presented a method for intersection prevention that prevent collisions

and repair the mesh by moving its vertices, if necessary. Even though these algorithms attempt to avoid inconsistencies on the output, the use of floating-point arithmetic may cause intersection prevention (or detection) to fail due to round-off errors. That is, intersection tests may give *false negative* results, producing intersections between faces.

Avoiding round-off errors is crucial to ensure topologically consistent results. Some approaches can be used to prevent round-off errors: epsilon-tolerance (or epsilon-tweaking) uses a tolerance value ϵ when comparing to floating-point numbers, i.e., $x = y$ if $|x - y| < \epsilon$. The downside of this method is that it may not be possible to find an ϵ that always avoid rounding errors; *Snap-rounding* divides the map into cells and rounds vertices to its corresponding cell center. It affects the models topology and may lead to inconsistencies; Finally, exact arithmetic using arbitrary precision rational numbers is a strategy that never produces rounding-errors [55].

4.3 The Proposed Method

The algorithm receives an input mesh M with n vertices, containing a list of vertices, list of face incidences, the target number of vertices t as a percentage of the original number of vertices, and produces mesh M_o with approximately $t \times n$ of the original number of vertices.

Initially, the data structures containing vertices V, edges E, faces F, and their correspondences are assembled into fixed size vectors. The mesh is then split by a 3D uniform grid [20]. The uniform grid is a structure that divides the mesh into cells of same size, each grid cell contains a subset of V (therefore E and F as well). Each grid cell i has a priority queue H_i containing the edges inside cell i whose endpoints are inside i. The top of the queue has the edge with the lowest cost among all others of that cell. We define a *legal edge collapse* as an edge collapse operation that does not cause intersections. For each cell, a legal edge collapse operation is applied until the number of edges in that cell is reduced to a certain threshold. This task can be carried out in parallel, by assigning each cell to a thread. If the target number of vertices is not achieved, the uniform grid resolution is reduced, that is, a new uniform grid with fewer and larger cells is used. As a consequence, some of the edges that crossed cell borders are now completely inside cells and could be eligible for removal. This process is repeated until the target number of vertices is reached. Further details are described in the following sections.

4.3.1 Data Structures

The input mesh contains a list of vertices and a list of faces. Vertices are stored into an array V of floating point coordinates, each three elements of the array correspond to the coordinates (x, y, z) of a vertex. Faces are stored into an array F of indices

to vertices, each three integers correspond to a face. Edges are not given by the input but are computed from the faces and its vertices, and are implemented as directed edges. A vector of edges stores pairs of indices to its vertices. Auxiliary arrays are created in order to keep track of which faces and edges are associated to each vertex. That is, for each vertex $v_i \in V$ we have a list of faces adjacent to it, a list of half-edges leaving v_i and a list of half-edges arriving at v_i.

4.3.2 Parallel Edge Collapse

When compared to other methods for mesh reduction, edge collapse presents one of the best quality-efficiency ratio. To collapse, or contract, an edge e, its source vertex v_1 is merged with its destination v_2. Every half-edge that left (or arrived at) v_1 before the collapse now leaves (arrives at) v_2. Vertex v_2 is then moved to the midpoint of e. Subsequently, one vertex (v_1), and up to two faces, become degenerate and are removed. Figure 4.3 illustrates the process of edge collapse on a triangular mesh. All edges from v_1 are moved to v_2, this causes the gray colored faces to be compressed, becoming degenerate, v_1 and both degenerate faces are removed from the mesh. Finally, v_2 is placed at point $(v_1 + v_2)/2$.

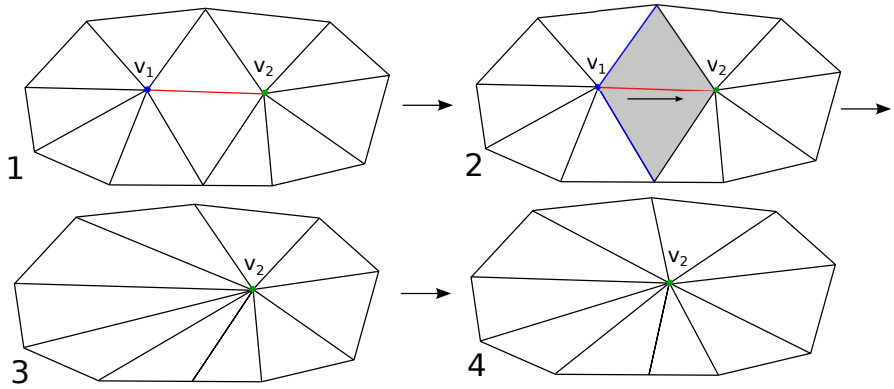
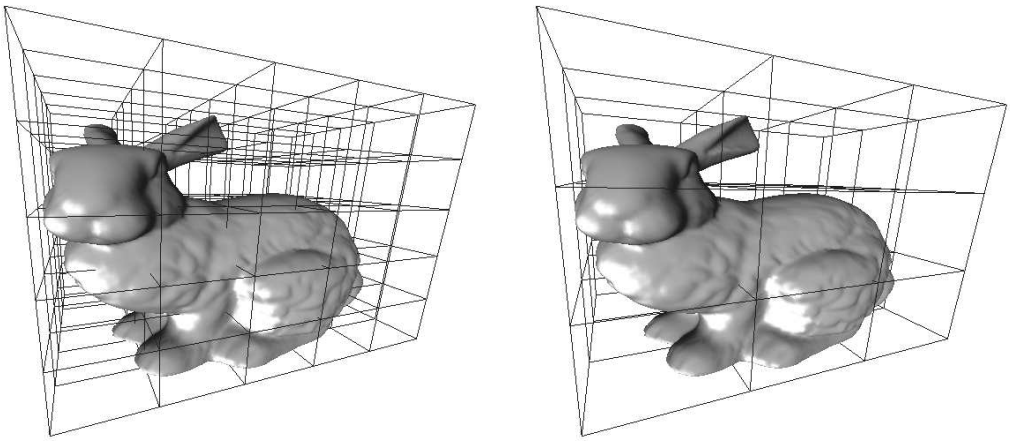


Figure 4.3 – Example of edge collapse. Faces in gray color become degenerate. Vertex v_1 is removed.

In order to perform parallel edge collapses, the mesh is overlapped by a 3D uniform grid (see Figure 4.4) on which each cell has a separate priority queue, implemented as a binary heap. Initially, each edge is inserted into a list of edges associated with its grid cell. The resolution of the uniform grid is a user-defined parameter. The default value for the grid resolution was 8 and it obtained empirically as the value resulting in the shortest execution time for all datasets.

To allow each grid cell to be processed independently, two constraints were included to select candidate edges. First, an edge needs to be entirely inside a cell. Second, all vertices adjacent to the given edge also need to be inside the cell. More precisely, by definition, the *star* of a vertex v is the set $S(v)$ of vertices adjacent to v. In addition, the star of edge (v_1, v_2) is $S(v_1) \cup S(v_2)$. Thus, an edge (v_1, v_2) is a candidate for removal if:



(a) (b)

Figure 4.4 – A triangular mesh overlapped by uniform grids of different resolutions: (a) $5 \times 5 \times 5$ cells; (b) $3 \times 3 \times 3$ cells.

- Both v_1 and v_2 are inside cell i of the uniform grid and;
- $S(v_1) \cup S(v_2)$ are also inside cell i.

In Figure 4.5, edge e_1 is a candidate for removal, the blue vertices indicate the star of e_1, which is entirely inside cell 0. The red vertices form the star of edge e_2, which is not entirely in a single cell, hence, e_2 is not a candidate for removal. The second constraint allows multiple threads to work on the mesh, simplifying different cells without thread collision.

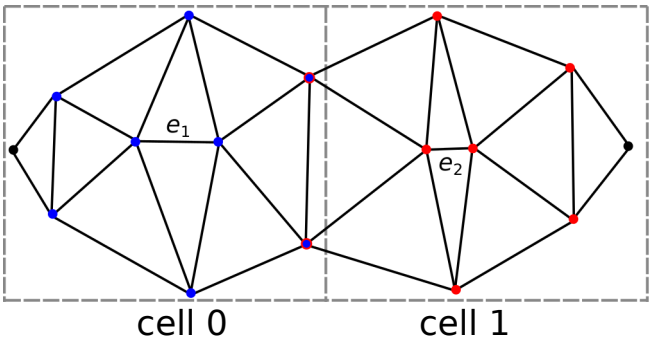


Figure 4.5 – The contraction of e_1 is allowed since its star is entirely inside cell 0. On the other hand, contracting e_2 could cause thread collision, therefore it is not legal.

To select edges for collapsing, we assign a cost for removing each edge. This cost is the error introduced to the mesh by deforming faces and moving vertices after contracting the edge. As presented by Garland [11], this error can be approximated with error quadrics. Each vertex has a 4×4 matrix Q assigned to it, the error ϵ at vertex $v = [v_x, v_y, v_z, 1]$ (in homogeneous coordinates) is given by $\epsilon = vQv^T$. After a collapse $(v_1, v_2) \rightarrow v_3$ the new matrix Q_3 can be derived by the additive rule $Q_3 = Q_1 + Q_2$.

The initial error quadrics for a vertex v are derived from the supporting plane equations of the faces adjacent to v. The fundamental error quadric K_p can be used to find the distance of a point in space to plane $p = [a, b, c, d]^T$. By adding multiple quadrics together, it is possible to represent a set of planes in a single matrix Q. The fundamental error quadric can be calculated as follows:

$$K_p = pp^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bd & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

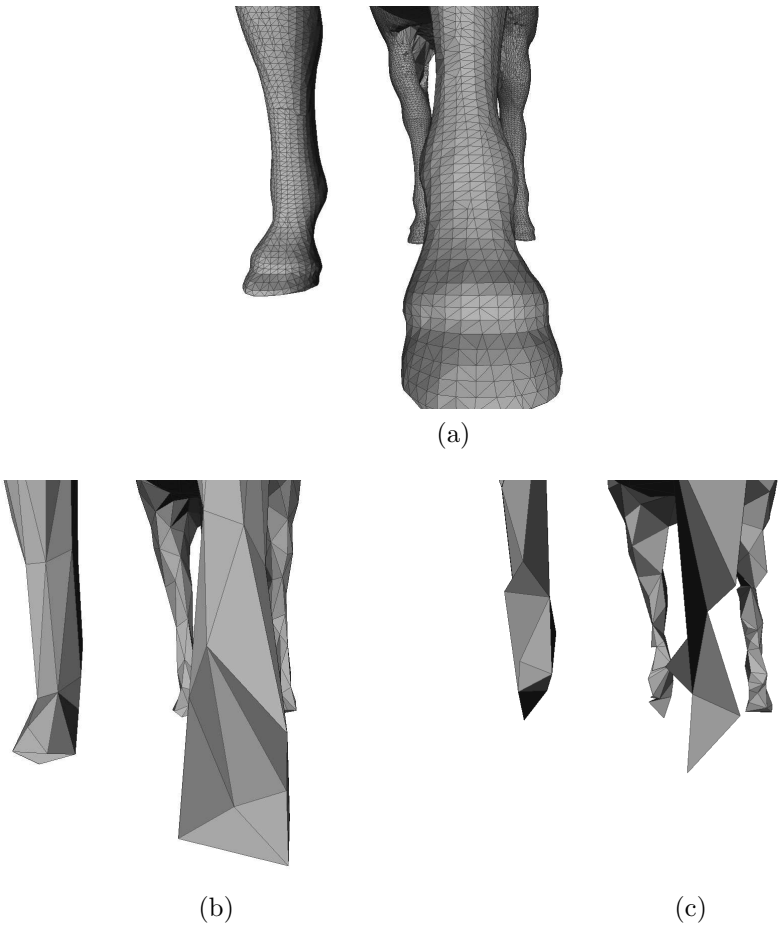
Thus, the initial error quadrics matrix Q_i for vertex v_i is obtained by adding the fundamental error quadrics of the supporting planes of all faces adjacent to v_i. Furthermore, the error of removing an edge is measured as the summation of the distances between the placement vertex and the planes supporting both vertices of the edge.

For each grid cell, candidate edges are ranked by the quadric error metrics and a pair $(edge\_id, cost)$ is inserted into the cell's respective heap, with lowest cost on top. Using multiple priority queues allows parallel edge collapse, given that each thread can simultaneously work on a different cell. After a contraction is finished, the edges adjacent to the affected vertices must have their costs updated.

4.3.3 Prevention of Intersections

To ensure that an edge collapse operation does not produce non-existing intersections, which would lead to a topologically inconsistent result, it is necessary to verify whether the operation is legal, that is, if the edge contraction will not produce intersections. Thus, a test is performed to determine if a contraction produces intersections; if the test is positive, that contraction is discarded and the next edge is popped from the heap. Since an edge contraction only affects simplices connected to their endpoints, it is enough to restrict the search for intersections to the bounding box formed by such simplices, that is, the contraction of edge e will only change faces and other edges adjacent to e, therefore it is possible to establish a boundary for the intersection search.

It is only necessary to test intersections between vertices and simplicies, and between edges, since the collision between triangles or between an edge and a triangle is always preceded by a collision of the former types. Figure 4.6(b) shows a simplification without self-intersecting faces. Figure 4.6(c) shows a mesh with self-intersecting faces after simplification. Both results were obtained by simplifying the horse model (Figure 4.6(a)) to 1% of the vertices using: (b) *UGSimp*; (c) the GPU method.



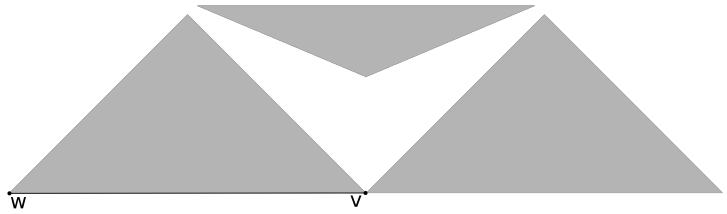
(a)

(b) (c)

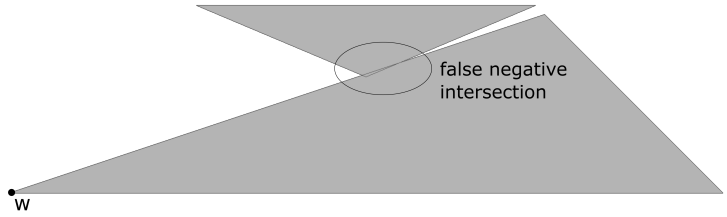
Figure 4.6 – (a) Original mesh horse's feet; (b) Topologically consistent simplified mesh (no self-intersections) by *UGSimp*; (c) Self-intersecting mesh as result of simplification by the GPU method.

When using floating-point numbers intersection tests may fail due to round-off errors. Figure 4.7 shows an example where, even using preemptive tests, the results are inconsistent. Collapsing edge (v, w) will cause an intersection between the resulting faces. However, due to round-off errors, the intersection test gives a false negative, and the contraction is realized.

When an intersection occurs and is not detected, it is called a *false negative*. When an intersection does not occur in reality, but is incorrectly detected, we call it a *false positive*. A false positive test does not imply topological inconsistencies, since it will only prevent a legal collapse from occurring. A false negative, on the other hand, may cause an illegal collapse to be performed, therefore producing intersections. To solve round-off errors arising from the use of floating-point numbers, multiple precision rational numbers, supported by GMP [44], were used when performing intersection tests.



(a)



(b)

Figure 4.7 – (a) A triangular mesh before the contraction of (v, w); (b) After contracting (v, w) an intersection appears but, due to round-off errors, is not detected. The result is topologically inconsistent.

4.4 Results and Discussion

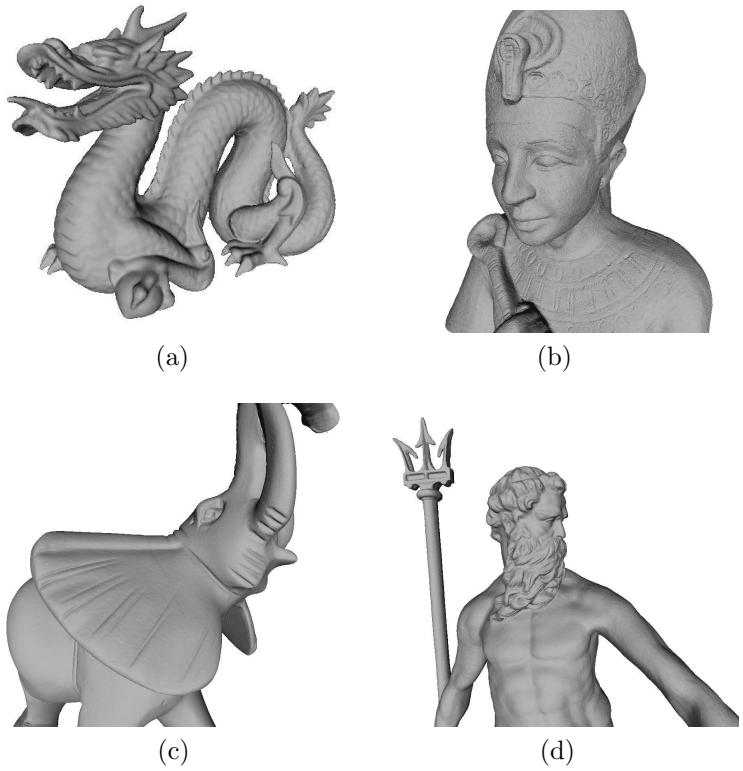
For the experimental evaluation, *UGSimp* was implemented in C++, using the library GMP [44] for multiple precision rational numbers manipulation as a method to avoid round-off errors originated from floating-point arithmetic. Different categories of tests were performed to evaluate the method. First, the execution time and quality of results of our method (*UGSimp*) were compared against other methods from the literature. Then, an artificial dataset was used to induce the occurrence of round-off errors and intersection test failures when using floating-point arithmetic. In addition, the influence of the uniform-grid resolution on the performance was also evaluated. Finally, tests were made to assess the overhead introduced by the use of multiple precision rational numbers and how the method escalates as the number of threads increases. The computer used for running the test has an Intel Xeon CPU E5-2630 with 8 cores and 16 threads, 64 GB of RAM and the GPU GeForce Titan X with 12 GB of global memory.

For the first tests, the processing times of several algorithms were measured on multiple datasets of large sizes: the *Chinese dragon*, *Ramses*, *Elephant* and *Neptune* (datasets 1, 2, 3 and 4, respectively), see Figure 4.8 and Table 4.1. Sequential

algorithms included in the tests were Lindstrom-Turk's memoryless simplification (LT), provided by the CGAL package [56] and Quadric Error Metrics (QEM) simplification provided by MeshLab [57]. The parallel method (GPU) developed by Papageorgiou [22] was also used for comparison.

Table 4.1 – Information of the datasets used in the tests for UGSimp.

Dataset	Vertices	Edges	Faces
Dragon	437645	1309256	871414
Ramses	826266	2478792	1652528
Elephant	1512290	4536882	3024588
Neptune	2003932	6011808	4007872



(a) (b) (c) (d)

Figure 4.8 – Snapshots of the datasets: (a) Chinese dragon; (b) Ramses; (c) Elephant; (d) Neptune.

The algorithms mentioned above were executed on the four different meshes, reducing each of the meshes to 10% of the original model. Table 4.2 shows the wallclock-time taken by each method to perform simplification of the meshes. Our method, *UGSimp*, was able to simplify meshes faster than the GPU method. Even though GPU devices can run a larger number of threads than a multi-core processor, its *data parallel* architecture makes it less efficient due to the occurrence of numerous divergent branches during the process of simplification. Furthermore, the GPU method was unable to simplify datasets with more than 2 million vertices, since it requires the entire model to be stored in the GPU's memory.

Table 4.2 – Processing time of algorithms (in milliseconds) for simplifying each of the datasets to 10% of the original amount of vertices.

Dataset	1	2	3	4
Size($\times 10^3$)	437	826	1500	2000
LT	61197	139494	263684	352938
QEM	8685	25068	42253	67970
GPU	1527	5010	9645	11871
UGSimp	563	2402	5152	7649

Table 4.3 – Mean Hausdorff distances of the results obtained by each method as a percentage of the models bounding box diagonal. Higher values indicate more deviation from the original mesh.

Dataset	1	2	3	4
LT	0.004	0.049	0.015	0.013
QEM	0.005	0.148	0.073	0.070
GPU	0.007	0.211	0.132	0.130
UGSimp	0.002	0.125	0.051	0.052

The quality of the results was evaluated by the mean Hausdorff distance on each result. The Hausdorff distance of a vertex v and a mesh M is the shortest distance between v and a vertex from M. These values were obtained by using the MeshLab tool, which provides a method to compute Hausdorff distances between two meshes. The results are given as a proportion of the model's bounding box diagonal, the higher the results, the more distant from the original model it is.

Despite using a local criterion for removing edges, *UGSimp* presents low values for the mean Hausdorff distances for most datasets, as shown by Table 4.3. The reason is that using the uniform grid structure forces short edges to be removed first. Removing short edges provoke a smaller displacement of simplices and introduces less error to the overall mesh. Figure 4.9 presents a color scale for Hausdorff distances on both *UGSimp* and GPU methods. The scale ranges from red, representing points with low Hausdorff distances, therefore less deviation from the original mesh, to blue, representing the highest Hausdorff distances among both simplified meshes. One can notice that the error in Figure 4.9(a) is uniformly distributed throughout the mesh, whereas in Figure 4.9(b) there are regions with error peaks (blue areas). This may occur because some independent regions of the GPU algorithms are reduced less than others, whereas on *UGSimp* all grid cells are uniformly reduced. Therefore, the error on Figure 4.9(a) is uniformly distributed, and on Figure 4.9(b) it is condensed in some regions. For another visual comparison of results, Figure 4.10 shows the original horse model (dataset 2) and the results after simplifying it to 1% using methods QEM, *UGSimp* and GPU. In this case, GPU presented the worst visual quality, with faces having their orientation flipped in some parts of the model.

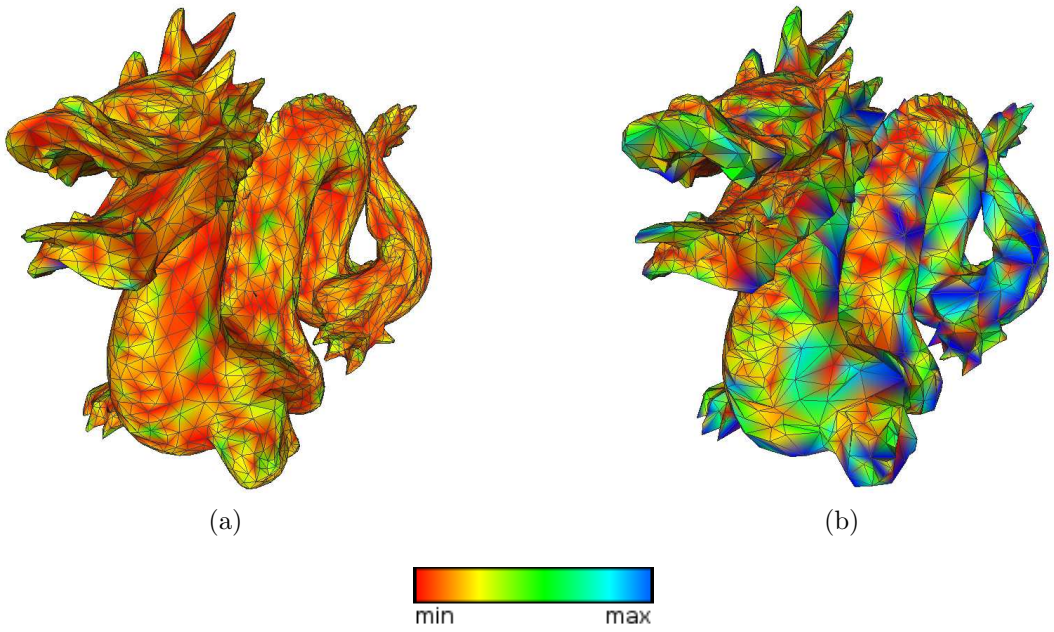
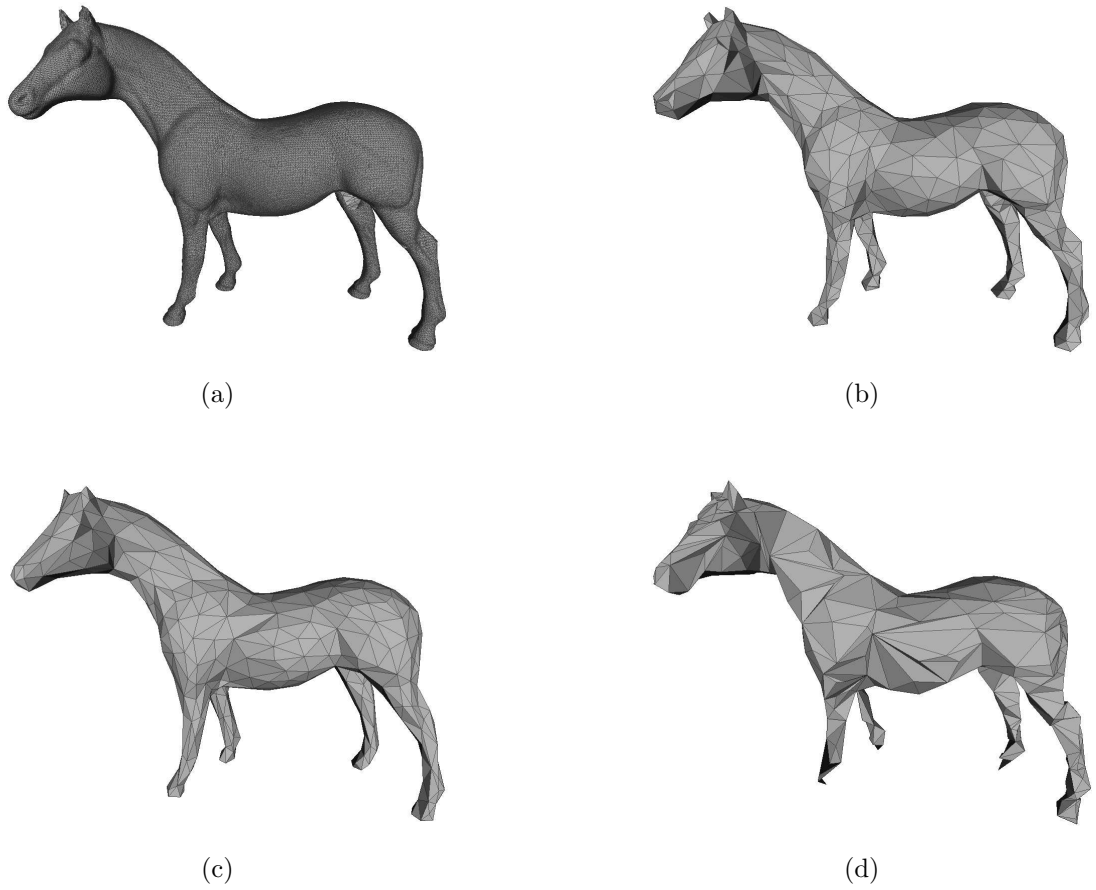


Figure 4.9 – Comparison of Hausdorff distances for (a) *UGSimp* ; and (b) the GPU method. Red colored regions have minimum Hausdorff distance value, blue areas have maximum Hausdorff distance value.

The next category of tests used an artificial dataset generated so that the test for intersection failed when using floating-point arithmetic. This would cause the method to produce topologically inconsistent result. Then, we replaced the floating-point tests for multiple precision rational numbers. As expected, the intersection tests no longer fail and produce a topologically consistent result, but the execution time increased. Table 4.4 and Figure 4.11 show the simplification times for *UGSimp* when using different numbers of threads and rational representation for intersection comparisons. The speedup on the simplification times was up to 6× when using 16 threads.

The highest increase on the speedup happened when changing from 4 to 8 threads. The difference in the running time when using 8 or 16 threads is considerably small. The performance increase on the initialization stage was not as much as the increase on simplification. This can be explained by the fact that there are many steps that cannot be carried out in parallel during initialization, therefore, increasing the number of threads did not affect the performance of such steps. It is worth mentioning that the initialization times for using floating-point did not differ from the rational representation because the rational numbers are only used for intersection tests during the edge collapse step, not affecting the initialization of data structures.

The performance of *UGSimp* is affected by the initial resolution of the uniform grid. As the resolution increases, cells become smaller and each cell has fewer candidate edges. A low resolution, on the other hand, causes cells to have more



(a) (b)

(c) (d)

Figure 4.10 – Visual comparison of simplification to 1% of the original model. (a) Original model; (b) QEM; (c) *UGSimp* ; (d) GPU.

Table 4.4 – Time (in ms) taken by *UGSimp* for simplifying the four datasets using floating-point and rational numbers using different number of threads.

	Dataset	1		2		3		4	
		Float	Rational	Float	Rational	Float	Rational	Float	Rational
Threads	1	2292	3937	8321	65041	18031	99637	24864	141856
	2	1101	3524	5275	56089	11266	94029	16886	125540
	4	903	2633	4239	45071	9922	89260	12953	120023
	8	667	1605	2660	31829	6933	70314	8660	86562
	16	598	1520	2135	22834	5152	54299	7609	71555

candidate edges, but a smaller number of cells means that the parallelism may not be used at its best. For instance, if the system is using 16 threads but there are only 8 cells on the grid (on a $2 \times 2 \times 2$ grid), half of the threads are not used during the process. The choice for a uniform grid resolution was determined experimentally so that it performed as fast as possible.

Tests were executed using different grid resolutions to verify the algorithm's performance when running with different number of cells. The tests consisted of simplifying to 50% all four datasets using different grid resolutions. The results on Table 4.5 show that *UGSimp*'s best performance was achieved by using a grid

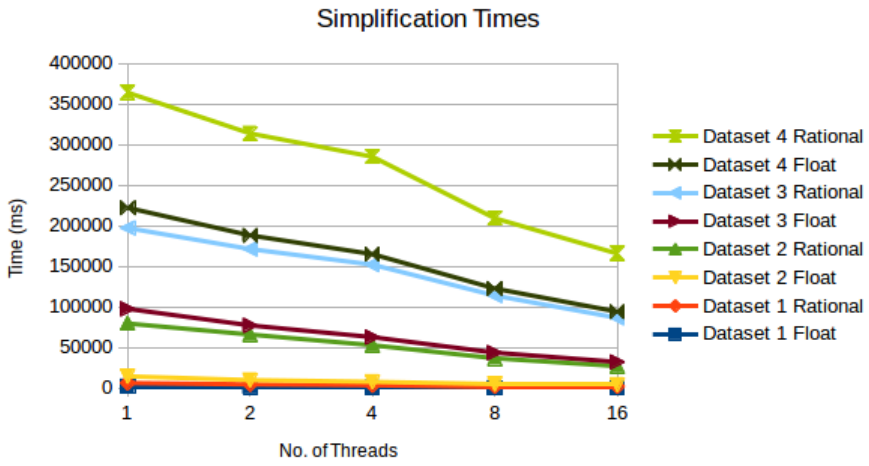


Figure 4.11 – Times (in ms) to simplify the four datasets using floating-point and rational numbers.

Table 4.5 – Execution times (in ms) for different grid-resolutions. On the four datasets, a resolution of 8 performed best.

Grid-Resolution	1	2	4	6	8	10	12	14	16
Dataset 1	1702	366	356	350	333	485	506	540	712
Dataset 2	4085	1516	1502	1311	1082	1275	1522	1874	1965
Dataset 3	7419	3443	3423	3230	2701	3503	3977	4221	4458
Dataset 4	11004	6611	4774	4710	3670	4933	5632	5978	6322

resolution of 8, that is, by dividing the model into $8 \times 8 \times 8$ cells. The execution times decrease when increasing the grid resolution from 1 to 8, however, increasing the grid-resolution to a value higher than 8 causes the performance to drop. At high grid resolution, cells become smaller and less *active* cells are present, therefore some cells may not contain a significant amount of edges to be simplified and a considerable amount of time is wasted by computing its edges and creating its priority queue.

4.5 Conclusions and Future Works

Considering the importance of mesh simplification over large triangular meshes and the importance of maintaining topological consistency to simplified results, we developed *UGSimp*, a method for simplification that does not introduce intersection or self-intersections to the simplified model by performing preemptive tests to determine if a mesh reduction operation would cause topological inconsistencies. Moreover, we investigated the occurrence of round-off errors on intersection tests, which would cause the algorithm to introduce intersections/self-intersections to the results. In order to prevent such errors, intersection tests were performed using multiple precision rational numbers. To compensate for overhead introduced when using multiple precision rational numbers, a parallel strategy for simplification was devel-

oped. Using a uniform grid structure, the mesh was divided into independent cells and parallel edge collapses were performed. Tests showed that *UGSimp* is faster than the most used methods, and the fastest GPU-parallel method found on the literature, when using floating-point arithmetic.

Although *UGSimp* uses a local cost strategy for reducing a mesh, the simplified models produced by it were very similar to, or even better than, those produced by the serial methods, which use a global cost strategy. The mean Hausdorff distances, on models simplified by *UGSimp* , on most cases, were lower than the other methods, presenting good-quality results.

For future improvements, other edge removal policies may be introduced to *UGSimp* such as visibility preserving simplification on open height fields. A GPU implementation of *UGSimp* may use dynamic parallelism for further performance improvement.

5 General Conclusions and Future Works

Maintaining topological consistency of geometric objects is crucial to ensure that models are a reliable representation of the real world. In this work, examples of topological inconsistencies arising from the process of simplification were presented. It was shown that such problems could be avoided by performing tests to prevent elements to intersect due to the removal of a vertex or an edge. However, the use of floating-point arithmetic is subjected to round-off errors and can make preemptive tests to fail, producing inconsistent results nevertheless. Rounding and ϵ-tweaking techniques may be an easy way to avoid round-off errors in many cases, but they always activate the *rounding to zero* property, besides, finding an ϵ value that works for every case is a difficult task. Exact Arithmetic is very robust since it does not activate any category of rounding and does not require any user-defined parameters. Parallel computing was used to increase the performance in both *EPLSimp* and *UGSimp* methods. The major challenge was to divide the problem into parts that could be processed in parallel.

Guaranteeing topological consistency means that some objects cannot be simplified to every level of detail. For instance, if every vertex on a polyline causes an inappropriate intersection, none of them would be a candidate for removal, therefore the algorithm would output the original line. This behavior could be seen on Chapter 2: in some cases, *TopoVW* removed fewer points than the original VW algorithm. This was expected because the original VW algorithm does not attempt to prevent topological inconsistencies, which implies that even though VW removed more points, the simplified models presented intersections and changing of sidedness.

Results in Chapters 3 and 4 showed how 2D and 3D simplification algorithms, including *TopoVW*, can fail and produce topologically consistent results when using floating-point representation. Because of rounding errors, preemptive tests gave false positive or false negative results, creating intersections. It was possible to completely avoid round-off errors by using arbitrary precision rational numbers. The overhead introduced by this approach was considerably reduced with parallel computing. The speedup of both *EPLSimp* and *UGSimp* algorithms was about 3 to 4$\times$ at 16 threads.

Even though some authors argue that maintaining topological consistency in simplification should not be of one's concern [21], it is critical to preserve features of the original model and avoid creating intersections. Inappropriate intersections can compromise the actuality of the model and affect results of other algorithms applied to the mesh, such as point location and visibility calculations.

For future improvements, novel metrics can be introduced to both *EPLSimp* and *UGSimp*, such as Ramer-Douglas-Peucker metric for line simplification. These methods can also be extended to perform simplification of models with color and

texture. Feature-preserving criteria can be added to *UGSimp*, such as visibility-preserving metrics and view-dependent simplification. Additionally, both methods can make use of the massive parallel processing power of GPUs. Current GPU architecture allows use of dynamic parallelism which could provide an enormous performance gain, especially in *UGSimp*, where the processing of each individual cell could make use of parallel computing, such as updating the edge costs after a contraction.

Bibliography

1 KETTNER, L. et al. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, Elsevier, v. 40, n. 1, p. 61–78, 2008.

2 ALLAN, R. J. et al. *High-performance computing*. [S.l.]: Springer Science & Business Media, 2012. 71–72 p.

3 PACHECO, P. *An introduction to parallel programming*. [S.l.]: Elsevier, 2011. 22 p.

4 KIM, Y.-J. et al. Efficient hausdorff distance computation for freeform geometric models in close proximity. *Computer-Aided Design*, Elsevier, v. 45, n. 2, p. 270–276, 2013.

5 MAGALHÃES, S. V. et al. Pinmesh—fast and exact 3d point location queries using a uniform grid. *Computers & Graphics*, Pergamon, v. 58, p. 1–11, 2016.

6 RAMER, U. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, Elsevier, v. 1, n. 3, p. 244–256, 1972.

7 DOUGLAS, D. H.; PEUCKER, T. K. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, University of Toronto Press, v. 10, n. 2, p. 112–122, 1973.

8 VISVALINGAM, M.; WHYATT, J. Line generalisation by repeated elimination of points. *The Cartographic J.*, p. 46–51, 1993.

9 SCHROEDER, W. J.; ZARGE, J. A.; LORENSEN, W. E. Decimation of triangle meshes. In: ACM. *ACM Siggraph Computer Graphics*. [S.l.], 1992. v. 26, n. 2, p. 65–70.

10 HOPPE, H. et al. Mesh optimization. In: ACM. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. [S.l.], 1993. p. 19–26.

11 GARLAND, M.; HECKBERT, P. S. Surface simplification using quadric error metrics. In: ACM PRESS/ADDISON-WESLEY PUBLISHING CO. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. [S.l.], 1997. p. 209–216.

12 UEBERHUBER, C. W. *Numerical computation 1: methods, software, and analysis*. [S.l.]: Springer Science & Business Media, 2012. 79 p.

13 VUIK, K. *Computer Arithmetic Tragedies*. 2016. Available from Internet: ⟨http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html⟩.

14 GOODRICH, M. T. et al. Snap rounding line segments efficiently in two and three dimensions. In: ACM. *Proceedings of the thirteenth annual symposium on Computational geometry*. [S.l.], 1997. p. 284–293.

15 SHEWCHUK, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, Springer, v. 18, n. 3, p. 305–363, 1997.

16 LI, C. *Exact geometric computation: theory and applications*. 82–89 p. Thesis (PhD) — New York University, 2001.

17 LI, C.; PION, S.; YAP, C.-K. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming*, Elsevier, v. 64, n. 1, p. 85–111, 2005.

18 GRUPPI, M. G. et al. An efficient and topologically correct map generalization heuristic. In: *ICEIS (1)*. [S.l.: s.n.], 2015. p. 516–525.

19 GRUPPI, M. G. et al. Using rational numbers and parallel computing to efficiently avoid round-off errors on map simplification. In: *Proceedings of the XVI Brazilian Symposium on GeoInformatics*. [S.l.: s.n.], 2015. p. 162–173.

20 FRANKLIN, W. R. et al. Uniform grids: A technique for intersection detection on serial and parallel machines. In: CITESEER. *Proceedings of Auto Carto*. [S.l.], 1989. v. 9, p. 100–109.

21 ROSSIGNAC, J. Simplification and compression of 3d scenes. In: CITESEER. [S.l.], 1997. p. 100–109.

22 PAPAGEORGIOU, A.; PLATIS, N. Triangular mesh simplification on the gpu. *The Visual Computer*, Springer, v. 31, n. 2, p. 235–244, 2015.

23 JOAO, E. *Causes and Consequences of map generalization*. [S.l.]: CRC Press, 1998. 30 p.

24 MACKANESS, W. A.; RUAS, A.; SARJAKOSKI, L. T. *Generalisation of geographic information: cartographic modelling and applications*. [S.l.]: Elsevier, 2011. 57 p.

25 SHEA, K. S.; MCMASTER, R. B. Cartographic generalization in a digital environment: When and how to generalize. In: *Proceedings of AutoCarto*. [S.l.: s.n.], 1989. v. 9, p. 56–67.

26 RUAS, A.; LAGRANGE, J.-P. Data knowledge and modelling for generalization. *MÜLLER, JC, LAGRANGE, JP, WEIBEL, R. GIS and generalization: metodology and practice. GISDATA I, serie editors*, p. 73–90, 1995.

27 ESTKOWSKI, R.; MITCHELL, J. S. Simplifying a polygonal subdivision while keeping it simple. In: ACM. *Proceedings of the seventeenth annual symposium on Computational geometry*. [S.l.], 2001. p. 40–49.

28 MAGALHÃES, S. V. de et al. Fast map generalization heuristic with a uniform grid. In: ACM. *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. [S.l.], 2014. p. 613–616.

29 WARE, J. M.; JONES, C. B.; THOMAS, N. Automated map generalization with multiple operators: a simulated annealing approach. *International Journal of Geographical Information Science*, v. 17, p. 743–769, 2003.

30 KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, n. 4598, p. 671–680, 1983.

31 SAALFELD, A. Topologically consistent line simplification with the douglas-peucker algorithm. *Cartography and Geographic Information Science*, v. 26, p. 7–18, 1999.

32 SILVA, A. C. D.; WU, S.-T. Preserving coincidence and incidence topologies in saalfeld's polyline simplification algorithm. In: *GeoInfo*. [S.l.: s.n.], 2005. p. 107–121.

33 LI, L. et al. An algorithm for fast topological consistent simplification of face features. *Journal of Computational Information Systems*, v. 9, n. 2, p. 791–803, 2013.

34 ROBERGE, J. A data reduction algorithm for planar curves. *Computer Vision, Graphics and Image Processing*, v. 29, n. 2, p. 168–195, 1985.

35 DETTORI, G.; FALCIDIENO, B. An algorithm for selecting main points on a line. *Computers & Geosciences*, v. 8, n. 1, p. 3–10, 1982.

36 ACM SIGSPATIAL Cup 2014. *GIS Cup Sample Data*. 2014. Mypages.iit.edu/∼xzhang22/GISCUP2014/ (accessed on 09/01/2014).

37 IBGE: Instituto Brasileiro de Geografia e Estatistica. *Malhas digitais dos municipios Brasileiros*. 2014. Geoftp.ibge.gov.br/malhas_digitais/municipio_2007/ (accessed on 09/01/2014).

38 United States Census. *US Counties Shapefiles*. 2014. Www.census.gov/cgi-bin/geo/shapefiles2013/main (accessed on 09/01 /2014).

39 JIANG, B.; LIU, X.; JIA, T. Scaling of geographic space as a universal rule for map generalization. *Annals of the Association of American Geographers*, Taylor & Francis, v. 103, n. 4, p. 844–855, 2013.

40 GOLDBERG, D. What every computer scientist should know about floating-point aritmetic. *ACM Computing Surveys*, v. 23, n. 1, p. 5–48, 1991.

41 HOBBY, J. D. Practical segment intersection with finite precision output. *Computational Geometry*, Elsevier, v. 13, n. 4, p. 199–214, 1999.

42 BERG, M. de; HALPERIN, D.; OVERMARS, M. An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, Elsevier, v. 36, n. 3, p. 159–165, 2007.

43 HERSHBERGER, J. Stable snap rounding. *Computational Geometry*, Elsevier, v. 46, n. 4, p. 403–416, 2013.

44 GRANLUND, T.; TEAM the GMP development. *The GNU Multiple Precision Arithmetic Library*. 2014. Available from Internet: ⟨http://gmplib.org/⟩.

45 HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. 2008.

46 ZINSMAIER, M. et al. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, IEEE, v. 18, n. 12, p. 2486–2495, 2012.

47 DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering*, v. 5, n. 1, p. 46–55, 1998.

48 NICKOLLS, J. et al. Scalable parallel programming with cuda. *Queue*, v. 6, n. 2, p. 40–53, 2008.

49 COHEN, J. et al. Simplification envelopes. In: ACM. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. [S.l.], 1996. p. 119–128.

50 RONFARD, R.; ROSSIGNAC, J. Full-range approximation of triangulated polyhedra. In: WILEY ONLINE LIBRARY. *Computer Graphics Forum*. [S.l.], 1996. v. 15, n. 3, p. 67–76.

51 CAMPOMANES-ÁLVAREZ, B. R.; CORDÓN, O.; DAMAS, S. Evolutionary multi-objective optimization for mesh simplification of 3d open models. *Integrated Computer-Aided Engineering*, IOS Press, v. 20, n. 4, p. 375–390, 2013.

52 SHONTZ, S. M.; NISTOR, D. M. Cpu-gpu algorithms for triangular surface mesh simplification. In: *Proceedings of the 21st international meshing roundtable*. [S.l.]: Springer, 2013. p. 475–492.

53 STAADT, O. G.; GROSS, M. H. Progressive tetrahedralizations. In: IEEE. *Visualization'98. Proceedings*. [S.l.], 1998. p. 397–402.

54 GUMHOLD, S.; BORODIN, P.; KLEIN, R. Intersection free simplification. *International Journal of Shape Modeling*, World Scientific, v. 9, n. 02, p. 155–176, 2003.

55 KONYUKHOV, A.; SCHWEIZERHOF, K. *Computational Contact Mechanics: geometrically exact theory for arbitrary shaped bodies*. [S.l.]: Springer Science & Business Media, 2012. v. 67. 75 p.

56 CGAL. *CGAL User and Reference Manual*. 4.8.1. CGAL Editorial Board, 2016. Available from Internet: ⟨http://doc.cgal.org/4.8.1/Manual/packages.html⟩.

57 CIGNONI, P.; CORSINI, M.; RANZUGLIA, G. Meshlab: an open-source 3d mesh processing system. *Ercim news*, v. 73, n. 45-46, p. 6, 2008.

Appendices

A Additional Tests

This chapter presents results for measurements of the execution times of *UGSimp*, GPU, and QEM methods, shown on Chapter 4. The tests were performed using the same system setup and datasets as in Chapter 4, although different simplification targets were used: 75%, 50%, 25% and 10%. Execution times (in ms) are shown on Tables A.1 to A.4.

Table A.1 – Times (in ms) for simplification of dataset 1 (*Chinese dragon*).

Dataset 1	75%	50%	25%	10%
UGSimp	137	290	384	567
GPU	399	551	850	1218
QEM	1757	2583	4345	6214

Table A.2 – Times (in ms) for simplification of dataset 2 (*Ramses*).

Dataset 2	75%	50%	25%	10%
UGSimp	458	974	1331	2023
GPU	641	1111	1713	2425
QEM	7400	9539	13299	17851

Table A.3 – Times (in ms) for simplification of dataset 3 (*Elephant*).

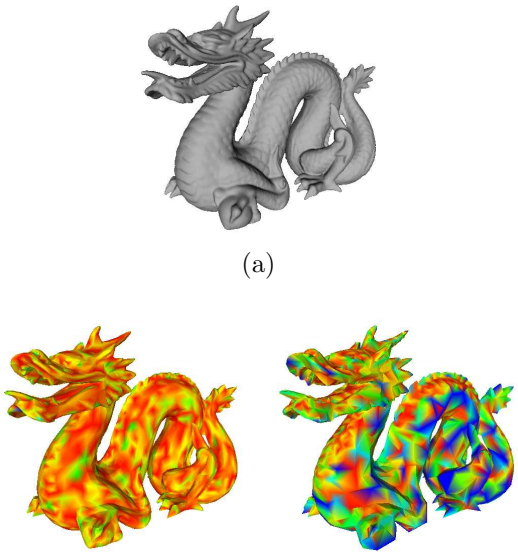
Dataset 3	75%	50%	25%	10%
UGSimp	901	2370	3290	5235
GPU	1149	2584	3648	6954
QEM	12750	16616	23889	30982

Table A.4 – Times (in ms) for simplification of dataset 4 (*Neptune*).

Dataset 4	75%	50%	25%	10%
UGSimp	1009	2998	4423	7662
GPU	1566	3648	4912	9192
QEM	21427	23209	32975	52008

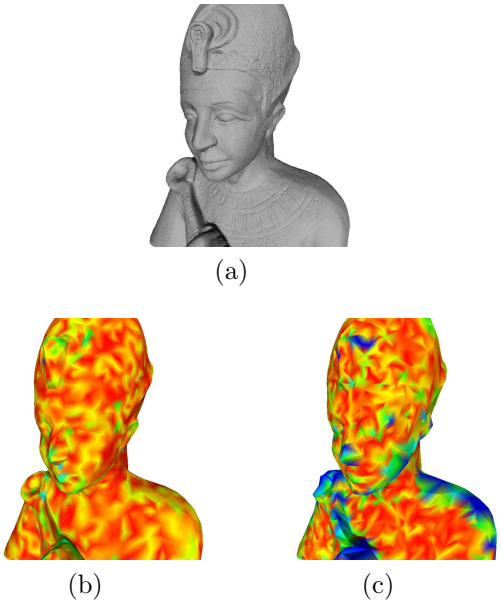
B Hausdorff Distance Scales

This chapter presents comparisons between *UGSimp* and the GPU method. The color scale indicates the Hausdorff distance value, with red representing the minimum value (usually zero) and blue representing the maximum Hausdorff distance value. Figures B.1 to B.4 present a color scale for simplifications of 50% of the original datasets.



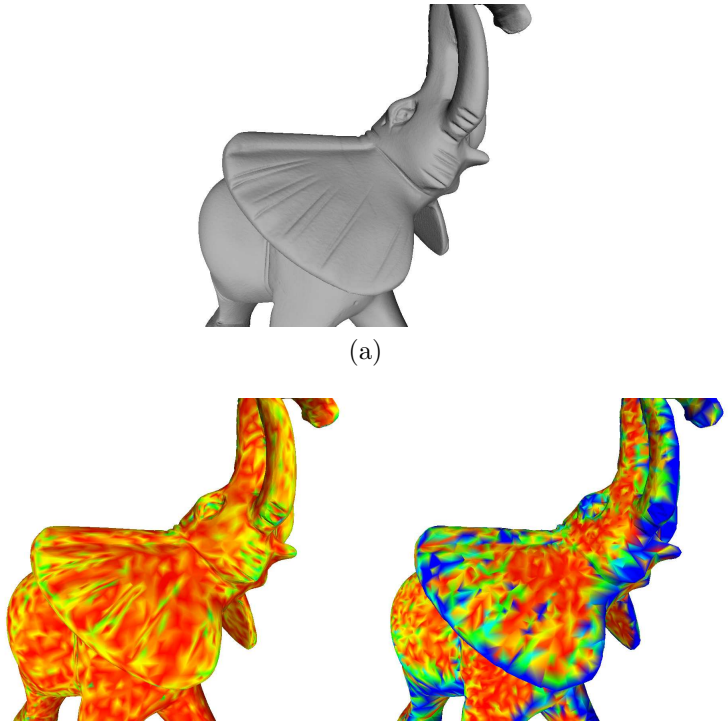
(a) (b) (c)

Figure B.1 – Original *Chinese dragon* model (a) and its simplified versions by: (b) *UGSimp*; (c) GPU.



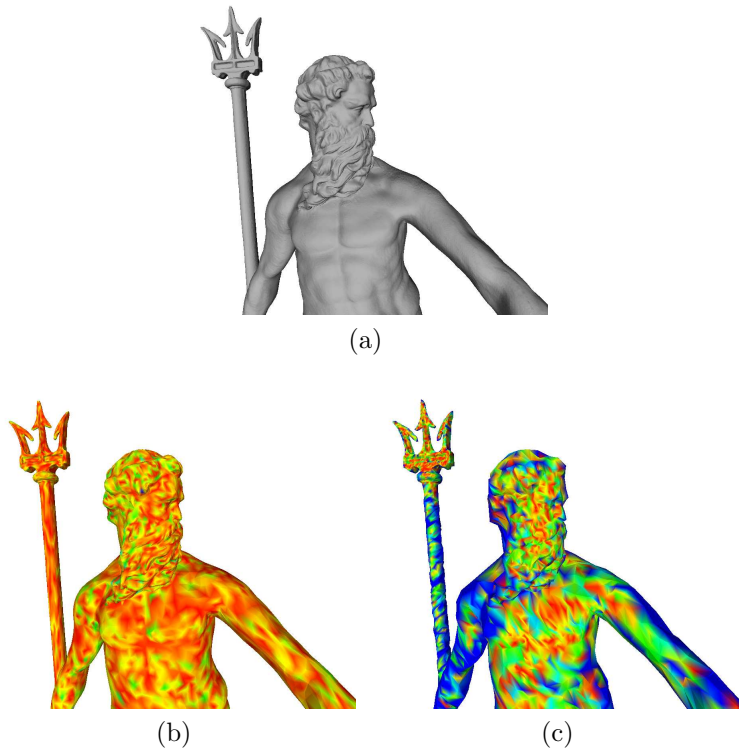
(a) (b) (c)

Figure B.2 – Original *Ramses* model (a) and its simplified versions by: (b) *UGSimp*; (c) GPU.



(a) (b) (c)

Figure B.3 – Original *Elephant* model (a) and its simplified versions by: (b) *UGSimp*; (c) GPU.



(a) (b) (c)

Figure B.4 – Original *Neptune* model (a) and its simplified versions by: (b) *UGSimp*; (c) GPU.

C Grid Resolution Experiments

This chapter presents the comparison of execution times for *UGSimp* using different grid-resolutions, shown by Table C.1 and Figure C.1. Table C.2 presents the number of active cells for different resolutions. A cell is *active* if it contains edges can be simplified. As example, for the Dragon model, for the resolution of 16, there are 4096 but only 1188 active cells. This means that only 1188 contain edges that can be simplified. The remaining cells are not useful.

Table C.1 – Execution times (in ms) of UGSimp for multiple grid resolutions.

Grid Resolution	1	2	4	6	8	10	12	14	16
Dataset 1	1702	366	356	350	333	485	506	540	712
Dataset 2	4085	1516	1502	1311	1082	1275	1522	1874	1965
Dataset 3	7419	3443	3423	3230	2701	3503	3977	4221	4458
Dataset 4	11004	6611	4774	4710	3670	4933	5632	5978	6322

Table C.2 – Number of cells that can be processed (*active*) when using different grid resolutions. Column Total indicates the total amount of cells for that resolution.

Grid Resolution	2		8		16	
Cells	Active	Total	Active	Total	Active	Total
Dragon	8	8	265	512	1188	4096
Ramses	4		105		187	
Elephant	4		72		291	
Neptune	4		71		110	

Figure C.1 – Grid Resolution vs Execution Time (in ms)