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

STUDY AND EVALUATION OF ALGORITHMS TO GENERATE POLYGON MESHES

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

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

CRISTIAN ANDRÉS PARRA OYARCE

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
CLAUDIO LOBOS YAÑEZ
IVAN SIPIRÁN MENDOZA
GILBERTO GUTIÉRREZ RETAMAL

Este trabajo ha sido parcialmente financiado por: Fondecyt 1181506

SANTIAGO DE CHILE
2021

Resumen

Actualmente, la diversificación de técnicas de modelado geométrico a través de mallas de polígonos en diferentes ramas científicas como la neurociencia, la ingeniería mecánica y la astrofísica, hace que nos interese su estudio. Las mallas de polígonos permiten modelar geometrías complejas y la simulación de fenómenos complejos o comportamientos de objetos, por lo que también se pueden utilizar en medicina, para extraer descripciones geométricas de imágenes digitales en procesos de resonancia magnética computarizada. Además, los métodos de Elementos Finitos (PFEM) y el método de Elementos Virtuales (VEM) son cada vez más populares hoy en día debido a su flexibilidad en el modelado de dominios complejos, una base matemática importante, eficiencia y precisión en la solución obtenida para algunos problemas complejos. Las mallas de polígonos más utilizadas son las basadas en el diagrama de Voronoi porque se pueden obtener fácilmente a partir de la triangulación de Delaunay.

En este trabajo de tesis estudiamos y desarrollamos diferentes enfoques para generar mallas de polígonos y compararlas utilizando diferentes métricas de calidad. Las mallas iniciales se obtienen usando (i) la técnica estándar de quadtree, (ii) usando quadtree con un algoritmo de división de puntos arbitrarios (incluyendo aleatorización) (iii) kd-trees. Los elementos se pueden refinar en puntos de borde arbitrarios hasta que cada elemento cumpla con algunos criterios de calidad especificados por el usuario, sí se generan y comparan mallas que satisfacen los requisitos.

Las diferentes estrategias para generar mallas de polígonos se implementaron como una Aplicación Web, desarrollada utilizando tecnologías actuales para los lenguajes Javascript y Typecript, como React y Node.js, con el propósito de lograr una interfaz de usuario interactiva en tiempo real para especificar métricas de calidad y el proceso actual de la malla que se está evaluando. En particular, la aplicación permite ver cómo se comportan el quadtree y el KD-tree en tiempo real en términos de la división del plano y la posterior generación de polígonos más pequeños.

La aplicación web se modeló utilizando patrones de diseño para obtener un software fácil de extender al agregar nuevas estrategias para generar mallas iniciales, algoritmos de refinamiento, recorte de polígonos, división de regiones y diferentes métricas de calidad. Esta aplicación está orientada a explorar y evaluar nuevos algoritmos y métricas de calidad pero no a generar implementaciones óptimas con respecto a los tiempos de ejecución.

Según los resultados obtenidos en nuestros experimentos, las mallas iniciales generadas por KD-trees se generan más rápidamente que las creadas por quadtrees, obteniendo un menor número de elementos y de mejor calidad. En comparación con Triangle, obtuvimos mejores resultados al refinar dentro de una región elegida por el usuario, obteniendo menos polígonos y un mayor número de puntos. Los resultados muestran ser buenos para la aplicación de simulaciones con métodos matemáticos como VEM, ya que el número de polígonos es menor, pero sigue cumpliendo los criterios de calidad impuestos por el usuario en la región de interés. En cuanto a las métricas, cuando se restringe el área máxima a un número determinado, los polígonos generados por un quadtree utilizando el algoritmo de Splitting Longest Edge obtienen mejores métricas en ER y CR que Triangle. Por otro lado, cuando se utiliza un KD-tree las métricas son todas más bajas en nuestros algoritmos excepto en CR utilizando un refinamiento de quadtree.

Abstract

Currently, the diversification of geometric modeling techniques through polygon meshes in different scientific branches such as neuroscience, mechanical engineering and astrophysics, makes us interested in their study. Polygon meshes allow the modeling complex geometries and the simulation of complex phenomena or object behaviours, so they can also be used in medicine, for extracting geometric description from digital images in computerized MRI processes. Moreover, the Particle Finite Element Methods (PFEM) and the Virtual Element Method (VEM) methods are each time becoming more popular today due to their flexibility in modeling complex domains, important mathematical basis, efficiency, and accuracy in the obtained solution for some complex problems. The most used polygon meshes are the ones based on the Voronoi diagram because they can be easy obtained from the Delaunay triangulation.

In this thesis work we study and develop different approaches to generate polygon meshes and compare them using different quality metrics. Initial meshes are obtained using (i) the standard quadtree technique, (ii) using quadtree with arbitrary point division algorithm (including randomization) (iii) kd-trees. Elements can be refined at arbitrary edge points until each element fulfills some quality criteria specified by the user. So meshes that satisfy the user requirements are generated and compared.

The different strategies to generate polygon meshes were implemented as a Web Application, developed using current technologies for the Javascript and Typescript languages, such as React and Node.js, with the purpose of achieving a real-time interactive user interface to specify quality metrics and the meshing step being evaluated. In particular, the application makes possible to see how the quadtree and KD-tree behaves in real-time in terms of the division of the plane and the subsequent generation of smaller polygons.

The web application was modeled using design patterns to obtain software easy to extend when adding new strategies to generate initial meshes, refinement algorithms, polygon clipping, division of regions, and different quality metrics. This application is oriented to explore and evaluate new algorithms and quality metrics but not to generate optimal implementations with respect to running times.

According to the results obtained in our experiments, the initial meshes generated by KD-trees are generated faster than those created by quadtrees, obtaining a smaller number of elements and of better quality. Compared to Triangle, we obtained better results when refining within a region chosen by the user, obtaining fewer polygons and a higher number of points. The results show to be good for the application of simulations with mathematical methods such as VEM, since the number of polygons is lower, but still meets the quality criteria imposed by the user in the region of interest. Regarding metrics, when the maximum area is restricted to a certain number, the polygons generated by a quadtree using splitting longest edge algorithm obtains better metrics in ER and CR than Triangle. On the other hand, when using a KD-tree the metrics are all lower in our algorithms except in CR using a quadtree refinement.

Este trabajo y todo lo que soy ahora, va dedicado a mi padre Juan Parra y a mi madre Alicia Oyarce.

Todo es gracias a ustedes, las personas más maravillosas, fuertes y hermosas de este mundo, que tuve la fortuna que fueran mis padres.

Agradecimientos

En primer lugar quiero agradecer a toda mi familia, en particular a mis padres Alicia y Juan, quienes han sido mis modelos a seguir y que siempre estuvieron ahí apoyándome y dándome todo lo que estuviera en sus manos para llegar hasta acá. Todo cuanto soy ahora se los dedico a ellos, personas de esfuerzo y que merecen mucho más de lo que soy capaz de escribir ahora. De verdad se los agradezco de corazón, los amo con todo mi ser. Este trabajo viene a ser una forma de reconocimiento y una pequeña luz de felicidad en estos tiempos difíciles que hemos tenido que sobrellevar, involucrando temas de salud y una Pandemia mundial, quedando de manifiesto que juntos siempre hemos sido capaces de superar todos los problemas y salir adelante.

Quiero agradecer ahora a mi gran amigo de infancia Leonardo Correa, que desde séptimo básico hasta ahora ha sido mi confidente y quién me ha apoyado en todo lo que he hecho. Gracias por aquellas conversaciones y risas, pasando por el Instituto Nacional, preuniversitario, hasta tiempos recientes en donde nos juntamos a comer algo y conversar de la vida para mantenernos actualizados.

Mi próximo agradecimiento es para mi mejor amiga de la universidad, Valentina Diaz. Quiero agradecer tu alegría, cariño y sobre todo paciencia, porque a pesar de todo siempre has estado ahí para mi. Tú sabes que te quiero mucho y que para mi has sido y espero que sigas siendo un gran apoyo y parte importante de mi vida.

Ahora quiero darles mis cariños y gracias a todos los que conforman el grupo del Hall Sur. En particular a Mariana, Fran, Elisa y Pablo, mis amigos de universidad con los que compartí una gran cantidad de almuerzos y muchas risas desde que eramos mechones. Ustedes saben que no soy mucho de demostrar mi cariño y puedo parecer distante, pero quiero que sepan que son personas realmente importantes para mi, y que los quiero mucho.

En mi paso por la universidad he sido auxiliar y ayudante, por lo que quiero agradecer a todos aquellos que he conocido gracias a dicho trabajo, porque me enseñaron a crecer y a superar un poco la timidez. Espero que a todos aquellos a los que les hice clases, haber contribuido un poco a su enseñanza y haberles sacado una sonrisa en momentos estresantes de la universidad. En particular quiero agradecer a Valentin Muñoz por ser mi compañero de trabajo en los años como auxiliar de programación. Aprendimos mucho, nos reímos, estresamos y disfrutamos, pero por sobre todo entregamos lo mejor de nosotros para dar una clase con cariño y esfuerzo.

Quiero agradecer a mi profesora guía Nancy Hitschfeld, la cual ha tenido una gran paciencia, dedicación y cariño tanto por este trabajo como hacia mi como estudiante. Gracias además porque junto al profesor Francisco Gutierrez me dieron la oportunidad de ser parte del equipo que llevó a cabo varios talleres de Scratch para fomentar el pensamiento computacional en niños y niñas de Chile. Gracias al profesor José Pino por haber confiado en mi para ser por primera vez un profesor auxiliar, y continuar creyendo en mi labor. Y por supuesto gracias a Angélica y a Sandra del DCC, quienes son las que mantienen en orden el DCC y hacen que todo funcione. Además, agradecer al Fondecyt 1181506 por haber financiado parcialmente este trabajo.

Finalmente quiero agradecer a Cornershop por ser mi primer trabajo y contratarme en tiempos difíciles de pandemia y a todas las personas que he conocido ahí, he aprendido mucho de ustedes y me han acogido con mucho cariño. Además dar gracias al Liceo 1 Javiera Carrera, en particular a Mariela Bozo por confiar en mi para dar clases de reforzamiento y al departamento de física completo por su importante y abnegada labor de sacar adelante a las futuras líderes de este país.

Contents

1. Introduction **1**
- 1.1. Objectives . . . 2
  - 1.1.1. Main objectives . . . 2
  - 1.1.2. Specific objectives . . . 2
- 1.2. Methodology . . . 3
- 1.3. Thesis content . . . 3

2. Background **5**
- 2.1. Closed Polygons . . . 5
- 2.2. Polygon Meshes . . . 5
  - 2.2.1. Basic Definitions . . . 5
    - 2.2.1.1. Manifold Meshes . . . 6
    - 2.2.1.2. Orientation of a Mesh . . . 7
  - 2.2.2. Mesh Representations . . . 7
    - 2.2.2.1. Vertex-Vertex . . . 7
    - 2.2.2.2. Face-Vertex . . . 8
    - 2.2.2.3. Halfedge . . . 9
- 2.3. Geometry Algorithms . . . 10
  - 2.3.1. Math Background . . . 10
    - 2.3.1.1. Cross Product . . . 10
  - 2.3.2. Points Related Algorithms . . . 11
    - 2.3.2.1. Point inside Polygon . . . 11
    - 2.3.2.2. Point inside Edge . . . 12
    - 2.3.2.3. Point inside Circle . . . 13
  - 2.3.3. Edge Related Algorithms . . . 13
    - 2.3.3.1. Edge intersection . . . 13
    - 2.3.3.2. Edge intersection with Quadrilateral . . . 15
    - 2.3.3.3. Edge intersection with Circle . . . 16
  - 2.3.4. Polygon Related Algorithms . . . 18
    - 2.3.4.1. Polygon intersection with Circle . . . 18
    - 2.3.4.2. Area of a Polygon . . . 19
    - 2.3.4.3. Centroid of a Polygon . . . 19
    - 2.3.4.4. Getting minimum angle of a Polygon . . . 20
- 2.4. Data Structures . . . 20
  - 2.4.1. Quadtrees . . . 20
  - 2.4.2. Generalized Quadtrees . . . 22
  - 2.4.3. KD-Trees . . . 23

3. Related Work **25**
3.1. Polygon Meshing Algorithms . . . 25
3.1.1. Geometric Meshes based on Quadtrees . . . 25
3.1.2. Geometric Meshes based on Voronoy Cells . . . 26
3.1.3. Geometric Meshes based on Centroid Voronoi Tessellation (CVT) . . . 27
3.2. Mesh Visualizers . . . 27
3.3. Polygon Mesh quality metrics . . . 28
3.3.1. Scale Dependent measures . . . 28
3.3.2. Scale Invariant measures . . . 28
3.4. Polygon Clipping Algorithms . . . 29
3.4.1. Greiner Hormann Algorithm . . . 29
3.4.1.1. Phase 1: Searching intersections . . . 30
3.4.1.2. Phase 2: Marking entry and exit points . . . 30
3.4.1.3. Phase 3: Constructing the clipped polygon . . . 31
3.4.1.4. Disadvantages of the algorithm . . . 32
3.4.2. Extended Greiner Hormann Algorithm . . . 33
3.4.2.1. Classification of Polygonal Chains . . . 33

4. Design **35**
4.1. Mesh Generation Process . . . 35
4.2. Analysis of possible solutions . . . 36
4.3. Proposed software architecture . . . 36
4.3.1. General Architecture . . . 36
4.3.2. General Implementation Choices . . . 37
4.4. Process Design . . . 37
4.4.1. User draws a contour geometry in our Application . . . 38
4.4.2. User uploads Contour Geometry . . . 39
4.4.3. User uploads Mesh by off File . . . 39
4.4.4. Quadtree refining process . . . 41
4.4.5. Quality refining process . . . 42
4.4.6. Quality inspection . . . 44
4.4.7. Exporting . . . 45
4.5. Model Design . . . 46
4.5.1. Geometry and Selector Region . . . 46
4.5.2. Polygon in detail . . . 47
4.5.3. Clipping Algorithms . . . 47
4.5.4. Criteria . . . 48
4.5.5. Quality Refining Algorithms . . . 50
4.5.6. Division Algorithms . . . 50
4.5.7. Mesh . . . 51
4.5.8. HalfEdges . . . 52
4.5.9. Storages . . . 52
4.5.10. Tree . . . 53
4.5.11. React frontend Modeling . . . 54
4.6. Experimental Design . . . 56

5. Implementation **57**
5.1. Mesh Representation . . . 57
5.1.1. Vertices . . . 57
5.1.1.1. Vertex representation . . . 57
5.1.1.2. Vertex Storage . . . 58
5.1.2. Edges . . . 60
5.1.2.1. Edge representation . . . 60
5.1.2.2. Edge Storage . . . 61
5.1.3. Polygons . . . 63
5.1.3.1. Polygon representation . . . 63
5.1.3.2. Polygon Storage . . . 64
5.1.4. Halfedge Connectivity . . . 65
5.1.4.1. Halfedge definition . . . 65
5.1.4.2. Creating halfedges for a polygon mesh . . . 66
5.1.4.3. Applying operations to polygons . . . 68
5.1.4.4. Obtaining the neighbors of a polygon . . . 68
5.1.4.5. Add a Polygon to the Mesh keeping while maintaining connectivity . . . 69
5.2. Web Application Views . . . 70
5.2.1. Integrating PixiJS . . . 71
5.2.2. Geometry creation and initial panel . . . 73
5.2.3. Quadtree Refining Panel . . . 76
5.2.4. Refining Panel . . . 79
5.2.5. Quality Component . . . 81
5.3. Algorithms Implementation . . . 81
5.3.1. Clipping Algorithms . . . 81
5.3.1.1. Sutherland Hodgman Algorithm . . . 82
5.3.1.2. Extended Greiner Hormann Algorithm . . . 84
5.3.1.3. Calculating intersections . . . 87
5.3.1.4. Determining the orientation of polygon chains . . . 89
5.3.1.5. Classifying intersections . . . 91
5.3.1.6. Marking Intersections Chains . . . 92
5.3.1.7. Building the *Entering* and *Exiting* lists . . . 93
5.3.1.8. Traversing the lists . . . 96
5.3.1.9. Complexity Analysis . . . 99
5.3.2. Point insertion in Tree Data Structures . . . 100
5.3.2.1. Half Point Division Algorithm . . . 101
5.3.2.2. Arbitrary Point Division Algorithm . . . 104
5.3.3. Generating the Initial Mesh . . . 107
5.3.3.1. Generating new polygons from a contour geometry . . . 108
5.3.3.2. Obtaining the possible problematic points . . . 108
5.3.3.3. Fixing polygons consistently with problem points . . . 110
5.3.4. Refinement Algorithms . . . 112
5.3.4.1. Tree Refinement . . . 112
5.3.4.2. Splitting Longest Edge . . . 114
5.3.4.3. Centroid . . . 117
5.3.4.4. Centroid with Replication . . . 118

6. Results **121**
6.1. Time Analysis . . . 121
6.1.1. Extended Greiner Hormann Algorithm . . . 121
6.1.2. Initial meshes creation time . . . 122
6.2. Initial Mesh Generation . . . 124
6.2.1. Initial Meshes . . . 124
6.3. Quality improvements . . . 126
6.3.1. Initial Meshes . . . 127
6.3.2. Quality Refinements to Bad Polygons . . . 128
6.3.2.1. Centroid refinement to Initial Meshes . . . 129
6.3.2.2. Centroid Replication refinement to Initial Meshes . . . 130
6.3.2.3. Splitting Longest Edge refinement to Initial Meshes . . . 132
6.4. Successive quality refinements . . . 134
6.4.1. Declaring an upper limit to the area of the polygons . . . 134
6.4.2. Upper limit equal to average mesh area . . . 134
6.4.2.1. Results obtained by centroid algorithm . . . 135
6.4.2.2. Results obtained by centroid replicate algorithm . . . 136
6.4.2.3. Results obtained by Splitting Longest Edge algorithm . . . 137
6.4.2.4. Results obtained by Quadtree Refining algorithm . . . 138
6.4.3. Upper limit equal to one tenth of the average area . . . 138
6.4.3.1. Results obtained by centroid algorithm . . . 139
6.4.3.2. Results obtained by centroid replicate algorithm . . . 140
6.4.3.3. Results obtained by Splitting Longest Edge algorithm . . . 141
6.4.3.4. Results obtained by Quadtree Refining algorithm . . . 142
6.4.4. Declaring an upper limit to the maximum edge length of the polygons 143
6.4.5. Upper limit equal to average edge length . . . 143
6.4.5.1. Results obtained by Splitting Longest Edge algorithm . . . 144
6.4.5.2. Results obtained by Quadtree Refining algorithm . . . 145
6.4.6. Upper limit equal to one half of the average edge length . . . 145
6.4.6.1. Results obtained by Splitting Longest Edge algorithm . . . 146
6.4.6.2. Results obtained by Quadtree Refining algorithm . . . 147
6.5. Comparison of quality metrics between different levels of refinement . . . 148
6.5.1. Results of imposing a upper limit to the maximum area . . . 148
6.5.1.1. Limit equal to the average area of the geometric mesh . . . 149
6.5.1.2. Limit equal to $\frac{1}{10}$ of the average area of the geometric mesh 150
6.5.1.3. Analysis of metric results . . . 150
6.5.2. Results of imposing a upper limit to the maximum length . . . 151
6.5.2.1. Limit equal to the average length of the geometric mesh . . . 151
6.5.2.2. Limit equal to $\frac{1}{2}$ of the average length of the geometric mesh 152
6.5.2.3. Analysis of metric results . . . 152
6.6. Comparing meshes with Triangle . . . 152
6.6.1. Comparing meshes . . . 153
6.6.1.1. Maximum area equal to 1979 area units . . . 153
6.6.1.2. Maximum area equal to 198 area units . . . 157
6.6.1.3. Maximum length equal 30 length units . . . 158
6.6.1.4. Maximum length equal 15 length units . . . 159
6.6.2. Comparison in quality metrics . . . 159

6.6.2.1. Initial Mesh: Quadtree with Mid Point strategy - Maximum area equal to 198 area units . . . 160
6.6.2.2. Initial Mesh: KD-tree - Maximum area equal to 198 area units 161
6.6.2.3. Initial Mesh: Quadtree with Mid Point strategy - Maximum length equal to 15 length units . . . 162
6.6.2.4. Initial Mesh: KD-tree - Maximum length equal to 15 length units . . . 163

7. Conclusions **164**
7.1. Application . . . 164
7.2. Results of Experiments . . . 165
7.2.1. Quadtree results . . . 165
7.2.2. KD-tree results . . . 165
7.2.3. Comparison with Triangle . . . 166

Bibliography **168**

Appendix A. .OFF file format **171**

List of Tables

6.1. Trendline function for clipping algorithm . . . 122
6.2. Trendline function for initial time mesh generation. . . . 123
6.3. Main study characteristics of the initial meshes. . . . 125
6.4. Main study characteristics of the initial meshes of the unicorn geometry. . . . 128
6.5. Main study characteristics of the initial meshes of the unicorn geometry using Centroid refinement. . . . 130
6.6. Main study characteristics of the initial meshes of the unicorn geometry using Centroid Replication refinement. . . . 131
6.7. Main study characteristics of the initial meshes of the unicorn geometry using Splitting Longest Edge refinement. . . . 133
6.8. Main study characteristics after successive Centroid refinements to bad polygons (Mean area). . . . 135
6.9. Main study characteristics after successive Centroid Replicate refinements to bad polygons (Mean area). . . . 136
6.10. Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (Mean area). . . . 137
6.11. Main study characteristics after successive Quadtree Refinement to bad polygons (Mean area). . . . 138
6.12. Main study characteristics after successive Centroid refinements to bad polygons (One tenth of the mean area). . . . 140
6.13. Main study characteristics after successive Centroid Replicate refinements to bad polygons (One tenth of the mean area). . . . 140
6.14. Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (One tenth of the mean area). . . . 142
6.15. Main study characteristics after successive Quadtree Refining to bad polygons (One tenth of the mean area). . . . 142
6.16. Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (Average edge length). . . . 144
6.17. Main study characteristics after Quadtree Refining to bad polygons (Average edge length). . . . 145
6.18. Main study characteristics after successive of Splitting Longest Edge refinements to bad polygons (One half of the average edge length). . . . 147
6.19. Main study characteristics after successive Quadtree Refining to bad polygons (One half of the average edge length). . . . 148
6.20. Comparing main study characteristics with Triangle (Max: 1979 area units). . . . 154
6.21. Comparison of refinement by region between our application using a (kdtree and splitting longest edge) and triangle. (Criterion: maximum area 1979 units). . . . 156

6.22. Comparing main study characteristics with Triangle (Max: 198 area units). . . 157
6.23. Comparing main study characteristics with Triangle (Max: 30 length units). . 158
6.24. Comparing main study characteristics with Triangle (Max: 15 length units). . 159

List of Figures

2.1. Example of polygon mesh . . . 6
2.2. Manifold and no-manifold Meshes. . . . 7
2.3. Vertex-Vertex representation example . . . 8
2.4. Face-Vertex representation example . . . 9
2.5. Halfedge example . . . 10
2.6. Projection P from the center C of the circle to a segment $\overline{AB}$ . . . 16
2.7. Vector display in the projection . . . 17
2.8. Calculating the angle between two vectors . . . 20
2.9. Quadtree bunny mesh example. . . . 23
2.10. KD-Tree example. . . . 24
3.1. Voronoi Diagram and dual Delaunay triangulation . . . 26
3.2. Example of degenerate intersection in quadtree. . . . 33
3.3. Possible cases of intersections without overlap between the segments. . . . 34
3.4. Possible cases of intersections with overlaping between the segments. . . . 34
4.1. Flow diagram of the process of drawing a polygon . . . 38
4.2. Flow diagram of the process of uploading a contour OFF file . . . 39
4.3. Flow chart of the process of uploading a mesh OFF file . . . 40
4.4. Flow chart of the process of quad-refining an initial mesh . . . 41
4.5. Flow chart of the process of quality refining . . . 43
4.6. Flow chart of the process of quality inspection . . . 44
4.7. Flow chart of the process of exporting meshes . . . 45
4.8. UML Diagram for Geometry and Selector Region . . . 46
4.9. UML Diagram of Polygon in detail. . . . 47
4.10. UML Diagram for clipping algorithms. . . . 48
4.11. UML Diagram for criteria. . . . 49
4.12. UML Diagram for refining algorithms. . . . 50
4.13. UML Diagram for division algorithms . . . 51
4.14. UML Diagram for Mesh . . . 52
4.15. UML Diagram for HalfEdges . . . 52
4.16. UML Diagram for Storages . . . 53
4.17. UML Diagram for Tree . . . 54
4.18. UML Diagram for React Components. . . . 55
5.1. General view of the application. . . . 71
5.2. Geometry creation and initial meshes panel. . . . 73
5.3. Application: Drawing a polygon with the given points. . . . 75
5.4. Quadtree Refining Panel. . . . 76
5.5. Application: Refining a polygon mesh with a quadtree. . . . 77

5.6. Application: Quadtree refinement for a polygon mesh, and subsequent user refinement. . . . 78
5.7. Refining Panel. . . . 79
5.8. Application: Refining a polygon mesh with quality algorithms. . . . 80
5.9. Quality Component. . . . 81
5.10. Example of cutting polygons. . . . 82
5.11. Sutherland Hodgman's algorithm example. . . . 82
5.12. Border cases of Greiner Hormann Algorithm. . . . 86
5.13. Calculation of the intersections between two polygons. . . . 87
5.14. Polygon lists before and after inserting intersection. . . . 88
5.15. Polygon chain orientation. . . . 89
5.16. Two polygons example. . . . 95
5.17. Classifying intersections and building lists. . . . 96
5.18. Example of constructing the cut polygons from both lists 99
5.19. Example of Half Division Algorithm. . . . 102
5.20. Example of Arbitrary Point Insertion Algorithm. . . . 104
5.21. Border cases in Arbitrary Point Insertion Algorithm. . . . 106
5.22. Getting problematic points after clipping. . . . 110
5.23. Fixing polygons after clipping. . . . 112
5.24. Refinement of a polygon using quadtrees. . . . 113
5.25. Example of splitting longest edge algorithm. . . . 115
5.26. Splitting longest edge problem. . . . 116
5.27. Example of refining using Centroid algorithm. . . . 118
5.28. Example of refinement using Centroid with replication algorithm. . . . 119
6.1. Time analysis for clipping algorithm. . . . 122
6.2. Time analysis for initial meshes. . . . 123
6.3. Geometry for initial mesh creation. . . . 124
6.4. Four different strategies to obtain an initial mesh. . . . 125
6.5. Unicorn geometry for quality refinements. . . . 126
6.6. Four different initial meshes of the unicorn geometry. . . . 127
6.7. Applying centroid refinement to the different initial meshes. . . . 129
6.8. Applying centroid replication refinement to the different initial meshes. . . . 131
6.9. Applying splitting longest edge refinement to the different initial meshes. . . . 133
6.10. Unicorn geometry with bad polygons (Mean area). . . . 134
6.11. Succesive Centroid refinement to bad polygons (Mean area). . . . 135
6.12. Succesive Centroid Replicate refinement to bad polygons (Mean area). . . . 136
6.13. Succesive Splitting Longest Edge refinement to bad polygons (Mean area). . . . 137
6.14. Succesive Quadtree Refinement to bad polygons (Mean area). . . . 138
6.15. Unicorn geometry with bad polygons (One tenth of the mean area). . . . 139
6.16. Succesive Centroid refinement to bad polygons (One tenth of the mean area). . . 139
6.17. Succesive Centroid Replicate refinement to bad polygons (One tenth of the mean area). . . . 140
6.18. Succesive Splitting Longest Edge refinement to bad polygons (One tenth of the mean area). . . . 141
6.19. Succesive Quadtree Refining to bad polygons (One tenth of the mean area). . . . 142
6.20. Unicorn geometry with bad polygons (Average edge length). . . . 143

6.21. Succesive Splitting Longest Edge refinement to bad polygons (Average edge length). 144
6.22. Succesive Quadtree Refining to bad polygons (Average edge length). 145
6.23. Unicorn geometry with bad polygons (One half of average edge length). 146
6.24. Succesive Splitting Longest Edge refinement to bad polygons (One half of the average edge length). 146
6.25. Succesive Quadtree Refining to bad polygons (One half of the Average edge length). 147
6.26. Metrics for Average Area. 149
6.27. Metrics for one tenth of the Average Area. 150
6.28. Metrics for Average Length. 151
6.29. Metrics for one half of Average Length. 152
6.30. Unicorn geometry refined by Triangle (Max: 1979 area units). 153
6.31. Unicorn region to be refined. 155
6.32. Unicorn region refined (Max: 1979 area units). 156
6.33. Unicorn geometry refined by Triangle (Max: 198 area units). 157
6.34. Unicorn geometry refined by Triangle (Max: 30 length units). 158
6.35. Unicorn geometry refined by Triangle (Max: 15 length units). 159
6.36. Initial Mesh: Quadtree - Quality metric comparison with Triangle (Max: 198 area units). 160
6.37. Initial Mesh: KD-tree - Quality metric comparison with Triangle (Max: 198 area units). 161
6.38. Initial Mesh: Quadtree - Quality metric comparison with Triangle (Max: 15 length units). 162
6.39. Initial Mesh: KD-tree - Quality metric comparison with Triangle (Max: 15 length units). 163

Chapter 1

Introduction

The increase of applications that use polygon meshes in real life, such as the proliferation of video games, the modeling of objects in mechanical engineering and simulations of the earth in geology and space in astronomy, makes the exploration of new algorithms to generate polygon meshes a relevant research topic.

In addition, the ease of access to graphic processing with the new video cards, and the popularity of mathematical methods for the convergence of solutions to model simulations, such as PFEM and VEM, make the implementation of a tool that generates meshes of polygons is a big challenge, since the comparison in terms of quality and number of elements with respect to other mesh generators is relevant. In particular, in this work we compare our results with Triangle, which is a program that generates geometric meshes composed only of triangles.

Therefore the motivation of this thesis work is the creation of a graphics and interactive web application that groups together an important group of topics, such as: 1) its open source nature, 2) flexible and sustainable code based on a class design and inheritance 3) exploration end evaluation of novel algorithms to generate an initial polygon mesh and 4) refine mesh elements in real time. When working with polygons with an arbitrary number of sides, this application allows greater flexibility when generating a polygon mesh, therefore we expect that its performance will be better in terms of requiring less elements, compared to others meshes that are only made up of triangles and quadrilaterals.

The research questions that we want to answer in this thesis are the following:

1. Does the quadtree structure and its generalizations allow the generation of proper polygon meshes?
2. What is the best point insertion strategy in a generalized quadtree to get a proper mesh of arbitrary polygons?
3. Is it possible to use a kd-tree structure to generate of proper polygon meshes?
4. Which strategy uses the least computational time to build an initial polygon mesh?
5. Which refinement strategy produces the best results according to the standard metrics?
6. Does an arbitrary polygon mesh allow modeling complex geometry using fewer elements than triangles and quadrilaterals?

7. Does a mesh of arbitrary polygons satisfy the quality criteria, even when fewer elements are expected in their conformation compared to the meshes of triangles and quadrilaterals?
8. Is it possible to create a flexible polygon meshing library that allows the testing, inclusion of new algorithms/metrics in an easy way and and the comparison between them?

Our specific hypotheses for this thesis are the following:

1. It should be possible to develop a novel meshing algorithm based on kd-trees to generate proper polygon meshes
2. The polygon mesh generator based on kd-trees should be able to produce meshes with fewer elements: points, edges, and polygons, compared to quadtrees.
3. The quality of the elements should be similar or better quality than those produced by triangle mesh generators, such as Triangle.
4. It should be possible to design a flexible, extensible and open source polygon meshing library and application

1.1. Objectives

The objectives set for the development of this thesis work are presented below. Both the general objective and the specific objectives are included, which define the goals and progress of this research.

1.1.1. Main objectives

The overall goal is to develop graphic and interactive web application that allows a user to explore and evaluate new strategies for the generation of polygon meshes based on generalized quadtrees and kd-trees. The polygons can be convex and non-convex, but not self-intersecting. The purpose of the work is to compare the different algorithms to generate polygon meshes and provide a framework that supports the integration, testing and comparison new strategies. The meshes created inside this application will also be compared with meshes composed of triangles, which satisfy the same quality metrics required by polygon numerical methods.

1.1.2. Specific objectives

The specific objectives show in depth the development of this thesis, defining those goals that must be met to achieve the general objective. In no specific order, the objectives are as follows:

- Develop a polygon mesh generator based on generalized quadtrees and kd-trees
- Develop a polygon mesh viewer that allows you to see the meshes created by the generator
- Develop a way for the user to interact with the generator and to refine polygon meshes

- Create different refinement algorithms to improve the quality of polygons
- Define and implement an appropriate set of quality metrics to compare different mesh types
- Compare the different types of polygon mesh creation algorithms with each other, in terms of number of elements and time taken
- Compare the different types of quality refinements with each other, in terms of quantity of elements and time taken
- Compare the algorithms developed in this work with Triangle, in terms of quantity of elements, time taken and quality of the generated polygons

1.2. Methodology

To address the design and implementation of different strategies for generating an initial mesh, evaluating different refinement algorithms and metrics, we divide the development into the following steps. Each step describes the approach:

1. Development and incorporation of refinement algorithms using quadtrees given an initial geometry. The algorithms focus on cutting polygons based on one another, depending on the regions that are generated by a quadtree or kd-tree when they partition the space on inserting points.
2. Based on the above, we investigate efficient algorithms and data structures for clipping polygons, and flexible representations of polygon meshes, due to the changing nature of the generator. These data structures allow adding and refining polygons according to different criteria.
3. Creation of a web application, based on React, that is capable of visualizing the initial polygon, and how the mesh is formed as the user decides to refine by zones or based on a certain criterion. The web application is built on the basis of drawing libraries that facilitate the use of WebGL for better use of the GPU in renderings.
4. Incorporation of different quality metrics for a polygon mesh. This in order to comparatively determine if one mesh is better or worse than another, regardless of the shape of the polygons that make it up.
5. Comparison of the results obtained based on a triangulation generator (Triangle), according to the quality metrics incorporated in this thesis. The design of the experiments is based on comparing similar procedures, such as the generation of a mesh based on a certain geometry, and the application of refinement based on a criterion provided by the user.

1.3. Thesis content

This thesis is organized as follows:

- Chapter 2 shows the most relevant prior knowledge you must have to understand this work, covering geometric concepts such as polygons and geometric meshes, quadtree

and kd-tree data structures, as well as basic algorithms of intersections, areas, and element memberships in others.

- Chapter 3 reviews the related work about mesh generation using quadtrees, polygon cutting algorithms, and existing polygon mesh generators.
- Chapter 4 deals with the high-level design of the solution presented for the development of the application and the experiments carried out in this work.
- Chapter 5 shows the implementation and the specific algorithms used for the elaboration of the generator.
- Chapter 6 shows the results obtained for the different experiments carried out.
- Chapter 7 shows the conclusions of our work and the future work that remains to be done to improve it.

Chapter 2

Background

This chapter explains some key concepts and knowledge, with the purpose of understanding this thesis. For this, we will detail the geometric concepts, data structures and algorithms used in this thesis.

2.1. Closed Polygons

Throughout this thesis, we continually use the concept of polygon, and therefore necessary to have a concrete definition of what a closed polygon is.

Definition 1 (Closed polygon)**.** *A closed polygon P is described by an ordered set of vertices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}$. The connectivity of the segments of the polygon is given by two consecutive vertices, so these are: $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_nv_{n-1}}, \overline{v_{n-1}v_0}$. Note that the fact that the polygon will be closed is implicit in the definition, since the last vertex is assumed to be connected to the first one of the set.*

2.2. Polygon Meshes

This work continually refers to the concept of polygonal mesh, hence this section will deepen some basic concepts of these, such as their definition, description, to finally describe some examples of typical representations.

2.2.1. Basic Definitions

Definition 2 (Polygon Mesh)**.** *A Polygon Mesh or polymesh is a collection of vertices, edges and faces that defines a shape in a 2D plane or 3D space. Formally, a Polygon Mesh is a triple (V, E, F) where V is a set of* `Vertices` *or points in a plane or space. $E \subset (V \times V)$, it is a set of* `Edges` *or line segments, and finally $F \subset E$, is a set of* `Polygons`*, also called* `Faces`*.*

Figure 2.1: Example of a polygon mesh of Tokyo.[1]

According to the shape of the Polygons, we can classify the two dimensional Meshes as `Triangle Meshes` if the mesh is conformed just by triangles, `Quadrilateral Meshes` if the mesh is conformed by polygons of four sides. For that mesh to be well conformed, we need to introduce a two new classifications for the edges.

Definition 3 (Manifold Edge)**.** *An edge of a mesh is manifold, if it is part of* ***exactly*** *two faces.*

Definition 4 (Boundary Edge)**.** *An edge of a mesh is boundary, if is part of* ***exactly*** *one face.*

2.2.1.1. Manifold Meshes

With these definitions, we can say define what is a well conformed mesh for most computer graphics applications.

Definition 5 (Manifold Mesh)**.** *A mesh is manifold if every edge in the mesh is either a boundary edge or a manifold edge. Moreover, we say a mesh is manifold closed, if the mesh is manifold and it is non-intersecting.*

[1] Image obtained from https://es.clipdealer.com/vector/media/A:112461842

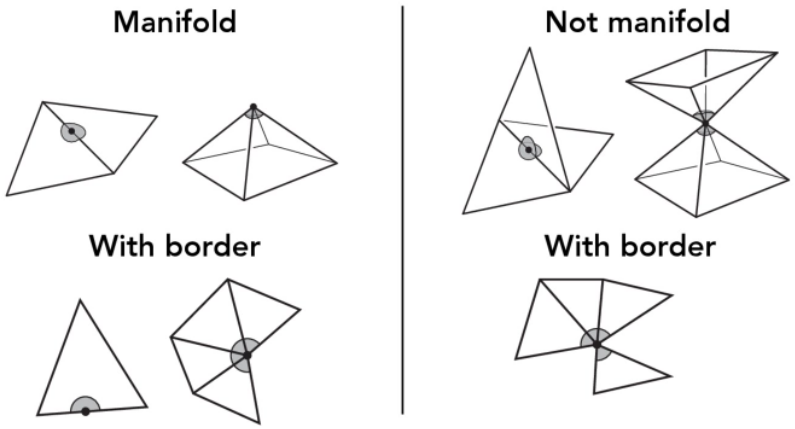


Figure 2.2: Examples of manifold and non-manifold meshes. [2]

If a mesh is a manifold then three very useful properties are met:

1. A border always connects exactly two faces, unless it is a boundary border.
2. An edge always connects two vertices.
3. A polygon or face consists of a closed set of vertices joined by edges.

2.2.1.2. Orientation of a Mesh

A key concept in meshes is the `orientation` of the faces. If we know the orientation of a polygon, we will know which side is inside and which side is outside of the region. The orientation can be `Counter Clockwise (CCW)` or `Clockwise (CW)`. Knowing this, we can define what an oriented manifold mesh is.

Definition 6 (Manifold Oriented Mesh)**.** *A manifold mesh is orientable, if every face in the mesh has consistent orderings, i.e., all faces are CCW or CW.*

2.2.2. Mesh Representations

In this subsection we face the problem of how a polygon mesh can be represented. In the literature there is a great variety of data structures that help us achieve this purpose, so in this work we only explain the most relevant ones.

2.2.2.1. Vertex-Vertex

The mesh is represented by a list of Vertices $v_0, v_1, v_2, ..., v_{n-1}$, with their respective coordinates. Each one of the vertices is connected to another list of Vertices, that represents the ones that it is connected with. For instance, let us say that v_i has coordinates (v_ix, v_iy), and it's connected to the list $[\hat{v}_0, \hat{v}_1...\hat{v}_{n-1}]$. That represents the fact that the point v_i is connected, or in other words, has as neighbors in the mesh, the points $\hat{v}_0, \hat{v}_1...\hat{v}_{n-1}$.

This leads to a clear disadvantage, because all the faces and edges are implicit, so every time we need to check for a polygon in the mesh, we must traverse the list of vertices and

[2] Image obtained from https://cs184.eecs.berkeley.edu/sp18/lecture/mesh-rep/slide_018

jump from one list to another to reconstruct the faces. In conclusion, the operations over edges and faces are not easily accomplished.

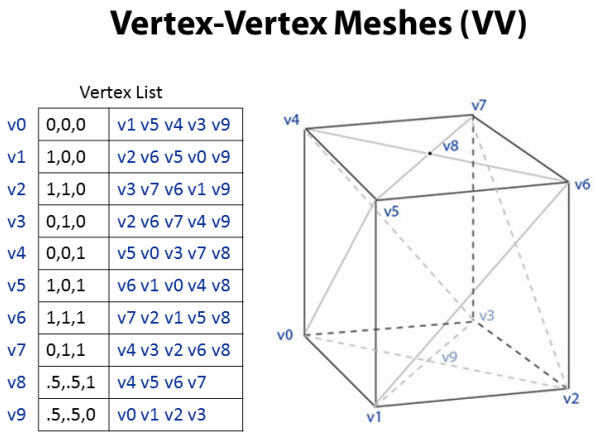


Figure 2.3: Example of the representation of a cube using a list of Vertices. [3]

2.2.2.2. Face-Vertex

The mesh in this case is represented by a list of Vertices $v_0, v_1, v_2, ..., v_{n-1}$, each one with a reference to their respective coordinates. A big difference is that now the vertices are stored only once, and we now explicitly have a list of Faces (in contrast with Vertex-Vertex). This list of Faces $f_0, f_1, f_2, ..., f_{n-1}$ represents each one of the Polygons in the mesh, and every f_i points to a list of Vertex references, so the space used by every Face is equal to the degree d of it, having d references.

This representation is important because it is used directly in the development of our work. The advantages it has are the simplicity of implementation, and the ease with which one can handle the information of points and polygons in a static mesh. However, due to the dynamic nature of our mesh generator, it was necessary to make changes for new insertions and deletions of points and polygons, therefore, we extended its operations to maintain the correctness of the information, such as polygons. have no references to points that no longer exist within the mesh.

The detailed implementation of how the Face-Vertex representation was used in our work can be found in the implementation section 5.1.

[3] Image obtained from https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Vertex-Vertex_Meshes_(VV).png

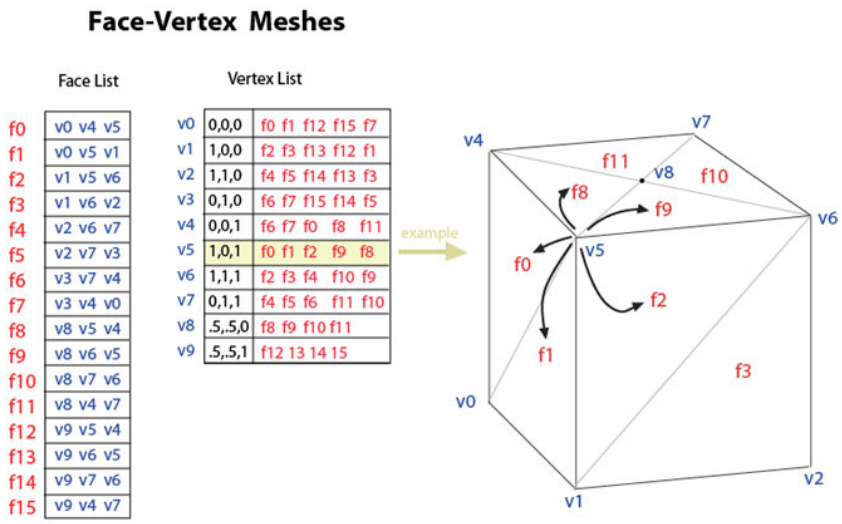


Figure 2.4: Example of the representation of a cube using two lists: one for vertices and other for faces. [4]

2.2.2.3. Halfedge

The halfedge data structure centers the mesh connectivity information on the edges of the polygons. Each edge is divided into two halfedges with opposite directions, each associated with a different polygon. In addition, each halfedge has information about the vertex that originates it, the edge to which it is attached, and a reference to the next halfedge within a polygon. Note that because each halfedge is steerable, therefore, the representable polygon meshes must also be steerable, either clockwise or counterclockwise.

In our work, this representation was used, with the following connectivity information:

1. Each vertex has a reference to a halfedge that has it as its origin.
2. Each polygon has a reference to any halfedge of one of its edges, in the correct direction (in our case all polygons are counterclockwise oriented).
3. Each edge is referenced to a halfedge in a certain direction, depending on the order in which the edge was visited at the time of connectivity construction.
4. Each halfedge has the following references:
   - 4.1. The vertex that is the origin of the halfedge.
   - 4.2. The polygon to which the halfedge belongs. In case of boundary edges of the polygon, the face of the opposite halfedge does not exist.
   - 4.3. The edge to which the halfedge is associated.
   - 4.4. The next halfedge inside the polygon, arranged counterclockwise.

[4] Image obtained from https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Mesh_fv.jpg

4.5. A reference to the opposite halfedge associated with the edge.

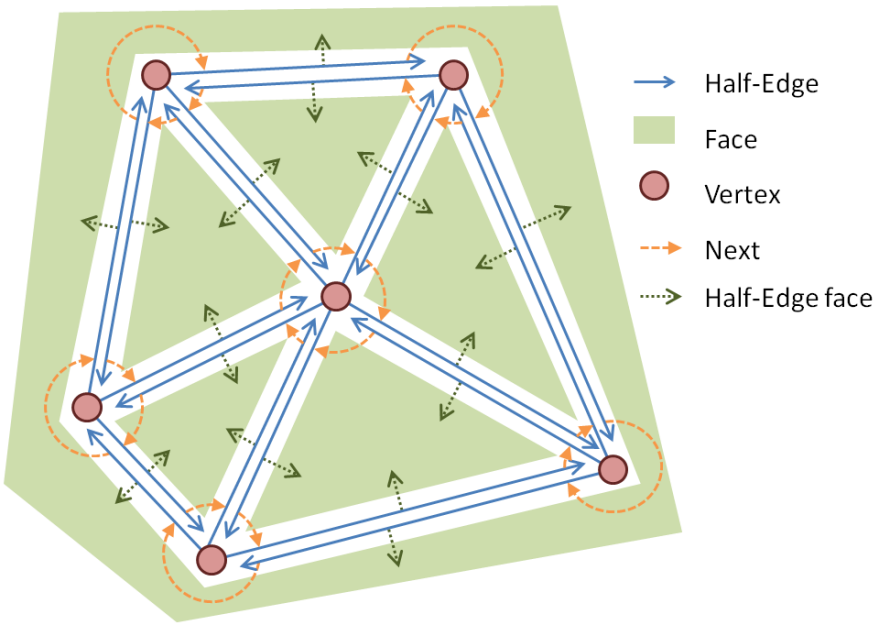


Figure 2.5: Example of representation of a mesh using halfedges. [5]

2.3. Geometry Algorithms

2.3.1. Math Background

2.3.1.1. Cross Product

Cross Product or Vector product of two vectors $\vec{A}$ and $\vec{B}$, denoted $\vec{A} \times \vec{B}$, it's a binary operation that gives another vector perpendicular to both $\vec{A}$ and $\vec{B}$. Cross product is defined over $\mathbb{R}^3$, and one of the property we're going to use in this investigation is its **anticommutativity**, i.e, $\vec{A} \times \vec{B} = -\vec{B} \times \vec{A}$. In other words, if we operate $\vec{A} \times \vec{B}$ and we examine its sign, we can determine if the resulting vector is in the positive or negative side of the plane.

In a formal way, and remembering that this thesis works only in two dimensions, we write any 2D vector as $\vec{A} = (a_x, a_y, 0)$ and $\vec{B} = (b_x, b_y, 0)$. We calculate the cross product as it follows.

[5] Image obtained from https://www.leonardofischer.com/wp-content/uploads/2011/11/dcel.png

$$\begin{aligned}\vec{A} \times \vec{B} &= (a_x, a_y, 0) \times (b_x, b_y, 0) \\ &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & 0 \\ b_x & b_y & 0 \end{vmatrix} \\ &= a_x b_y - a_y b_x\end{aligned}$$

Now, given three points $P = (p_x, p_y)$, $Q = (q_x, q_y)$, $R = (r_x, r_y)$. We are interested in knowing the cross product of $\overrightarrow{PQ} \times \overrightarrow{PR}$, and examine the sign of the result. With that information, we later explain how we can say in which side of the vector $\overrightarrow{PQ}$ is the point R. Notice that in this case, our vector $\vec{A}$ is $(q_x - p_x, q_y - p_y)$ and our vector $\vec{B}$ is $(r_x - p_x, r_y - p_y)$. Replacing this in the formula given before, we obtain:

$$\overrightarrow{PQ} \times \overrightarrow{PR} = (q_x - p_x) \cdot (r_y - p_y) - (q_y - p_y) \cdot (r_x - p_x)$$

With that formula, we can elaborate an algorithm to calculate the cross product of three points P, Q, R, representing in that order the cross product $\overrightarrow{PQ} \times \overrightarrow{PR}$.

Algorithm 1 Cross Product of three Points P, Q and R

```
1: Input
2:     P        Point with the properties p_x and p_y
3:     Q        Point with the properties q_x and q_y
4:     R        Point with the properties r_x and r_y
5: Output
6:              Numeric value of the cross product between PQ and PR

7: function CrossProduct(P, Q, R)
8:     return (q_x − p_x) · (r_y − p_y) − (q_y − p_y) · (r_x − p_x)
9: end
```

2.3.2. Points Related Algorithms

2.3.2.1. Point inside Polygon

The Ray casting algorithm is the one used to determine if a point is inside or outside of a polygon. This algorithm consists in extending a ray from the point you want to examine, in a certain direction to infinity. Subsequently, the number of times the ray intersects the edge of the polygon is examined. If the point is outside the polygon, then the ray must intersect an even number of times, otherwise, if the point was inside, it must intersect an odd number of times.

This algorithm has complications when the point is just on the edge of the polygon, or if the ray intersects a corner point of the polygon. In addition, the precision problem must be taken into account, so in our work we always work using an epsilon for all operations.

Algorithm 2 Point inside Polygon

Input
  P Point to be checked if it's inside $Poly$
  $Poly$ Polygon with a list of vertices $v_0, v_1, ..., v_n$
Output
  True if P is inside $Poly$. False otherwise.

function INSIDE(P, $Poly$)
  $inside \leftarrow False$
  $points \leftarrow Poly.points$
  for i $\leftarrow$ 0 **to** n-1 **do**
    $j \leftarrow (i + 1) \mod points.length$
    $p_i \leftarrow points[i]$
    $p_j \leftarrow points[j]$
    if P *is above* p_i *and below* p_j *or* P *is above* p_j *and below* p_i **then**
      if P *intersects the segment* (p_i, p_j) **then**
        $inside \leftarrow \neg inside$
  end
  return inside
end

2.3.2.2. Point inside Edge

To establish whether a point P lies within an edge E of end points E_{init} and E_{end}, it is necessary to examine the coordinates of the point $P = (P_x, P_y)$ and verify that:

1. P_x is in the range defined by the minimum and maximum of the X coordinates of E_{init} and E_{end}.
2. Similarly for P_y, it must be in the range defined by the minimum and maximim of the Y coordinates of E_{init} and E_{end}.

With this we verify that the point is in a quadrilateral region based on the E_{init} and E_{end} points, so it remains to be verified if the point is to collinear with the edge. For that, we use the cross product to discard all those cases in which its result is not close to zero in an epsilon factor.

Algorithm 3 Point inside Edge

Input
  P Point to be checked if it's inside *Edge*
  Edge Edge with two Points, *init* and *end*.
Output
  True if P is inside *Edge*. False otherwise.

function INSIDE(P, *Edge*)
  $cross \leftarrow$ CROSSPRODUCT*(Edge.init, Edge.end, P)*
  if $|cross| > \epsilon$ **then**
    return False
  $xInside \leftarrow min(Edge.init_x, Edge.end_x) \leq P_x \leq max(Edge.init_x, Edge.end_x)$
  $yInside \leftarrow min(Edge.init_y, Edge.end_y) \leq P_y \leq max(Edge.init_y, Edge.end_y)$
  return xInside and yInside
end

2.3.2.3. Point inside Circle

Verifying if a point is inside a circle is not a complex task, since it is enough to consider that by definition of a circle, all the interior points are at a distance less than or equal (if we consider the boundary) to the radius of the circle. Therefore, just take the distance between the point you want to consult, and the center of the circle, and compare it with the radius: If the distance is less than or equal to the radius, then the point is inside the circle, otherwise it is outside it.

Algorithm 4 Point inside Circle

Input
  P Point to be checked if it's inside *Circle*
  Circle Circle with a given *radius* and *center*
Output
  True if P is inside *Circle*. False otherwise.

function INSIDE(P, *Circle*)
  $distance \leftarrow \sqrt{(P_x - Circle.center_x)^2 + (P_y - Circle.center_y)^2}$
  return $distance \leq Circle.radius$
end

2.3.3. Edge Related Algorithms

2.3.3.1. Edge intersection

An edge is a segment that connects two points P_{start} and P_{end}. In this case, the Edge has a direction, pointing to the ending point P_{end}. Hence, in order to know the intersection between two edges, the procedure will be calculate for each one, the line equation that contains P_{start} and P_{end}, and with these two linear equations, solve for a solution (x, y).

It is known that given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, the line that contains the two points is described by the following equation.

$$y - y_1 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1)$$

If we then manipulate that equation multiplying and associating, we can get a formula of the form $Ax + By = C$.

$$\begin{aligned} y - y_1 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1) \\ (y - y_1)(x_2 - x_1) &= (y_2 - y_1)(x - x_1) \\ y(x_2 - x_1) - y_1(x_2 - x_1) &= x(y_2 - y_1) - x_1(y_2 - y_1) \\ \underbrace{(y_1 - y_2)}_{\text{A}}\, x + \underbrace{(x_2 - x_1)}_{\text{B}}\, y &= \underbrace{y_1(x_2 - x_1) - x_1(y_2 - y_1)}_{\text{C}} \end{aligned} \tag{2.1}$$

That is convenient because we do not deal with undefined slopes for vertical lines, and division by zero. Therefore, given two linear equations for each Edge we solve the system with the Cramer's rule.

$$\begin{aligned} \ell_1 &= A_1 x + B_1 y = C_1 && \text{Line equation for first Edge} \\ \ell_2 &= A_2 x + B_2 y = C_2 && \text{Line equation for second Edge} \end{aligned}$$

We can then calculate the determinant of the system $S = \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}$. If S it's zero, then the system may have infinite solutions, or not solutions at all, so we will have to check additionally if the first edge is collinear with the second edge, checking if the cross product is zero.

Otherwise, we will calculate the solution (x, y), having in consideration that the point may lie outside of the line segments, because we were considering them as infinite lines. The intersection is calculated as it follows.

$$x = \frac{1}{S}\begin{vmatrix} C_1 & B_1 \\ C_2 & B_2 \end{vmatrix}, \quad y = \frac{1}{S}\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}$$

Finally, the algorithm that follows the method explained before, to get the intersection of two edges, is described now.

Algorithm 5 Intersection between two Edges

Input
  $Edge_1$ $Edge$ with start Point A and end Point B
  $Edge_2$ $Edge$ with start Point C and end Point D
Output
  Intersection Point (x, y) if it exists, *null* otherwise.

function INTERSECTION($Edge_1$, $Edge_2$)

Calculating line $\overline{AB} \coloneqq a_1x + b_1y = c_1$
  $a_1 \leftarrow B_y - A_x$
  $b_1 \leftarrow A_x - B_x$
  $c_1 \leftarrow a_1(A_x) + b_1(A_y)$

Calculating system determinant S *and solutions*
  $S \leftarrow a_1b_2 - a_2b_1$
  collinear $\leftarrow |\text{CROSSPRODUCT}(A, B, C)| < \epsilon$ *and* $|\text{CROSSPRODUCT}(A, B, D)| < \epsilon$
  if $|S| < \epsilon$ *and collinear* **then**
    return *null*
  else
    $x \leftarrow \frac{1}{S}(b_2c_1 - b_1c_2)$
    $y \leftarrow \frac{1}{S}(a_1c_2 - a_2c_1)$
    $P \leftarrow$ *new Point*(x, y)
    if INSIDE*(P, Edge*$_1$*)* *and* INSIDE*(P, Edge*$_2$*)* **then**
      return P
    else
      return *null*
end

2.3.3.2. Edge intersection with Quadrilateral

The problem of determining if an Edge intersects a Quadrilateral becomes easy with the function that returns the intersection of two edges. We just have to check if the given Edge intersects at least one of the four Edges of the Quadrilateral. If that occurs then we say they intersect, if not, then there is no intersection.

Algorithm 6 Intersection between Edge and Quadrilateral

Input
    Edge *Edge* with Points *init* and *end*
    Quad Quadrilateral with four *Edges*
Output
    True if *Edge* intersects *Quad*. False otherwise.

function INTERSECTS(*Edge*, *Quad*)
    $(leftEdge, rightEdge, topEdge, bottomEdge) \leftarrow Quad.getEdges()$
    $left \leftarrow$ INTERSECTION*(Edge, leftEdge)*
    $top \leftarrow$ INTERSECTION*(Edge, topEdge)*
    $right \leftarrow$ INTERSECTION*(Edge, rightEdge)*
    $bottom \leftarrow$ INTERSECTION*(Edge, bottomEdge)*
    return *(left exists) or (top exists) or (right exists) or (bottom exists)*
end

2.3.3.3. Edge intersection with Circle

To determine if a circle is intercepted by any $\overline{AB}$ segment, we decided to use the vector projection strategy that joins the segment's starting point with the center of the circle, on segment $\overline{AB}$. The problem with this strategy is that the point P corresponding to the intersection of the projection with the edge $\overline{AB}$ does not always fall on it, as can be seen in the figure 2.6.

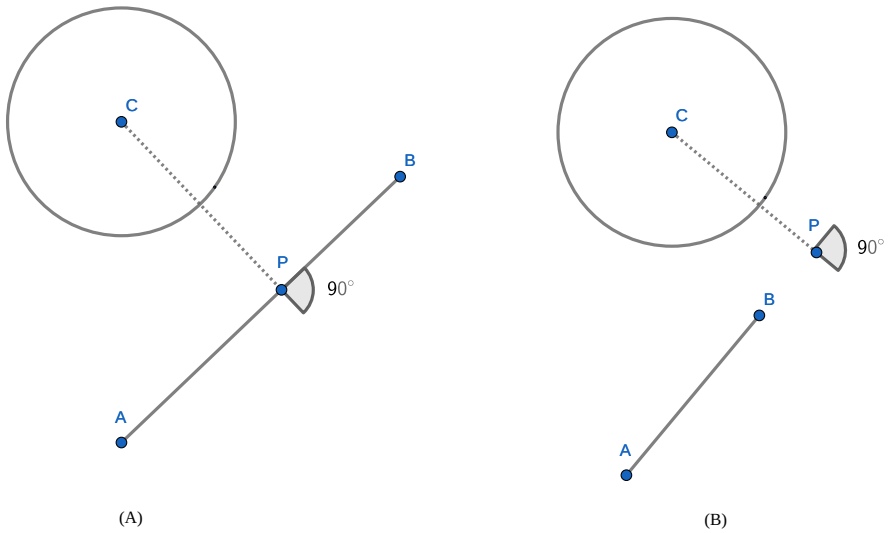


Figure 2.6: Projection P from the center C of the circle to a segment $\overline{AB}$. (a) The projection falls on the segment. (b) The projection falls outside the segment.

To solve that, we add border cases to face such problems. The first is that if there is any endpoint of the segment that is within the circle, then necessarily the segment is intercepted. The second is that if the point P of the projection exists, and it is also inside the segment,

then the circle is intercepted, as long as the distance between the center and P is less than or equal to the radius.

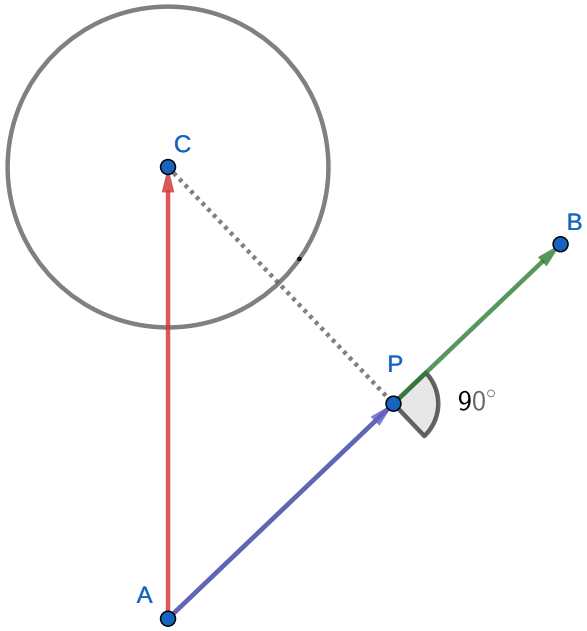


Figure 2.7: Vector display in the projection

Using mathematics and geometry, and considering the vectors shown in the figure 2.7, we know that the projection of $\vec{AC}$ on $\vec{AB}$ can be calculated as:

$$proj_{\vec{AB}}\vec{AC} = \frac{\vec{AC} \cdot \vec{AB}}{\left\|\vec{AB}\right\|^2}$$

Therefore the point P can be calculated using this scalar, weighting the vector $\vec{AB}$ and carrying out the translation in the plane with the point A:

$$P.x = A.x + proj_{\vec{AB}}\vec{AC} * \vec{AB}.x$$
$$P.y = A.y + proj_{\vec{AB}}\vec{AC} * \vec{AB}.y$$

Next, we show the algorithm that uses this procedure in a function called `projection`, which receives a circle and a segment, and returns the point P, which is the intersection of the projection of $\vec{AC}$ on $\vec{AB}$. The Intersects function checks the edge cases mentioned above and returns true if the segment intersects the circle, or false otherwise.

Algorithm 7 Intersection between Edge and Circle

Input
  Edge *Edge* with start Point A and end Point B
  Circle Circle with a center Point C and radius r
Output
  True if *Edge* intersects *Circle*. False otherwise.

function INTERSECTS(*Edge*, *Circle*)
  insideStart ← INSIDE*(Edge.A, Circle)*
  insideEnd ← INSIDE*(Edge.B, Circle)*
  if *insideStart or insideEnd is True* **then**
    return True
  P ← PROJECTION*(Edge, Circle)*
  if *not* INSIDE*(P, Edge)* **then**
    return False
  if *Distance between Circle.C and P is less or equal than Circle.r* **then**
    return True
  else
    return False
end

2.3.4. Polygon Related Algorithms

2.3.4.1. Polygon intersection with Circle

Checking if a polygon intersects a Circle is made much easier since we have the `Intersects` function that checks if an Edge intersects a Circle. Just go through each of the edges of a polygon and invoke the function: if its result is True, then the polygon intersects the circle, otherwise there is no intersection. However, there is a border case in which the circle is completely contained within the polygon or vice versa, since in both cases the above is not true, but there is an intersection. To verify that, after processing each of the edges, we ask if the center of the circle is inside the polygon, returning True because there is an intersection, otherwise we return False.

Algorithm 8 Intersection between Polygon and Circle

```
1: Input
2:     Polygon    Polygon to be checked
3:     Circle     Circle with a center Point C and radius r
4: Output
5:                True if Polygon intersects Circle. False otherwise.

6: function INTERSECTS(Polygon, Circle)
7:     Edge ← Get edges from Polygon
8:     foreach Edge in Edges do
9:         if INTERSECTS(Edge, Circle) then
10:            return True
11:    end
12:    return INSIDE(Circle.C, Polygon)
13: end
```

2.3.4.2. Area of a Polygon

Since our polygons do not have holes nor do they self-intersect, we can calculate the area of a polygon as the sum of the cross products having each of the vertices of the polygon as a pivot. This result will give us **twice** the area of the polygon, since for each vertex we calculate the area of the parallelogram and not the corresponding triangle.

There are two considerations. The first is that the Nth point is equivalent to the point with index 0. The second is that depending on the order in which the vertices are visited, it will be the sign of the area, being counterclockwise a positive area, and clockwise negative area. This is why we must take the absolute value to know the area. The formula for this is as follows [7]:

$$\mathbf{A} = \frac{1}{2}\left|\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|$$

2.3.4.3. Centroid of a Polygon

As for the centroid calculation, there is a formula, written by Paul Bourke in [7]. In the formula, A represents the area, and as before, the Nth point corresponds to the point of index 0.

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$
$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

2.3.4.4. Getting minimum angle of a Polygon

To calculate the angle of a vertex of a polygon, what we must do is obtain the vectors corresponding to the two segments that share the vertex in the polygon, and apply the definition shown in the figure 2.8.

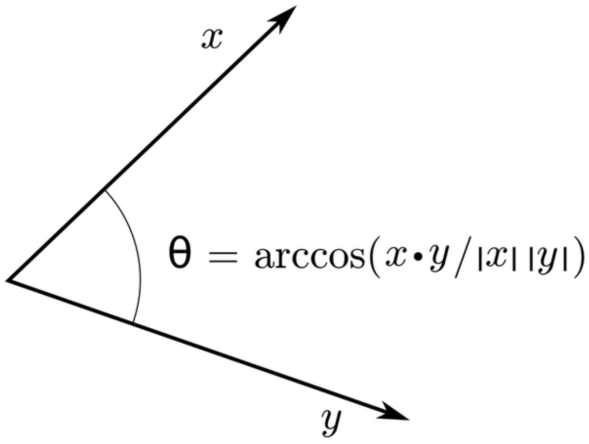


Figure 2.8: Calculating the angle between two vectors

Therefore, to obtain the minimum angle it is enough to go through each one of the vertices of a polygon and calculate each one of the angles, keeping the minimum in each iteration. Since the number of vertices is finite and bounded to an integer, then the operation is considered to be $\mathcal{O}(1)$.

2.4. Data Structures

2.4.1. Quadtrees

A *quadtree* is a recursive data structure, that partitions a region of the two dimensional plane, into four new regions bounded by two straight lines parallel to each of the Cartesian axes. There is a quadrant called *root* that covers the entire domain, which is recursively divided into four *children* quadrants, each time getting smaller. Then, the complete collection of recursively generated quadrants forms a tree.

The refinement done by a *quadtree*, is applied every time we insert a point in the plane, allowing the existence of different partition algorithms of each region, depending on how we split them. The refinement of a region done by a regular *quadtree* when a point is inserted, can be described as it follows.

Algorithm 9 Quadtree Point Inserting

```
1: Input
2:     Q          quadtree
3:     p          point to be inserted in the tree
4: Output
5:                True if the point was inserted. False otherwise.

6: function INSERT(Q, p)
7:     if current quadrant does not contain p then
8:         return False
9:     if p is already inserted on Q then
10:        return False
11:    if Q is a leaf then
12:        Q_points ← current points stored in Q
13:        if capacity of Q is enough to handle p then
14:            add p to Q_points
15:            return True
16:        else
17:            (Q_nw, Q_ne, Q_sw, Q_se) = divide Q in four quadrants.
18:            Q.northWest ← Q_nw
19:            Q.northEast ← Q_ne
20:            Q.southWest ← Q_sw
21:            Q.southEast ← Q_se
22:            reinsert every point of Q_points into the quadtree
23:            return SUBDIVIDEDINSERTION(Q, p)
24:    else
25:        return SUBDIVIDEDINSERTION(Q, p)
26: end
```

Note that the Algorithm 9 uses a subroutine called `SubdividedInsertion`, that tries to insert the point in each of the four Quadrants of the current *quadtree*. The details are given below.

Algorithm 10 Subroutine of point insertion in Quadrants

```
1: Input
2:     Q          quadtree
3:     p          point to be inserted in the tree
4: Output
5:                True if the point was inserted. False otherwise.

6: function SubdividedInsertion(Q, p)
7:     if Insert(Q.northWest, p) then
8:         return True
9:     else if Insert(Q.northEast, p) then
10:        return True
11:    else if Insert(Q.southWest, p) then
12:        return True
13:    else if Insert(Q.southEast, p) then
14:        return True
15:    else
16:        return False
17: end
```

If we model a Polygon as a set of Points, ordered in clockwise or counter clockwise, and we insert every Point inside a Quadtree, then we will generate a mesh, adapting to the geometry in every insertion. As the Quadtree is refining the plane, each Quadrant will be *cutting* the polygon until every Vertex is inserted. The result is a initial mesh of the polygon.

It is important to note that according to the Bern-Eppstein-Gilbert theorem, a balanced *quadtree* generates a well-formed mesh for any 2D domain.

Note that the process of dividing a *quadtree* into four Quadrant is not mandatory, because sometimes we will want to divide horizontally or vertically depending of the local polygon. Moreover, we would want to divide the *quadtree* not using the center of the Quadrant, as the original Algorithm does, so we will extend this data structure to be able to handle this operations.

2.4.2. Generalized Quadtrees

Definition 7 (Generalized Quadtree)**.** *A quadtree is called generalized, if the division of the quadrants is variable. It is not divided always by half, but by any point inside the region, or even be divided in two quadrants instead of four.*

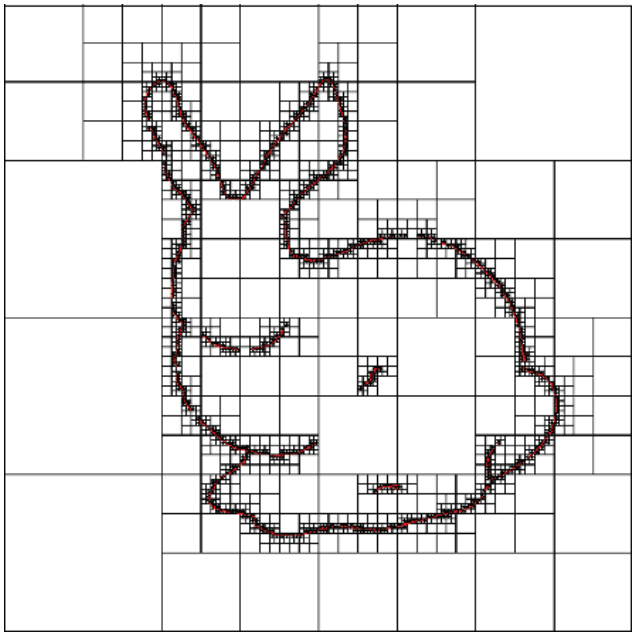


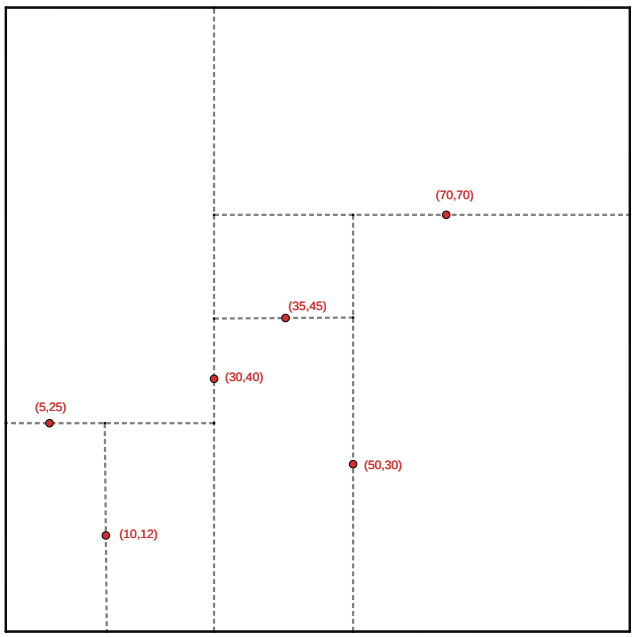
Figure 2.9: Quadtree bunny mesh example.

The generalized quadtree will help us to obtain initial meshes of different types, in order to investigate which one produces the best results, and if these compare and resemble those obtained with triangulations and quadrilateral meshes. The implementation section details the algorithms for forming polygon meshes, as well as the different types of partition of the quadrants.

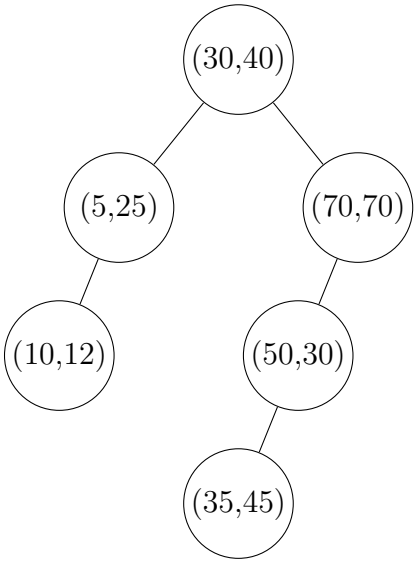
2.4.3. KD-Trees

A KD-tree is called a k-dimensional tree, because it is a tree-like data structure that partitions space in the specified dimension. In our case, the interest dimension is 2, so at each level, dimensional cuts are obtained according to a certain axis (X or Y in the two-dimensional case). A property of these trees is that all those points that are to the left of the hyperplane cut by a dimension, are represented by the generated left subtree, and those that are to the right of the hyperplane, are in the right subtree.

The cutting direction is alternated node by node between dimension X and dimension Y, starting with X from the root of the tree. The cut of the space according to a dimension, for example X, corresponds to a perpendicular hyperplane (in this case vertical), which divides into two planes: all the points with the smallest X coordinate in the left tree, and with the largest X coordinate in the right tree.



(a) Division of the space after inserting the points



(b) KD-Tree formed with the insertion of points.

Figure 2.10: KD-Tree example with the following points inserted in order: (30,40), (5,25), (10,12), (70,70), (50,30) and (35,45).

In the example in the figure, the nodes where the cut dimension is in X are (30,40), (10,12) and (50,30). On the other hand, the nodes in which the cut dimension is in Y are (5,25), (70,70) and (35,45).

The insertion of points can be done in the order in which they are entered, however, the tree can be unbalanced. In order to solve this, a preprocessing of the points is used, selecting the median as a pivot, inserting it into the tree and continuing to insert with the points to the left of the median in the left subtree, and the points to the right in the right subtree.

Chapter 3

Related Work

3.1. Polygon Meshing Algorithms

The main algorithms used to generate polygon meshes can be categorized in two strategies: (i) The use of a partition of space for creating a mesh, for example, the partition generated by a quadtree (ii) The creation of Voronoi cells from a previously generated Delaunay triangulation with the required point density, and (iii) The generation of a Centroid Voronoi Tesellation (CVT). The centroid of each Voronoi cell is inserted and those points are the generators of the polygon mesh. [11] [17].

3.1.1. Geometric Meshes based on Quadtrees

The first use of quadtrees in the context of finite elements is descriibed by Yerry and Shephard [44] [27]. In recent decades, it has been a trend to develop the FDM and FVM based on structured or unstructured irregular meshes with quadrilaterals, triangles and polygons, which have the grid flexibility of the FEM and the merits of the classic FDM and FVM [43]. In that context, meshes generated by quadtrees gained popularity in the last years.[27] [30]

The usual procedure for generating geometric meshes using a quadtree is to create a bounding box that contains all the points of the geometry. Subsequently, the plane is subdivided into four quadrants recursively until a term condition is met, such as, for example, verifying if the number of points that remain on a leaf of the tree are less than a certain threshold. Another approach is to refine the quadtrees depending on its neighbours, for example in [40] the condition of partitioning is the maximum difference between the level of refinement of the adjacent elements.

After obtaining the partition, each of the leaves of the tree is traversed and each generated polygon is triangulated. Finally, all the triangles generated in the leaves are joined and the triangulation is formed. However, there is a problem with those points that are on the edges or in the corners of the quadrants of the quadtree [29], which is solved by modifying the coordinates of the point (and therefore losing precision).

Other strategies involves the use of quadrilateral finite elements. As described in [30] one alternative is subdivide each triangle into three quadrilaterals, and then a node is added

at the centroid of the triangle and at the middle of each edge. But we decided to use the strategy described in [41], where every element generated by a quadtree is considered as a Polygon element, therefore we do not triangulate and for that reason, we expect a lower number of elements. To generate the elements we followed the grid-based method [30], where the resulting grid of the quadtree is superimpose over the initial geometry, and each Polygon element is created using a clipping algorithm.

3.1.2. Geometric Meshes based on Voronoy Cells

Given a set of points belonging to a certain Ω domain, the Voronoi Region corresponding to a given point P_i consists of all the points in Ω that are closer to P_i than to any other in the set. The set of all regions forms a partition of Ω which is called the Voronoi Tessellation or Voronoi Diagram of Ω. The initial set of points is called the generating points or generators. The dual graph corresponds to a Delaunay Triangulation, which is formed by connecting pairs of generator points, corresponding to adjacent Voronoi regions [5] [19].

The generation of geometric meshes using Voronoi algorithms are widely used because there are currently several tools and algorithms that make them suitable for VEM meshes [12]. However, the triangulations may contain triangles with small angles, or triangles with greatly varying area so that most of the algorithms related to the Voronoi–Delaunay triangulations do not provide a guarantee about the quality of the resulting mesh. [15]

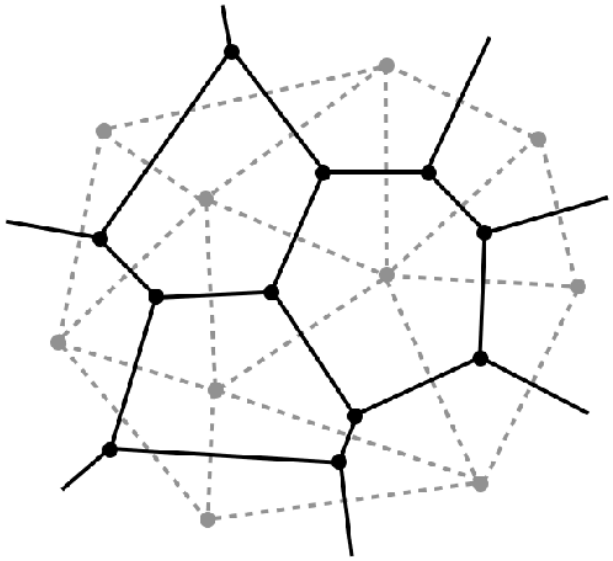


Figure 3.1: Voronoi Diagram and dual Delaunay triangulation. [1]

[1] Image obtained from https://www.researchgate.net/figure/Voronoi-Dual-of-a-Delaunay-Triangular-Mesh_fig37_262772501

3.1.3. Geometric Meshes based on Centroid Voronoi Tessellation (CVT)

One of the alternatives to Voronoi diagrams to produce higher quality geometric meshes is the concept of centroidal Voronoi tessellation based Delaunay triangulation (CVDT). Given a set of points and a positive density function ρ, a Voronoi tessellation is a centroidal voronoi tessellation (CVT), if the generators of each Voronoi region are also the centers of masses of those regions. The dual graph Delaunay triangulation is called Centroidal Voronoi-Delaunay triangulation, which provides a higher quality meshes than a regular Delaunay triangulation [18].

The CVDT provides, in some sense, an optimal distribution of generating points. The construction of CVDT generalizes many existing local smoothing techniques. In fact, a centroidal Voronoi tessellation (CVT) is constructed based on an associated density function and cost (or error, or distortion measure, or energy) functional [18] [15] [14].

3.2. Mesh Visualizers

There are several options for displaying and generating geometric meshes that are open-source, however, no references have been found in the literature to applications that use quadtrees or kd-trees to generate polygon meshes. Examples of viewers are: (1) TetView [37], which is a small graphic program that allows to view tetrahedral meshes, which was created specifically to work with TetGen [36]. (2) MeshLab [13] is a 3D geometry viewer, which allows you to store different mesh formats and interact with them in real time. (3) Camaron [8][9] is an application that allows to view polygon and polyhedron meshes, allowing triangle, quadrilateral and mixed element meshes, but it lacks real-time editing. (4) Triangle [35] is a C program for two-dimensional mesh generation and construction of Delaunay triangulations, constrained Delaunay triangulations, and Voronoi diagrams. Triangle does not have a way to refine in a certain sector, therefore to achieve a certain refinement density in an area the entire mesh must be refined.

In our work we take the ideas of MeshLab of an interactive and user-friendly management to visualize and edit the polygon meshes generated through quadtrees and kd-trees. For this we create a web application that allows the manipulation of geometric meshes through mouse clicks on a canvas. The user can select a refinement region, click on individual polygons for refinement, choose the refinement algorithm, choose the construction strategy of the grid using quadtrees or kd-trees and view the quality characteristics of the mesh through a histogram. Because the application can be accessed through a web browser, the user must not go through the compilation process and the use of flags to perform operations on a polygon mesh, however this implies that the application will not be optimized because it is not implemented using a low-level language like C or C++.

3.3. Polygon Mesh quality metrics

In the literature, there are not many works that address which quality metrics of arbitrary polygons must have to be considered of good quality before the new mathematical methods applied on polygon meshes, such as PFEM and VEM. In [38], different metrics associated with triangles are discussed, and in [2], they discuss the non-consensus of what are the appropriate metrics are to evaluate the quality of a polygon mesh in terms of its performance against a solver, and propose a series of metrics, which we adopt in our work.

3.3.1. Scale Dependent measures

In our work we have used two measurements that are classic to evaluate a polygon: Area, and Minimum length of an edge. These measurements depend on the scale of a polygon, and therefore for the same mesh, we can have different results depending on the size of the elements.

3.3.2. Scale Invariant measures

Because two different meshes (such as our and Triangle mesh) cannot be compared with the previous metrics, we decided to adopt the invariant quality metrics at scale defined in their work [2].

- The first is `Circle Ratio (CR)`, which consists of the division between the radius of the maximum circle inscribed in the polygon and the radius of the minimum circle that encompasses all the points of the polygon (not necessarily passing through all of them). Its range is between $[0, 1]$ and the higher its value, the better the metric.
- The second is `Perimeter Area Ratio (PAR)`, also called compactness of a polygon $\mathcal{P}$, defined by $2\pi * area(\mathcal{P})/perimeter(\mathcal{P})^2$. Its range is between $(0, \infty)$ and the lower its value, the better the metric.
- The third is `Edge Ratio (ER)`, defined as the ratio between the smallest and longest edge of the polygon. Its range is between $(0, 1]$ and the higher its value, the better the metric.
- Fourth is `Minimum Angle` of all interior angles of the polygon.
- Finally, the last metric used is `Normalized Point Distance (NPD)`, which consists of the division between the minimum distance between not necessarily consecutive points, and the diameter of the minimum circumference that encompasses all the points of the polygon. Its range is between $(0, 1]$ and the higher its value, the better the metric.

As mentioned before, these metrics are important since they do not depend on the scale and size of the elements, so they serve to compare the quality of a mesh generated by Triangle, with one generated by our application.

3.4. Polygon Clipping Algorithms

In this section we describe in a general way the Greiner Hormann polygon cutting algorithm and expose its disadvantages in relation to degenerate intersections (those in which one polygon has an edge segment on the edge of another). Subsequently, we show a recent extension to this algorithm that allows to solve the problem of degenerate intersections. The implementation of the Greiner Hormann algorithm and its extension will be discussed in detail in the Section 5.3.1.2.

3.4.1. Greiner Hormann Algorithm

In their work [23], Greiner and Hormann details an algorithm for clipping polygons in 1998. As they define, given two polygons: a `clipper` polygon, and the one to be cut, called `subject`, the `clipped polygon` consists of all point interior to the *clipper* polygon that lie inside the *subject* polygon. The *clipped polygon* can be just one polygon, or a set of polygons. In other words, clipping a polygon against other, means determining the intersection of two polygons.

The data structure used in their algorithm is the following:

```
vertex = {
    x, y       : coordinates;
    next, prev : vertexPtr;
    nextPoly   : vertexPtr;
    intersect  : boolean;
    entry_exit : boolean;
    neighbour  : vertexPtr;
    alpha      : float ;
}
```

They used a doubly linked list of nodes, where each node has a vertex inside with the previously shown information. The basic information is the coordinates of the vertex, as float numbers for x and y, and a reference to the previous and next vertex in the polygon. The *nextPoly* information is to refer the possible new polygons generated after the clipping process, pointing to the first vertex of the next Polygon generated.

The *intersect* field is a flag that is used when determining the intersection points between *subject* and *clipper* polygons. All these intersection points have to be inserted into the *subject* and *clipper* points, in the proper place to keep the list ordered. When doing that, the intersection inserted inside the *subject* keeps a reference to the same intersection inserted into the *clipper* polygon, by the *neighbor* field of the data structure. The *alpha* field is used to sort, indicating where the intersection point lies relative to the start and end point of the edge. Finally *entry_exit* is a flag that records whether the intersection point is an entry or exit point to the other polygon's interior.

Using that data structure, we can define the algorithm in three phases:

3.4.1.1. Phase 1: Searching intersections

In this phase, we search for all intersection points between the *subject* and *clipper* polygons. For doing that, we test every edge of the subject polygon to the ones of the clipper polygon, checking if they intersect or not. If they intersect, then we get *alpha* values between 0 and 1, indicating where the intersection point lies relative to the start and end of both edges. Using the alpha values, they insert the intersections in the proper place inside the data structures of the subject and clipper polygons.

If no intersection is found, we have then three possible cases: subject point lies completely inside the clipper polygon or vice versa, or that both polygons are disjoint. Using the algorithm point inside a polygon (*even-odd rule*), we can determine which case we have, and return the correct result.

Algorithm 11 Pseudo-code for Phase one

```
1:  foreach vertex S_i in subject polygon do
2:      foreach vertex C_j in clipper polygon do
3:          if S_iS_{i+1} intersects with the edge C_jC_{j+1} then
4:              I_1 ← new Vertex(S_i, S_{i+1})
5:              I_2 ← new Vertex(C_j, C_{j+1})
6:              link intersection points I_1 and I_2 by neighbour
7:              sort I_1 into subject polygon
8:              sort I_2 into clipper polygon
9:      end
10: end
```

3.4.1.2. Phase 2: Marking entry and exit points

For labeling the point as entry and exit, they presented an analogy with a chalk cart. Let us imagine we are pushing a chalk cart along the subject polygon boundary. If we start on a vertex point we manipulate the hatch of the cart setting it as *closed* if we're outside the clipper polygon, or *open* if we are inside. We push the cart along the boundary of the subject polygon, toggling the position of the hatch as open/closed when we cross an edge of the clipper polygon. We stop when we reach the initial vertex. As a result of this, all the segments of the subject polygon that are inside the clipping polygon are marked with chalk.

Using the same technique for the clipper polygon, we discover the parts that are inside the subject polygon. Then, we find that the clipped polygon will be the result of merging the parts discovered before.

Algorithm 12 Pseudo-code for Phase two

```
1: for both polygons P do
2:     if P_0 inside other polygon then
3:         status ← exit
4:     else
5:         status ← entry
6:     foreach vertex P_i in polygon do
7:         if P_i.intersect then
8:             P_i.entry_exit ← status
9:             toggle status
10:    end
11: end
```

3.4.1.3. Phase 3: Constructing the clipped polygon

In order to construct the new clipped polygon, they created two routines: `newPolygon` and `newVertex`. The routine *newPolygon* registers the beginning of a new polygon, and the *newVertex* adds the new vertex to the last polygon created by the routine *newPolygon*. A simple example they give is the following:

```
newPolygon
newVertex(A)
newVertex(B)
newVertex(C)
newPolygon
newVertex(D)
newVertex(E)
newVertex(F)
newVertex(G)
```

The instructions create two polygons: $P_1 = ABC$ and $P_2 = DEFG$. For A, the parameter *nextPoly* is set pointing to D. Taking this into consideration, the pseudo code for the third phase is the following.

Algorithm 13 Pseudo-code for Phase three

```
1: while unprocessed intersecting points in subject polygon do
2:     current ← first unprocessed intersecting point of subject polygon
3:     newPolygon
4:     newVertex(current)
5:     do
6:         if current.entry then
7:             do
8:                 current ← current.next
9:                 newVertex(current)
10:            while current.intersect is False
11:        else
12:            do
13:                current ← current.prev
14:                newVertex(current)
15:            while current.intersect is False
16:        current ← current.neighbour
17:    while Polygon is not closed
18: end
```

Essentially, the third phase consists in moving through the boundaries of the subject and clipping polygon, reconstructing every clipping polygon. Making the analogy with the chalk cart, first we move to one of the intersection points and open the hatch, considering this as the creation of a new polygon. We then move along the subject polygon's edge, in the direction of *entry_exit* label of the start point. If the flag was *entry*, then we move forward in the list, otherwise, we move backward.

When we find a vertex, we call the routine *newVertex* to add it to the polygon in creation. Note that when we find the next intersection point, it means we leave the clip polygon's interior, and change the traversing boundary to the *clipper* polygon. The direction of movement is chosen again with the value of *entry_exit* label.

This process continues until we arrive at the starting vertex. There we close the hatch, and the polygon is closed and finished. The process of traversing *entering* and *exiting* lists continue as long as there are still intersection points that have not been processed.

3.4.1.4. Disadvantages of the algorithm

The algorithm works as long as there are no *degenerate* intersections. For Greiner and Hormann, they call a *degenerate* intersection to every vertex of a polygon that lies in the edge of another polygon. The problem in the paper is solved by perturbing these points, moving them from their original place. This allows the algorithm to continue, but leads to inaccurate solutions to the mathematical models used on the meshes.

This problem is important to our thesis because we deal with degenerated intersections when we clip using the quadrants of a quadtree. If we try to disturb those intersections, that would lead us to errors about the false belonging of points to quadrants, and with it,

an erroneous conformation of the polygon mesh. In addition, it would be known in advance that disturbing the points would alter the original geometry, and therefore, the mathematical results of applying a VEM model on the mesh would not be accurate.

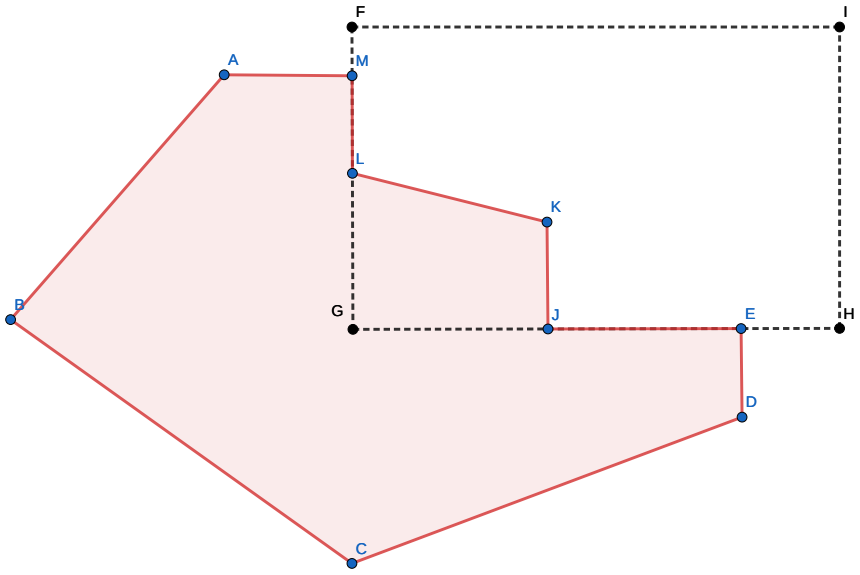


Figure 3.2: Example of degenerate intersection in quadtree

According to the figure 3.2, suppose that we have a polygon $ABCDEJKLM$, which we will call the subject polygon, which will be cut by a quadrant of the quadtree represented by the cut polygon $FGHI$. Note that the intersections between both polygons are points E, J, L and M.

These intersections are degenerate intersections, since they are cases where there is overlap between the edges of the subject polygon and the cutting polygon, a situation that occurs very often when working with quadtrees. The problem with these intersections is that they lead to ambiguities in terms of generating entry and exit lists, and consequently, erroneous results.

3.4.2. Extended Greiner Hormann Algorithm

In their work [21], Foster et al propose an extension of the algorithm, maintaining the simplicity of the stages. What they add is a more exhaustive classification of the intersections, analyzing locally the segments that accompany the intersection in the subject polygon, compared to the cut polygon.

3.4.2.1. Classification of Polygonal Chains

Definition 8 (Degenerate Intersection)**.** *We say that an intersection is degenerate, if it is a point of a polygon P, which is contained in a segment of the polygon Q, or coincides with some point of the polygon Q. The case is analogous for the points of Q, which comply with the above properties in P.*

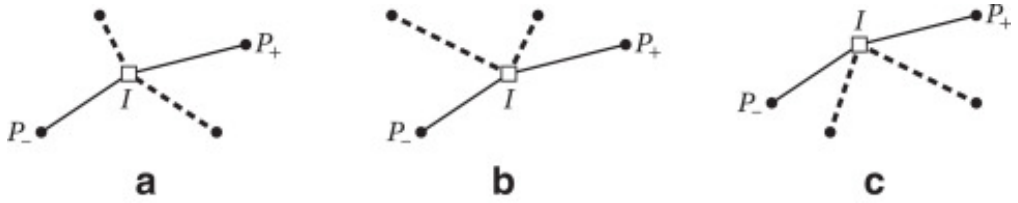


Figure 3.3: Possible cases of intersections without overlap between the segments. (A) Crossing (B) and (C) Bouncing. Image taken from [21]

Let P_- and P_+ be two points of the Subject Polygon, I be an intersection point with Clipper Polygon, as the Figure 3.3 shows. Let Q_- and Q_+ be the points before and after the intersection I in the Clipper Polygon respectively, then we have the following cases: If Q_- and Q_+ are on opposite sides of the polygon chain P_-IP_+, then it is said to be a **crossing intersection**. If Q_- and Q_+ are both on the same side, either to the right or to the left, then we are facing a **bouncing intersection**.

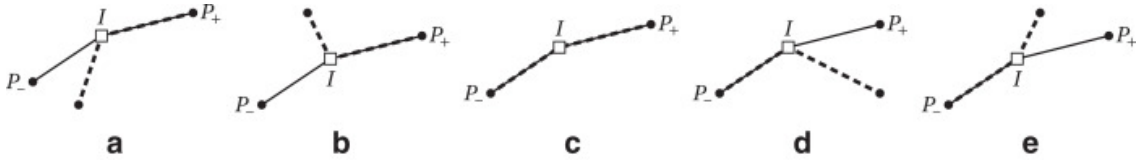


Figure 3.4: Possible cases of intersections with overlaping between the segments. (A) Left-on. (B) Right-on. (C) On-on. (D) On-left. (E) On-right. Image taken from [21]

Regarding the intersections that overlap, as shown in the Figure 3.4, we must go to the clipper polygon in the segment $\overline{Q_-I}$ and $\overline{IQ_+}$, and see how the segments $\overline{P_-I}$ and $\overline{IP_+}$ behave:

- If $\overline{IP_+}$ and $\overline{IQ_+}$ overlap, then then we must study the segments $\overline{P_-I}$ and $\overline{Q_-I}$. If $\overline{P_-I}$ is to the left of $\overline{Q_-I}$, we will say that the intersection is **"left on"**, otherwise it is **"right on"**.
- Similarly, if $\overline{P_-I}$ and $\overline{Q_-I}$ overlap, if $\overline{IP_+}$ is to the left of $\overline{IQ_+}$, it is an **"on left"** intersection, whereas if it is to the right, it is an **"on right"** intersection.
- If there is overlap between both segments, then it is an **"on on"** intersection.

Chapter 4

Design

In this chapter, we describe at a high level how the mesh generator application based on quadtrees and kd-trees was designed. We analyze the different requirements that the application should meet, the possible solutions that we propose, and later, the chosen solution together with the class and flow diagrams.

4.1. Mesh Generation Process

The main steps of any mesh generation process can be summarized as follows:

1. Build or read the geometry object
2. Generation of an initial mesh that fits the geometry domain
3. Generation of an intermediate mesh that satisfies the density requirements specified by the user
4. Generation of an improved mesh that satisfies the quality criteria
5. Generation of the final mesh

For each step the designed application provides the following functionality:

1. The input geometry can be read from a file with extension .off, or drawn by the user. In addition, an already generated mesh can also be uploaded.
2. The initial meshes can be either generated by a quadtree or kd-tree approach. In particular, different ways of inserting points and dividing the space are applied when using a quadtree, such as random insertion of points, and subdivision using the midpoint or an arbitrary point.
3. The initial mesh generated by a quadtree or kd-tree can continue to be refined both at the level of a single polygon, a selected region or the entire mesh. The user is in charge of carrying out this process, selecting what he wants to refine intuitively with the mouse.
4. The intermediate mesh can receive more specific quality refinements to a single polygon, to a specific region, or to the entire mesh. The quality refinements included in the application consist of splitting longest edge, centroid and centroid with replication.

5. Finally, when the user wants to finish the process, he can export his mesh in .off format, in addition to consulting the different statistics associated with the mesh visually, such as the number of polygons, the minimum angle, the minimum side, among others.

4.2. Analysis of possible solutions

Regarding the possible solutions to develop the polygon mesh generator, we found two:

1. Make a program under the C++ language, optimizing memory and reaction time. However, the display of the meshes becomes complex because integrating drawing libraries is not something direct or trivial. In addition, this path implies a fairly high learning curve due to the handling of graphical algorithms at a very low level.

2. Make a web application under the Javascript and Typescript languages, using the React framework and some graphic library to show the generated meshes. The advantage of this path is the language friendliness, and the ease with which a web application can be integrated with new WebGL technologies to use the GPU with high-level instructions. The great disadvantage of this path is that the application will not be optimized in time or memory, so it would remain an application for learning and displaying meshes.

Based on the pros and cons of the two paths, we decided to follow that of the web application, due to the rapid need to obtain a visualization of the generated polygon meshes and thus focus on the logic of the mesh generator, in particular the obtaining smaller polygons with a consistent cutting algorithm. In addition, the ease and experience with the Javascript and Typescript languages, make the learning curve focus only on the graphic field and the corresponding algorithms.

4.3. Proposed software architecture

In this section we show our architecture proposal for the application we want to create. We focus on the first instance on the decisions we make for the creation of the software, and then analyze the design of processes, the modeling of classes and components, and finally the design of experiments.

4.3.1. General Architecture

Because our polygon mesh generator must be created from scratch, it means that there is a greater degree of freedom in terms of the architecture that we must choose for the application. As mentioned before, we have decided to create the mesh generator as a web application using the new technologies that are in vogue, such as the Javascript and Typescript languages with Node.js and React. The architecture on which these applications are based is the use of components, which we can see as classes, with communication and information flow from top to bottom. This means that there is a larger component that renders various small components by passing global information to them via *props*, while each small component has its own information, encapsulating its behaviors.

However, this structure is made only for rendering the views of the application. Regarding the logic and generation of the polygon mesh itself, we follow the guidelines of a framework for a modular component mesh generator using object-oriented programming [4]. This logic is encapsulated in a layer far from the views, and is in charge of defining each element of the application, such as points, edges, polygons, algorithms, among others, and how these interact with each other to generate polygon meshes.

Regarding how to display the results, we decided to use a graphics-focused library that can be easily integrated into web applications, and that is optimized as much as possible to render quickly using WebGL. The use of a library makes integration to a web page much more efficient and simple, focusing on the interface that the library provides for managing drawings on a canvas, instead of going into direct WebGL commands that are beyond reach of this thesis.

4.3.2. General Implementation Choices

As mentioned before, we decided to use Javascript and Typescript, in particular we used React and NodeJs for creating the views of the application. The logic of the application is done with Typescript, using the benefits of classes, types and cleaner code.

In relation to the graphic library, we used PixiJS, also written in Javascript, so its incorporation into the web application is much easier. This library is optimized for two-dimensional rendering work using the WebGL engine to speed up the drawing of polygons on web pages using the GPU through a high-level and easy-to-use interface.

4.4. Process Design

In this section we address the different processes that a user is able to follow in our application. We explain how a user can start the process by drawing a polygon by clicking on a canvas to insert its vertices, or by uploading a file in off [1] format with the initial geometry, or a mesh already created. Additionally, we see the process of how the user can refine the initial meshes generated by the application, and check their quality according to quality criteria. Finally, the user can export their results in off, or Veamy file format.

[1] Files in .off format are explained in more detail in Appendix A.

4.4.1. User draws a contour geometry in our Application

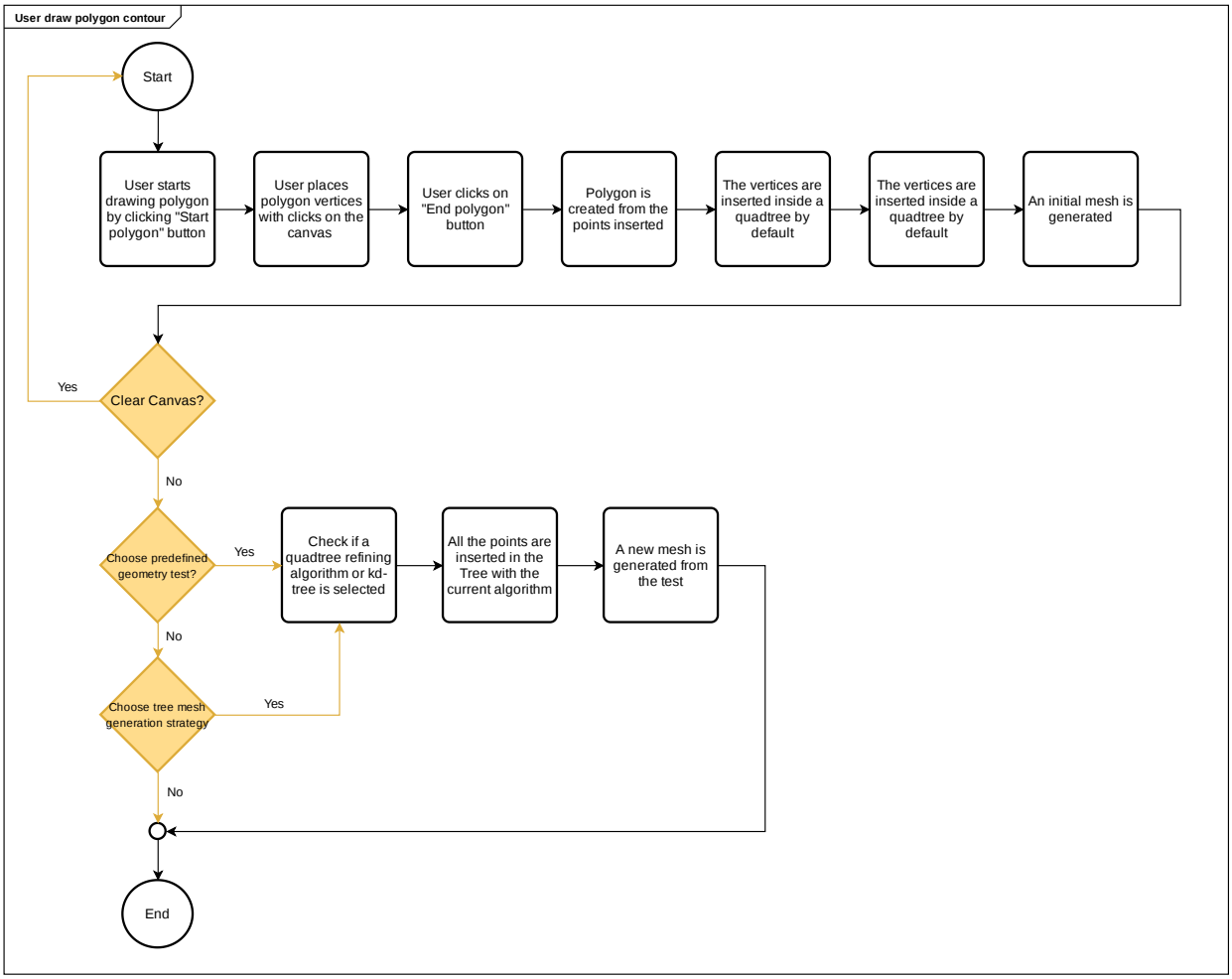


Figure 4.1: Flow diagram of the process of drawing a polygon.

In this process the user does not have an initial geometry, so the application offers the possibility of drawing one on the canvas. To do this, the user must click on the canvas, in each place where there is a vertex of the geometry. When the user presses the "end polygon" button, the points are joined in the same order in which they were created, generating the initial geometry. If the user is not satisfied with his geometry, he/she can click on the "clear canvas" button to go back to the beginning. The user also has the ability to choose test geometries already defined within the application. If you choose one, it overwrites the existing geometry on the canvas.

In any case, the initial points of the geometry (either drawn by the user or a pre-existing one), are inserted in a quadtree that by default implements the division of its quadrants using the midpoint algorithm. In case the user wants to choose another quadtree division algorithm, or use a kd-tree, the points are reinserted and a new initial mesh is generated. This last step concludes the process in which the user creates an initial geometry from the application.

4.4.2. User uploads Contour Geometry

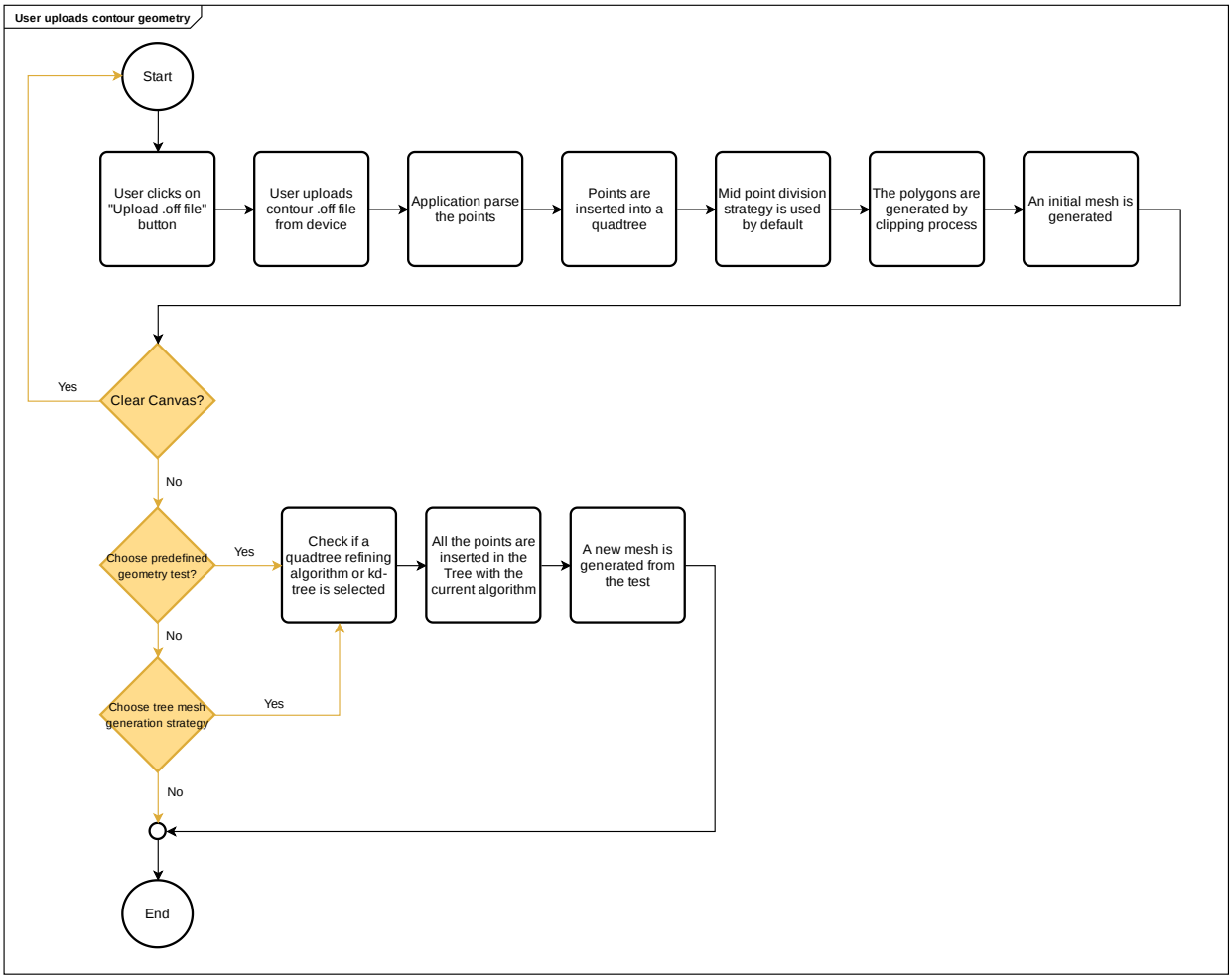


Figure 4.2: Flow diagram of the process of uploading a contour OFF file.

Here the user uploads to the application a file with the extension .off, which can contain only points, or a list of points with a single polygon. The conformation of the .off file is assumed to be correct, that is, the defined points conform exactly to the initial geometry, and there are no excesses. Initially the application parses the file, obtaining all the points and their coordinates, and as in the previous process, inserts them in a quadtree with division algorithm using the midpoint. From now on, the process is identical to the previous one, since the user can select predefined tests, draw a geometry and change the type of tree or its division algorithm.

4.4.3. User uploads Mesh by off File

We now explain the process of how the application acts when a user uploads a mesh represented in a off file. First, the user must upload the file to the application, then internally, it parses the information, getting the points and storing them considering their indexes. After that, the application parses the polygon information, creating each one of them, assuming

that the information of the mesh inside the file is correct.

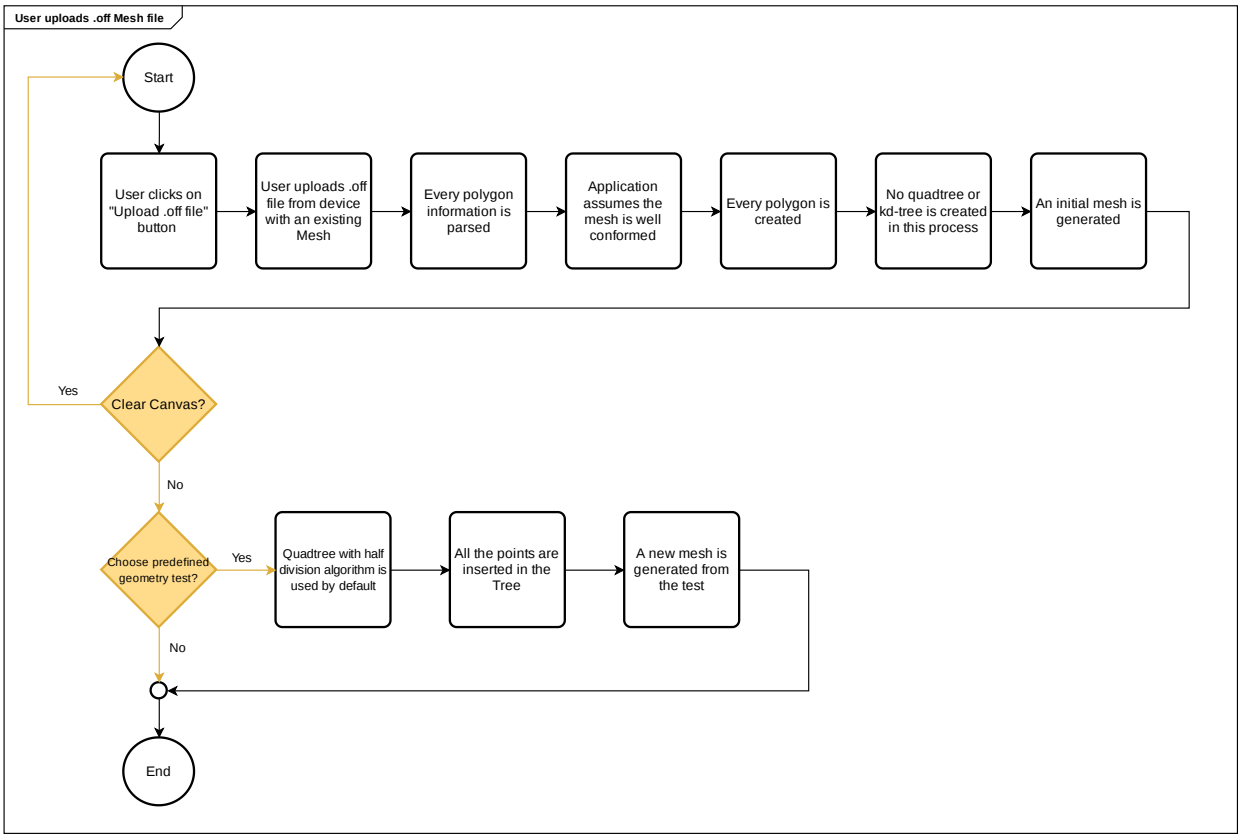


Figure 4.3: Flow chart of the process in which the user uploads a mesh OFF file.

However, there is a big difference with respect to the two previous processes, since here it is not possible to create a quadtree or kd-tree. This is because it is very difficult to obtain a tree whose subdivision is capable of generating the mesh that the user enters. Consequently, each polygon does not have a reference to the quadrant that generated it, so the options to change the tree type and its division algorithm are not available for this process.

4.4.4. Quadtree refining process

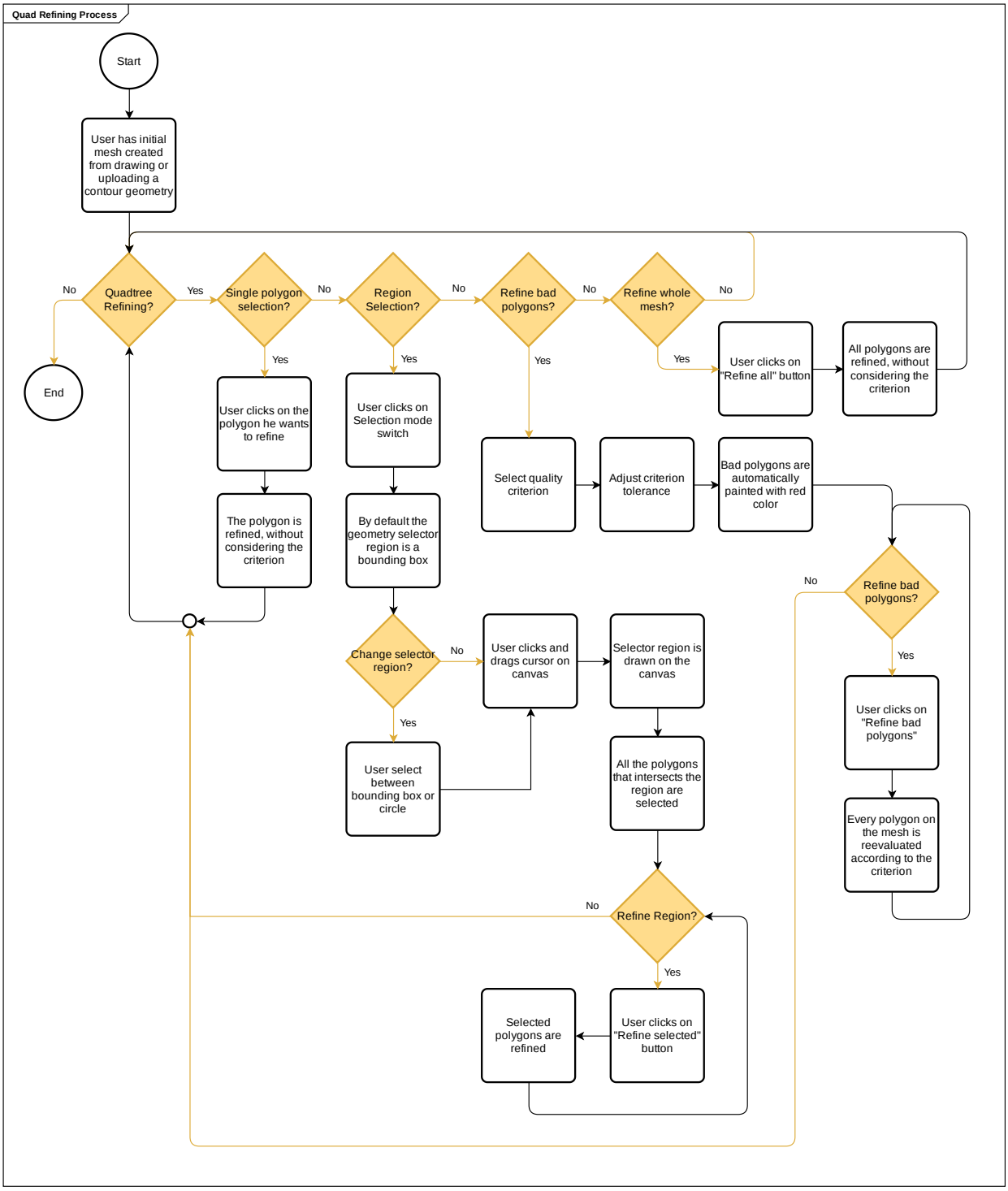


Figure 4.4: Flow chart of the process of quad-refining an initial mesh.

It consists of a subdivision using the midpoint of the quadrant that generated a certain initial polygon, either using a quadtree or kd-tree. We decided to call this process "quadtree refining" because only the midpoint division strategy is available, leaving other possible division implementations as future work. This procedure is available only for those meshes that

were generated by a quadtree or kd-tree, that is, one available for all the cases explained above except when the user uploads an already created mesh. In case the user wishes to carry out this refinement, the user has four alternatives:

1. **Refine a single polygon**: If the user moves the pointer towards the canvas, they can see that the polygons are marked red when it intersects them. If the user clicks, the polygon is marked as "selected" and further refined. This procedure can be done many times.

2. **Refine a delimited region**: If the user wants to refine all the polygons that are in a certain region, he can do so by enabling the "selection mode" switch. If the user clicks on the canvas, and holds it down, he can see how the selection region is increasing or decreasing in size while moving the pointer. There are two selection regions in the app: `Bounding Box` and `Circle`. Still, the integration of new selection geometries is easy due to the class modeling done for this. When the user stops clicking, all those polygons that intersect the selection region will be marked in red, and they can be refined by pressing the "refine selected" button, or you can redraw a new selection region.

3. **Refine polygons marked as bad**: If the user decides to apply a quality criterion, such as marking as "bad polygons" all those that have an area greater than an arbitrary number, then he can choose to refine only those polygons, leaving the others intact. For that, as mentioned, a quality criterion and a tolerance must be chosen, and then click on the "Refine bad polygons" button.

4. **Refine whole mesh**: Finally, if the user decides to refine all the polygons of the mesh, he can do so by clicking on the "Refine all polygons" button. In this option all polygons are marked as if they were selected and refined.

4.4.5. Quality refining process

It starts with the assumption of an existing initial mesh, and consists of applying quality refinement algorithms to the polygons of said mesh. It is important to note that if a refinement of this type is performed, refinement by quadtree is automatically disabled, due to edge conditions and inconsistencies. This refinement has the same four alternatives as the previous process, with the difference that a quality refinement algorithm must be previously chosen. This application implements the following three types of algorithms:

1. **Centroid**: Refining using the Centroid of the polygon.

2. **Centroid with Replication**: Refining using the Centroid and replicating the polygon around it.

3. **Splitting Longest Edge**: Refining splitting the longest edge of the polygon by the mid point, and create two polygons with similar area ratio.

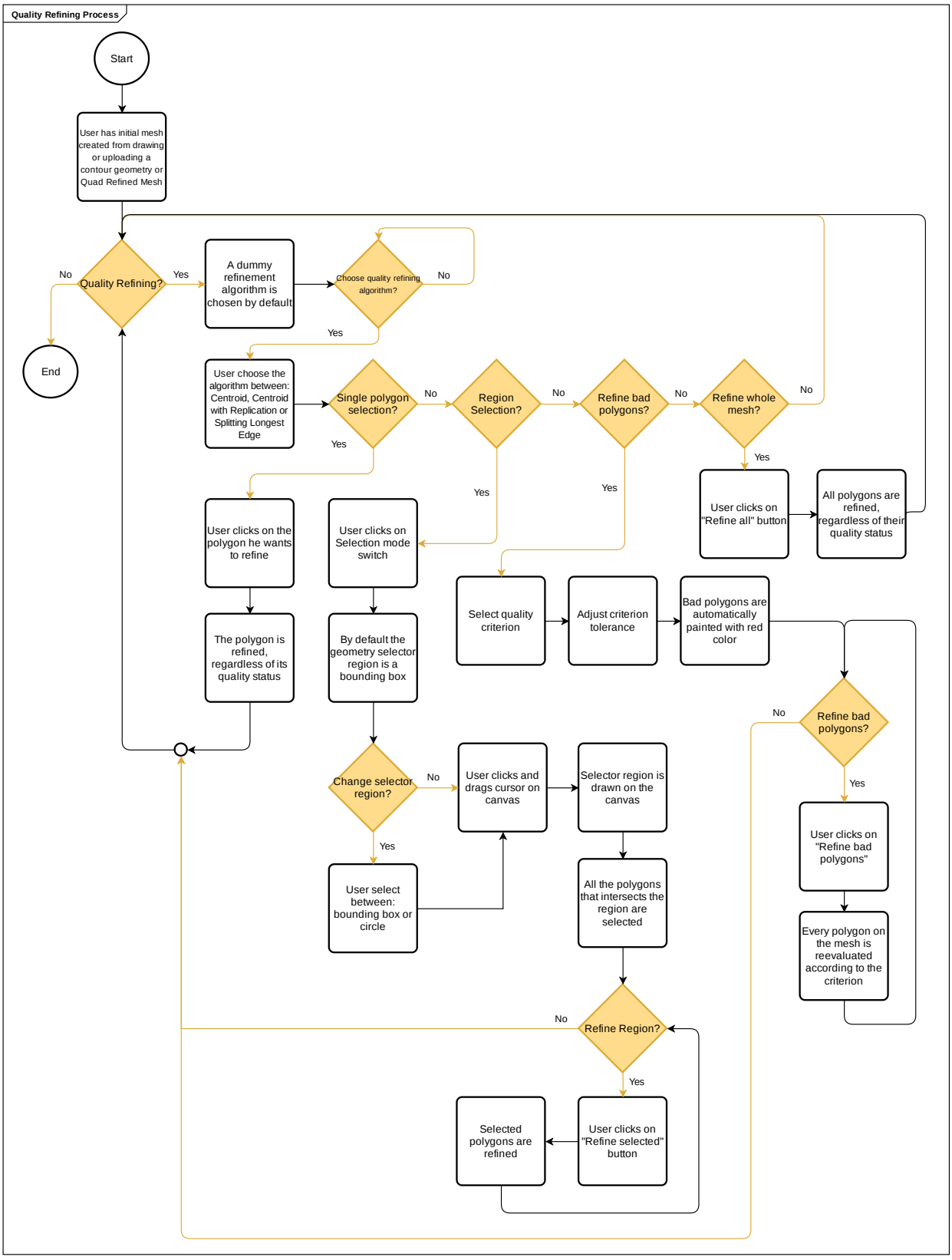


Figure 4.5: Flow chart of the process of quality refining.

4.4.6. Quality inspection

In the middle of any of the processes explained above, the user can check the quality of the mesh according to typical metrics for a mesh: minimum and maximum average area, minimum and maximum average edge length and average maximum and minimum angle of the mesh. The information is shown through a bar histogram, whose horizontal axis corresponds to each polygon, and the vertical axis to the value of the analyzed metric. Based on this information, the user can enter a tolerance value, which will automatically mark as bad polygons, all those that are under or above the said value, depending on whether the metric is maximum or minimum respectively.

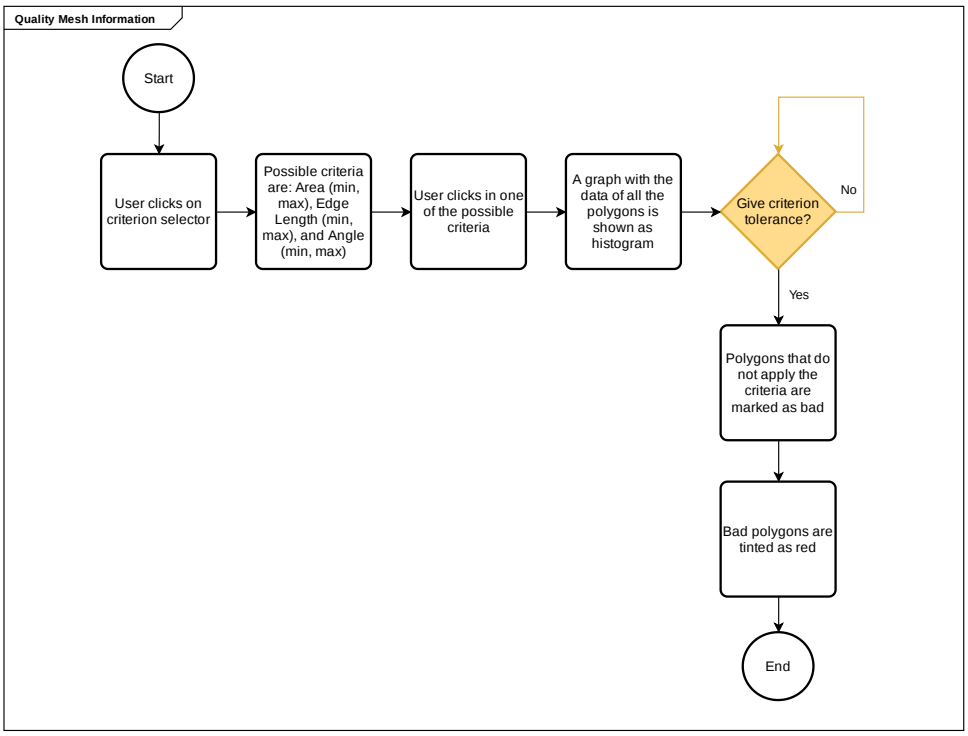


Figure 4.6: Flow chart of the process of quality inspection.

4.4.7. Exporting

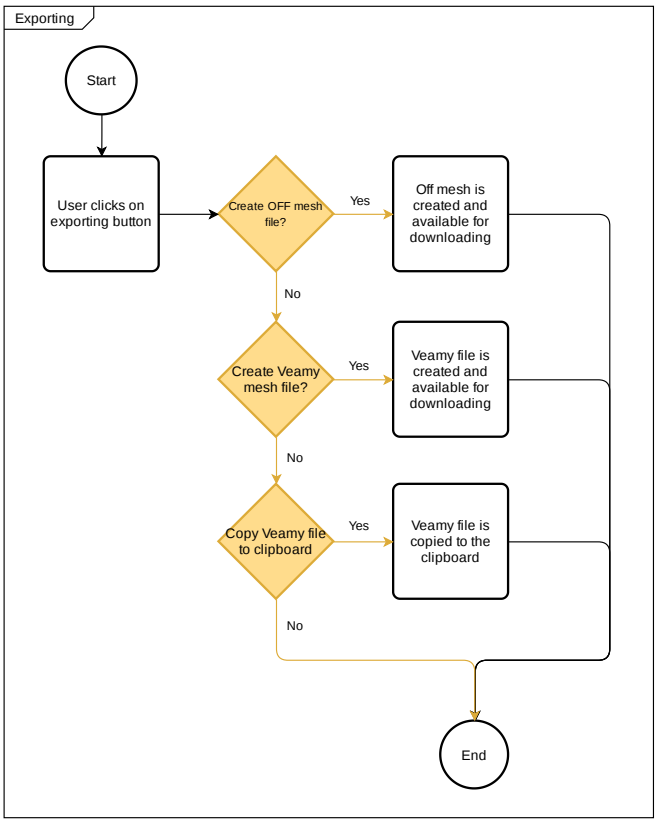


Figure 4.7: Flow chart of the process of exporting meshes.

Finally, if the user wants to download an OFF file or Veamy file with the result of its mesh after its refinements, the user can click on the exporting button, and then download the file. The user can also choose to copy the results and keep them on the clipboard, however, this process is not always possible due to the large number of lines that the file may have. It is important to emphasize that the only mesh available for download is the last one that the user sees, that is, all the intermediate steps that made up the mesh are not recoverable, leaving the implementation of the feature of undoing or redoing a certain refinement to a mesh as future work.

4.5. Model Design

We detail now the class modeling used in our application. First we explain the modeling of the internal classes, where each of the main operations are carried out, such as the refinement of the polygons, the storage of the mesh information or the different algorithm strategies. Subsequently, we cover the frontend modeling of the application through React, visualizing the interactions between the different components created.

4.5.1. Geometry and Selector Region

In our modeling we consider the following basic classes: `Point`, `Edge`, `Polygon`, `Circle` and `Bounding Box`. We separated Circle and Bounding Box from Polygon for convenience, due to the number of particular methods they had, and their later use in the creation of selection regions. To group these classes, we create an interface called **Geometry**, which must implement the `isEqual` and `toString` methods.

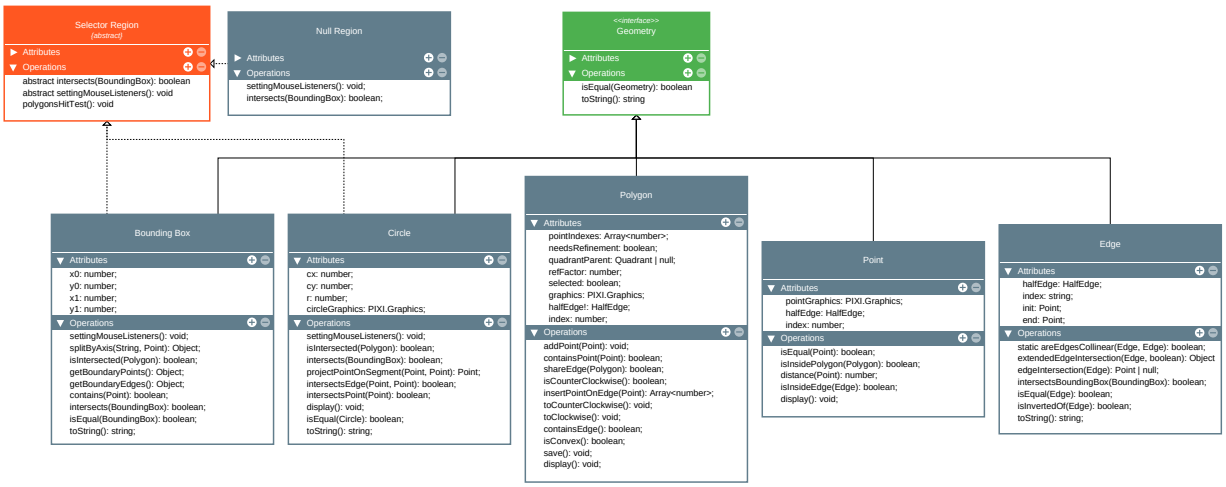


Figure 4.8: UML Diagram for Geometry and Selector Region.

Among the classes that implement the Geometry interface, there are two that are used as a selection region: Circle and Bounding Box. In order to encapsulate the behavior of generating a polygon selection region, we created an abstract class called Selector Region. This class defines the following:

1. **Intersects**: An abstract method that describes how selection geometries intersect a quadrilateral (Bounding Box). This is necessary because for optimization reasons, the intersection between the selection region and a polygon is actually done with a Bounding Box that contains the polygon. This lowers the selection in precision, but increases the speed of the process.

2. **Setting Mouse Handlers**: Another abstract method that indicates how mouse events, such as click, drag and drop, are reflected in the selector geometry in real time.

3. **Polygons Hit Test**: This method is not abstract, and consists of iterating over each of the polygons, extracting its bounding box and detecting those that intersect with

the selection region. This process is done by invoking the `intersects` method mentioned above.

4.5.2. Polygon in detail

The polygon class is one of the most relevant within the implementation, so below we show its modeling in more detail.

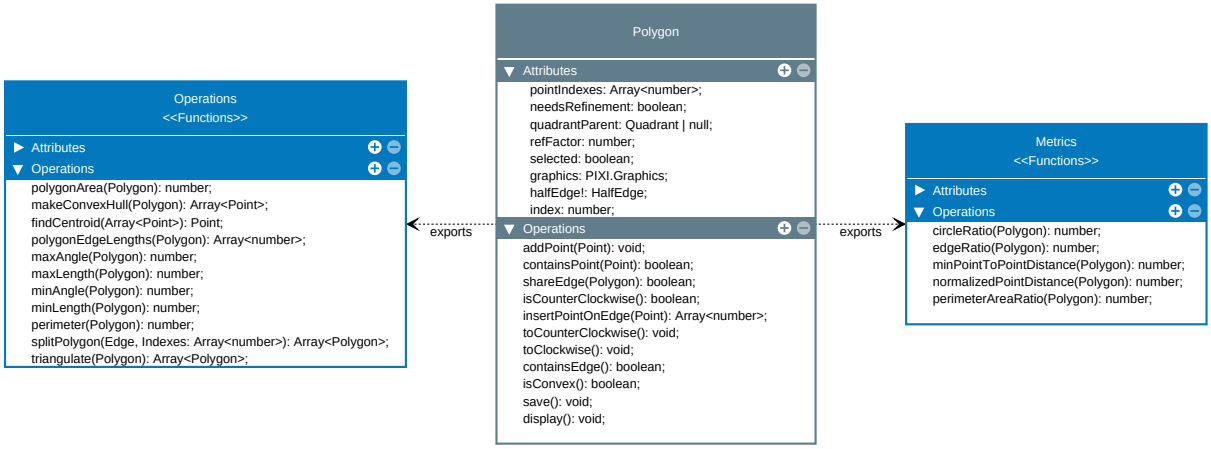


Figure 4.9: UML Diagram of polygon in detail.

As we can see in the Figure 4.9, the polygon class was divided into three sections, due to the great complexity that it acquired. We decided to create a separate module from the class that contains all the operations allowed for a polygon, such as obtaining the maximum area, calculating the convex lock, obtaining the centroid, among others. Also, we create another module that contains all the functions to calculate the metrics of a given polygon. This modeling allows the polygon class to focus on aspects of geometry, such as orientation, adding points to its contour and saving the polygon in storage.

4.5.3. Clipping Algorithms

After the quadtree or kd-tree generates a grid over the input polygon, the next natural step is clip the polygon according to each quadrant generated by the tree. In order to do that, we designed an interface with a method `clip(subjectPolygons, clipperPolygons)`. So every strategy for clipping algorithms must override that method.

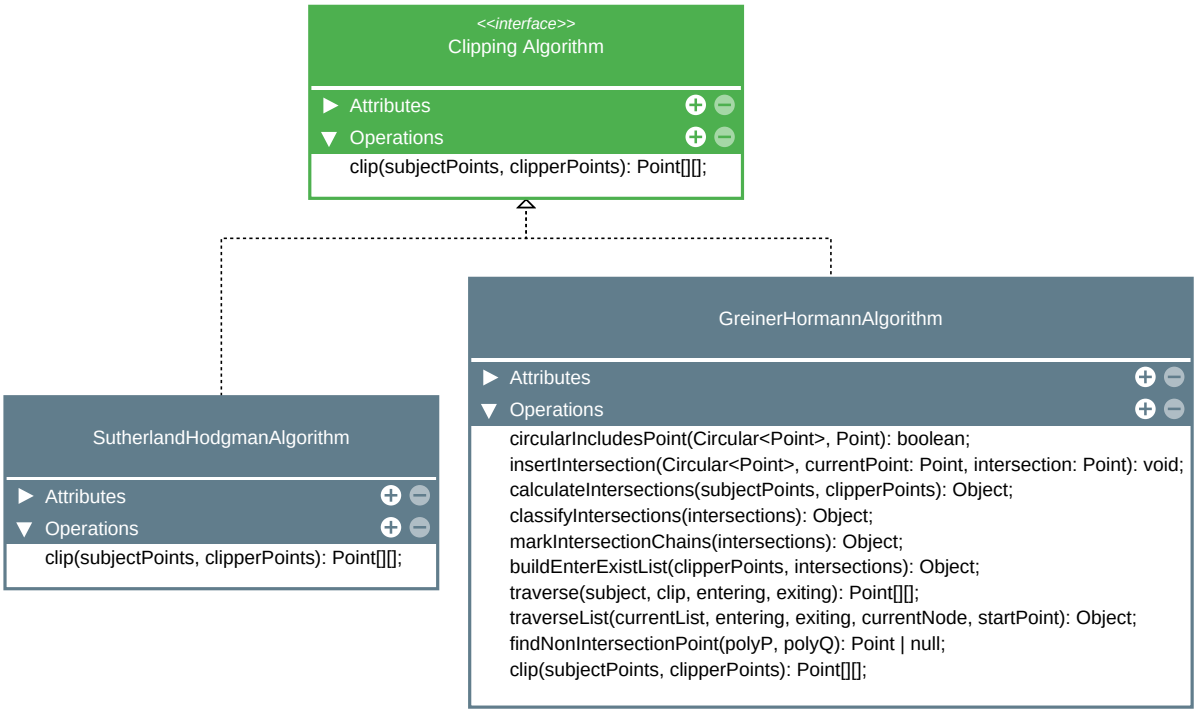


Figure 4.10: UML Diagram for clipping algorithms.

We implemented two algorithms for clipping polygons: Sutherland Hodgmann and Extended Greiner Hormann. Both original algorithms have critical disadvantages for the purpose of this thesis, so we decided to extend the Greiner Hormann algorithm according to the recent work [21] in order to accomplish the polygon cutting with degenerated intersections. The details of the implementation of each algorithm and the extension of the last one, will be on the corresponding section.

4.5.4. Criteria

We model the criteria using the strategy pattern. In this case, all the criteria extends from an abstract class and not from a interface. This is because we need to implement `getTolerance` and `setTolerance` for every criteria. The tolerance is the value that defines the criterion, for example, tolerance for `MinAngleCriterion` is the minimum angle allowed for discerning between a bad or good polygon.

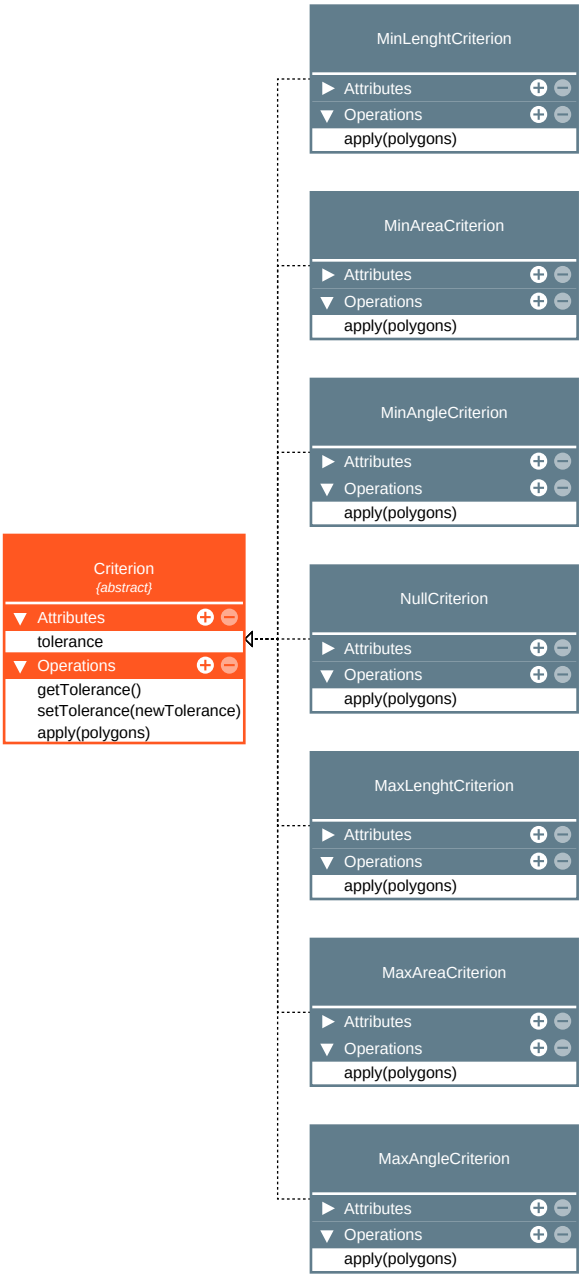


Figure 4.11: UML Diagram for criteria.

The abstract method is `apply(polygons)`, so every criterion must set a flag in the polygons that needs refinement if they not pass the criterion. Every class is a new strategy for applying a criteria to the polygons, making the process of adding a new one a simple task.

We define a `NullCriterion` in case of the user doesn't want to apply a criterion to the mesh he's constructing. This criterion is the default of the application, so if the user wants to apply criteria, he needs to change it to another one. The others criteria are the standard ones used in meshes, related to the length of the edges, the area and the angles of the polygons.

4.5.5. Quality Refining Algorithms

We use an abstract class to define the different types of refinements, implementing in the parent class how a polygon can be triangulated or divided into convex parts (polygonize) and the subsequent arrangement that must be made to the mesh so that it is well constituted. The different quality refinement strategies must implement the abstract `refine` method, so exchanging strategies or implementing new ones is a simple task. The implementation of each of the strategies shown is explained in the corresponding section.

Figure 4.12: UML Diagram for refining algorithms.

We implemented a `NullRefinementAlgorithm`, that does not affect the intersected polygons, in other words, just returns the list of polygons, with no modifications. This *algorithm* is important because it is the default refinement algorithm in the application, and is changed only if the user wants to do so. In other words, if the user doesn't want to refine the mesh, then the default algorithm will leave the mesh intact.

4.5.6. Division Algorithms

For the division algorithms we use the strategy again, and so we can add different partitions of a quadtree without altering its code. We model the *DivisionAlgorithm* as an interface, with three methods: `divide`, `insert` and `divideQuadrant`. Every class that implements the

interface, must implement the logic of insertion and division of a quadtree, in particular, the way the quadrants are divided.

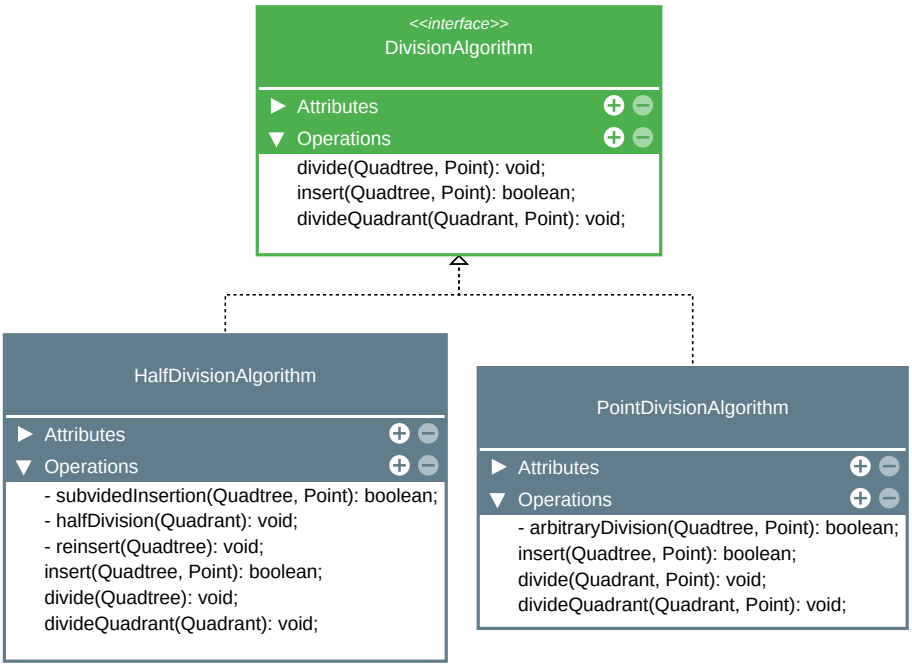


Figure 4.13: UML Diagram for division algorithms.

The default algorithm of quadtree division is `HalfDivisionAlgorithm`, meaning every quadrant is divided using the mid points of the bounding edges. While the other algorithm consists of the arbitrary insertion of points, dividing each of the quadrants according to the coordinates of the points in question. This strategy is called `PointDivisionAlgorithm`, and it also implements the *DivisionAlgorithm* interface.

4.5.7. Mesh

We have designed the Mesh class thinking that its task is to maintain the integrity of all polygons. For this, it has a reference to the initial geometry that generated the mesh, and the quadrants generated by a quadtree or a kd-tree. The methods offered by the class are based on these references to arrange the polygons around one that was refined, reorder the links between the halfedges, among other operations. The Mesh class uses the Integrity class to maintain a consistent state after each operation.

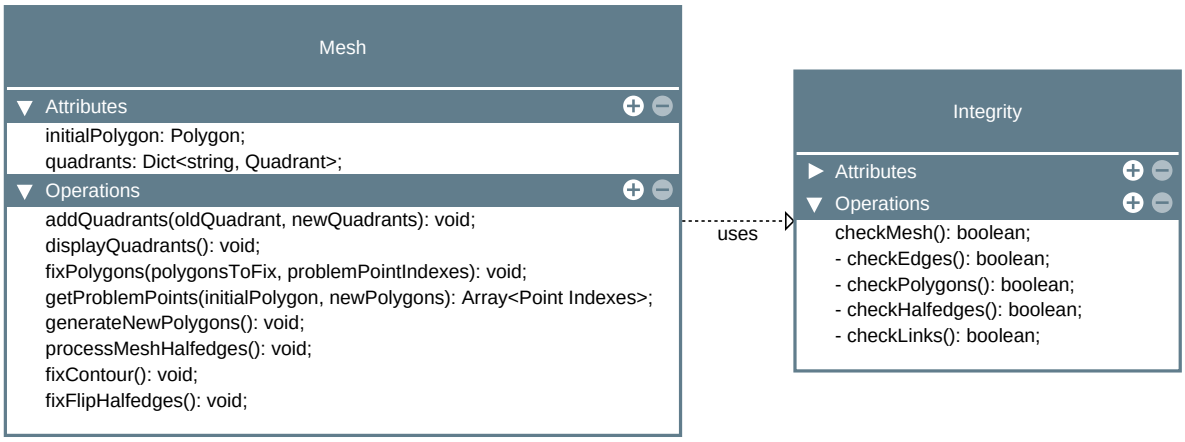


Figure 4.14: UML Diagram for Mesh.

4.5.8. HalfEdges

The HalfEdge class is important for the development of the application, because the consistency of the mesh depends on a good implementation of the links between the HalfEdges. This complexity was extracted in two modules: One dedicated to performing operations such as creating a polygon and inserting an Edge, and another dedicated to querying the mesh, such as obtaining all the polygons that share a certain Edge, the HalfEdges that they share a point, the neighbors of a polygon, among others.

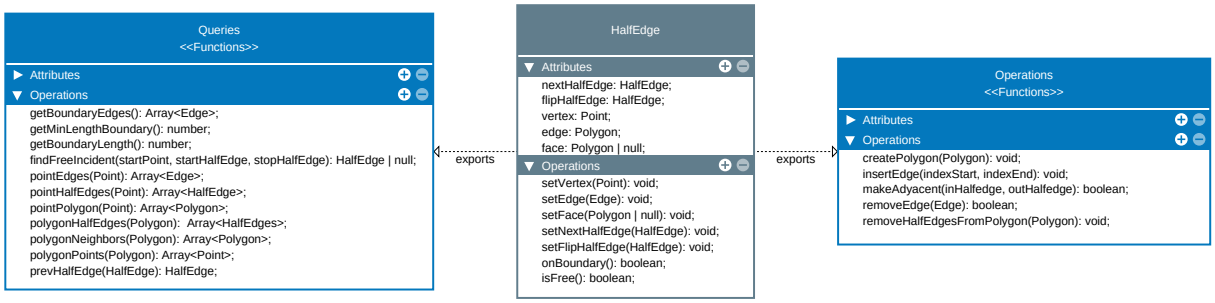


Figure 4.15: UML Diagram for HalfEdges.

4.5.9. Storages

Previously we saw that the Mesh class had no reference to its elements, and this is because its storage was assigned to different Storages classes, one for points, one for edges, and another for polygons. Each of the classes is a global singleton within the application, and they are required to maintain the consistency of the mesh elements.

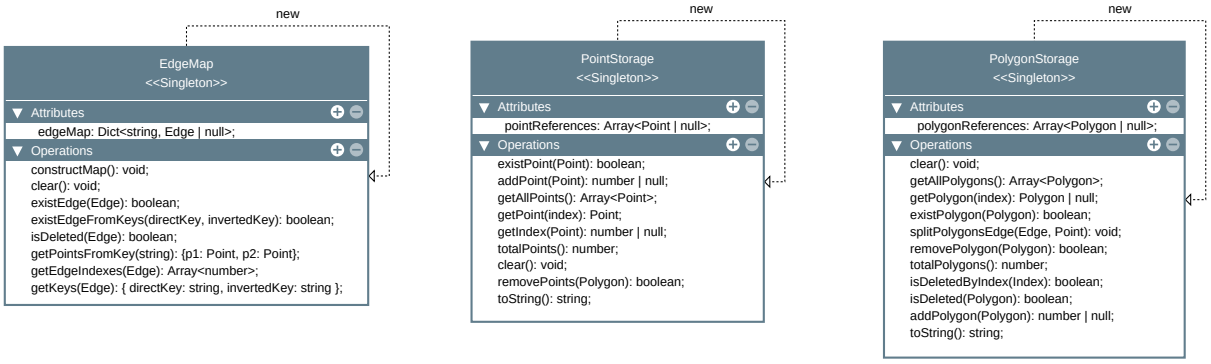


Figure 4.16: UML Diagram for Storages.

Storages must keep the state of items consistent, and handle the cases in which they are added or removed. For this there are associated methods that are responsible for this work, and that depend on each other. For example, the deletion of a polygon in its Storage, invokes the elimination of edges from `Edge Storage` and later the elimination of points from `Point Storage`.

4.5.10. Tree

For the modeling of the trees, we decided to create an abstract class that defines for all the trees the process of inserting random points, however, each class must implement its way of being displayed on the screen and how the points are inserted. Each tree contains Quadrant objects, which represent the leaves of the trees after the insertion of all the points. These Quadrant are responsible for producing each of the initial mesh polygons, which will save a reference to them for further refinement.

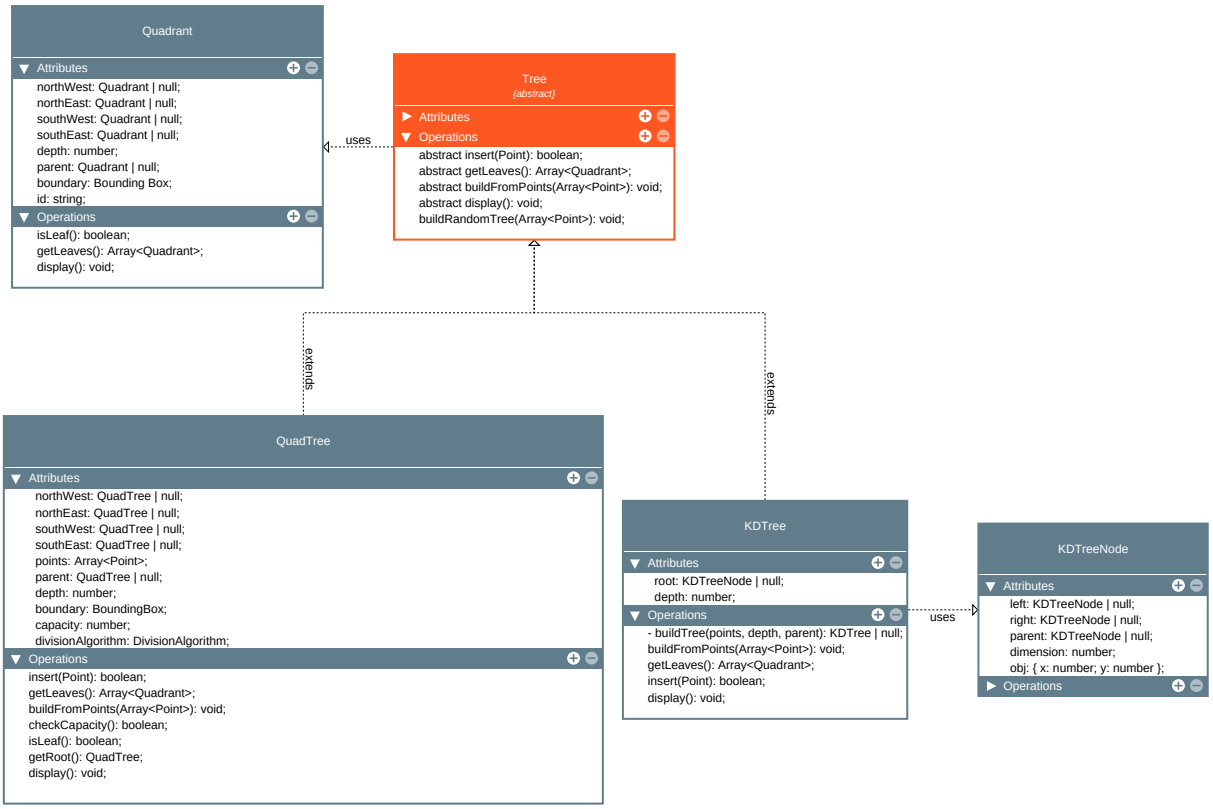


Figure 4.17: UML Diagram for Tree

According to the Figure 4.17, we see that a Quadtree is recursively defined, that is, its children are still Quadtrees. In the case of a KD tree, it was decided to create an additional class for the tree nodes, in order to make the process of division by axes much easier.

4.5.11. React frontend Modeling

React is a framework that facilitates the work of creating user interfaces. These interfaces are created by using *components*, which extend from a predefined component in React. This makes each interface a different class or functional component, with its own states and methods, therefore modularizing the code. In addition, interfaces that wish to display information must implement the `render()` method using the `.jsx` format, which is a Javascript extension for writing elements using tags, in order to format the presentation of the application, the same way as HTML does.

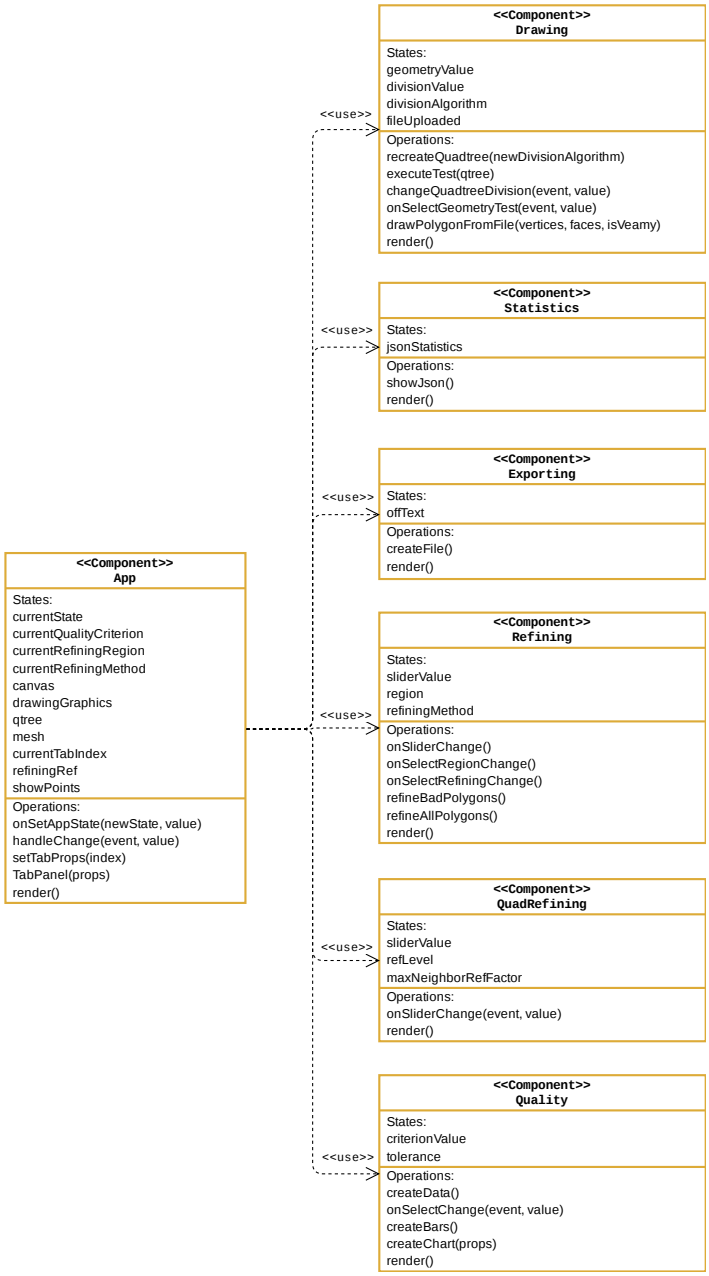


Figure 4.18: UML Diagram for React Components.

In the first instance, we create a main component called `App`, which will be in charge of using the other components (from now on, called *child components*) to correctly display the application. The `App` component has, within itself, the information on the current state in which the application is found (If we are exporting to an OFF file, if we see statistics, or we are refining), as well as the essential elements such as the drawing canvas, the polygon mesh, and the quadtree.

All this information is passed to the child components through the React properties, called `props` in the framework, in conjunction with a function called `onSetAppState`, which consists of a callback function, which modifies the status of the global component from a child component. This is how the application is always updated, depending on the actions carried

out by each child component, therefore, it is enough to implement the logic of refinement, statistics, among others, in each of these components, and then update the application global, so that the information is consistent.

4.6. Experimental Design

We have to conduct experiments to compare the meshes generated through generalized quadtrees and kd-tress with triangulations and other types of meshes. For this, what we do is to test certain geometric domains both in our software, as in Triangle, to later compare the obtained metrics and establish conclusions.

The task described above requires that the input files for each software be compatible with each other, which is not possible without preprocessing. This is because the formats in which the geometries are presented vary between one program and another.

In the first instance, what we did was only compare the results obtained by our application between the different algorithm strategies:

- We compare the initial meshes of each of the tree types that we implement in the thesis work, with their different strategies, and we see which of them have the fewest elements, what their average angles and edges are, and how long they take to generate the initial meshes.
- Subsequently, we verified how the refinement by quadtree behaves successively, comparing the procedure with and without considering the neighbors of the polygons.
- Then we applied a quality refinement to each of the polygons of each of the initial meshes generated on a unicorn geometry, and we compared the relevant data mentioned above to verify which ones have the best performances.
- With a given initial mesh, we successively applied the same quality criteria to all the polygons of the mesh, in order to study how it behaves while more polygons exist, in terms of time and number of elements generated. Finally, we study how quality metrics behave through different refinements for the same quality refinement algorithm, in order to study whether the result improves or worsens.

Finally, given a geometry created by our polygon mesh generator, what we do is compare each of the relevant metrics with triangle, such as the number of elements, the minimum angle, minimum length and computation time used. Then we compare each of the quality metrics for both programs, in order to check the performance of the meshes generated by our generator, with respect to the Triangle triangulations.

Chapter 5

Implementation

In this chapter we describe the main algorithms and data structures inside our mesh generator based on kd-trees and modified quadtrees, and the visualizer that helped to see if operations were correct. First we explain the implementation of the web application, and after we address the algorithms behind that accomplish the generation of polygonal meshes.

5.1. Mesh Representation

In this section we explain how we implement those essential elements for the representation of a polygon mesh, which are: Vertices, Edges, Polygons and Halfedges. For each element, we explain how it was represented as a class, and the implementation of their respective Storages.

5.1.1. Vertices

Vertices are the most basic elements and are responsible for giving consistency to our application. We first define the information that an object of the `Point` class stores, and then, how it is stored to keep the information in a consistent way.

5.1.1.1. Vertex representation

The points or vertices of the meshes have basic information such as their components on the X axis and on the Y axis. Additionally, we add three fields: A reference to a halfedge, an index corresponding to the place the point occupies in the Storage array, and a reference to the `graphics` object for faster rendering.

```
1 class Point implements Geometry {
2   pointGraphics: PIXI.Graphics;
3   halfEdge!: HalfEdge;
4   index: number;
5
6   constructor(public x: number, public y: number) {
7     this.pointGraphics = null;
8     this.index = -1;
9   }
```

```
10 }
```

5.1.1.2. Vertex Storage

The implementation of vertex storage, called `PointStorage`, consists of an array that has references to objects of the Point class. When a point is added to the array, it will always remain with that index, unless explicitly removed. In that case, that index can no longer be used again, thereby ensuring that the consistency of the information is maintained. This means that spaces are left with flags, which indicate that the point in that index is undefined, which guarantees that the already existing point indexes are always used.

Algorithm 14 Adding a Point inside the PointStorage

```
1: Input
2:     point        The Point to be saved
3: Output
4:                  Returns null if the point couldn't be saved.
5:                  Otherwise, returns the index of the point.

6: function AddPoint(point)
7:     Ensure the point is correct
8:     if not ExistPoint(point) then
9:         add the point to the array of the PointStorage
10:        set the point index property as the index of the point in the array
11:        return point.index
12:    return null
13: end
```

The `addPoint` function in the first instance performs a verification of the properties of the delivered point, making sure that they are correct. Later in line 8, we see the great dependency that exists in the `existPoint` function that is detailed below. If the point is missing, then it is added, adding to the point the index it occupies in the array, otherwise the point cannot be added.

Algorithm 15 Verify if a Point exists inside the PointStorage

```
1: Input
2:     point        The Point to be saved
3: Output
4:                  Returns true if the point exists. false otherwise.

5: function ExistPoint(point)
6:     Ensure the point is correct
7:     if index of point is equal to -1 then
8:         foreach storedPoint in all stored points in PointStorage do
9:             if storedPoint is equal to point then
10:                assign point.index to storedPoint.index
11:                return true
12:        end
13:        return false
14:    Ensuring that the index does not point to an undefined location in PointStorage
15:    Ensure that the point saved in the storage is equal to point
16:    return true
17: end
```

The `existPoint` function is responsible for performing the important consistency check of the points inside the `PointStorage`. First it is in charge of verifying if the point is well defined, and later on line 7, it verifies if the index is -1 with two possible cases:

1. If it is, the point is a *candidate* to be saved, however, there is a possibility that a new instance of an already existing point has been created (therefore it would have index -1 with the same coordinates), and in that case, we would have two repeated points in the storage. To avoid that, we sweep through all the saved points (line 8) and perform the heavy comparison process of identifying if the float coordinates are equal with an epsilon tolerance. If they are the same, then the index of the `point` instance is changed to the index of the point stored in the array, and it returns true. If no point is the same, the function can safely return that point does not exist.
2. In case the index is not -1, we should still make consistency comparisons. The first is to verify that the said index points to a place that is defined in the array, that is, that there is a point in the array in that index. If so, then we must ensure that point is equal to the `point` we want to add. Only in that case, the function can safely return that the point actually exists.

With this procedure, we optimize the number of times the sequential lookup is accessed through all the saved points, since it is only done with the `Point` objects that are created for the first time and have index -1. For all the rest, checking if a point exists is done at $\mathcal{O}(1)$, going directly to the index of the array.

Using consistent indexes makes other tasks also $\mathcal{O}(1)$. For example, to determine if two points are equal: it is enough to compare the integer indices (if both are different from -1), since by construction, if two points have the same coordinates, they must have the same

indices. This allows us to avoid floating point comparisons and their associated errors, increasing the performance of the application.

Regarding the elimination of points, it is a complex process that introduces many inconsistencies in the connectivity of the mesh if it is not done well. This is why the elimination process is done according to a polygon that you want to eliminate, as follows:

Algorithm 16 Delete points from PointStorage

```
1: Input
2:     polygon     The polygon object to which its points are removed
3: Output
4:                 Returns true after all the points are processed.

5: function RemovePoints(polygon)
6:     Ensure the polygon is well defined
7:     foreach pointIndex in point indexes of polygon do
8:         canBeDeleted ← true
9:         foreach poly in all other polygons in the neighborhood do
10:            if poly contains polyIndex then
11:                canBeDeleted ← false
12:        end
13:        if canBeDeleted is true then
14:            ind ← Get index from polygon
15:            set array[ind] of PointStorage to null
16:    end
17:    return true
18: end
```

For each index of points that make up the polygon, we initially set a *flag* on line 8 as a possible *candidate* to be eliminated. Later we inspect the neighborhood of the polygon to delete, and if there is any other polygon that shares this point, then its elimination would bring inconsistencies, so we change the *flag* to false. Finally on line 13, we ask if the flag is still true, in that case the function can safely delete the point, acquiring its index and setting the `PointStorage` array at that position as null.

5.1.2. Edges

5.1.2.1. Edge representation

An `Edge` to be wel defined in its construction depends on two objects of the `Point` class: *initPoint* and *endPoint*. Therefore, when creating a new edge, we must make sure that the points are added to the PointStorage first. Taking that into consideration, the information stored for an object of the `Edge` class are the references to the *initPoint* and *endPoint* points, a reference to the associated Halfedge, and the index that identifies the Edge, which is a `String` formed by the indexes of the initPoint and endPoint points joined by a hyphen.

```
class Edge implements Geometry {
  halfEdge!: HalfEdge;
  index: string;
  init: Point;
  end: Point;

  constructor(initPoint: Point, endPoint: Point) {
    const processPoints = (init: Point, end: Point): string => {
      PointStorage.addPoint(init);
      PointStorage.addPoint(end);
      const directKey = `${init.index}-${end.index}`;
      return directKey;
    };

    this.index = processPoints(initPoint, endPoint);
    this.init = initPoint;
    this.end = endPoint;
  }
}
```

5.1.2.2. Edge Storage

Due to the nature of the edge index, we use a HashMap called `EdgeMap` to save them. For example the edge $E_1 = (P_1, P_2)$, with P_1 and P_2 `Point` objects of indices i_1 and i_2 respectively, is saved as "$i_1 - i_2$" in the Edge Storage. We save the Edges only once, then if we ask for the existence of the edge in the other direction $E_2 = (P_2, P_1)$ EdgeMap replies affirmatively, but it's not saved in reality. To achieve this, when verifying the existence of an edge $E = (P, Q)$ in the EdgeMap, we extract its index and ask if the key $P.index$-$Q.index$ or $Q.index$-$P.index$ exists. If true, then the edge is saved, otherwise the edge does not exist on the mesh.

The creation of Edges for an initial mesh assumes that all the polygons are well generated according to our restrictions, that is, that they do not intersect or have repeated points. The process is simple, and is shown below:

Algorithm 17 Adding edges to EdgeMap

```
1: Output
2:                 Mutates the EdgeMap adding all the edges of the polygons

3: function ConstructMap
4:     Ensure the EdgeMap is empty
5:     foreach polygon in all the polygon in the mesh do
6:         foreach firstIndex in point indexes of polygon do
7:             get the nextIndex after firstIndex in polygon
8:             directKey ← create String 'firstIndex-nextIndex'
9:             invertedKey ← create String 'nextIndex-firstIndex'
10:            if not ExistEdgeFromKeys(directKey, invertedKey) then
11:                (P1, P2) ← get the points from PointStorage using the directKey
12:                newEdge ← new Edge(P1, P2)
13:                create entry into the EdgeMap associating directKey with newEdge
14:        end
15:    end
16: end
```

What the `constructMap` function does is go through each of the saved polygons of the mesh, and for each one, go through the indexes of the points that make it up. Consecutive points are obtained, creating a new border, and it is added to the `EdgeMap` as long as it does not exist, so this function depends on `existEdgeFromKeys`. Due to the nature of a HashMap, we can verify if an edge is found or not, thanks to the fact that the keys are made based on the indexes of the vertices, which we know are not duplicated.

Algorithm 18 Check if edge exists in EdgeMap

```
1: Input
2:     directKey        String representing the index key of the edge
3:     invertedKey      String represented the inverted index key of the edge
4: Output
5:                 True if the edge exists, False otherwise

6: function ExistEdgeFromKeys(directKey, invertedKey)
7:     Ensure the directKey and invertedKey are well defined
8:     return EdgeMap has directKey or invertedKey inside its keys
9: end
```

Regarding the elimination of edges, the task is made easier thanks to the incorporation of halfedges, since we can immediately access the opposite polygon that occupies the same edge if it exists. In that case you cannot delete the edge because it introduces inconsistencies, so you can only delete the edges that are on the boundary and the associated polygon is deleted, or the edge is shared between two polygons and both are deleted. To delete an Edge, due to the properties of the HashMap, just search for the record that contains the direct or inverted key of the Edge, and delete it from the Map, as shown below:

Algorithm 19 Delete edge from EdgeMap

Input
    edge The *edge* object to be deleted
Output
    Returns true if the *edge* could be deleted. False otherwise.

function RemoveEdge(edge)
    Ensure the *edge* is well defined and it's not already deleted
    directFace ← *Polygon of edge*
    flipFace ← *Polygon associated to edge in the other direction*
    deletable ← *False*
    if *halfedge of edge is on boundary* **then**
        if *directFace exists and is in process of elimination* **then**
            deletable ← *True*
        else if *flipFace exists and is in process of elimination* **then**
            deletable ← *True*
    else
        if *both flipFace and directFace are deleted* **then**
            deletable ← *True*
    if *deletable* **then**
        (directKey, invertedKey) ← *get keys from edge*
        Delete *directKey* key from `EdgeMap` if it exists.
        Otherwise, delete *invertedKey* key from `EdgeMap`.
        return True
    else
        return False
end

5.1.3. Polygons

5.1.3.1. Polygon representation

A `Polygon` in our application is represented as an array of integers, corresponding to the indices of the Point objects that make it up. The polygon is well defined if the points are properly stored in `PointStorage`, and if there is no repetition of indexes in their definition.

```
export default class Polygon implements Geometry {
  pointIndexes: Array<number>;
  needsRefinement: boolean;
  quadrantParent: Quadrant | null;
  refFactor: number;
  selected: boolean;
  graphics: PIXI.Graphics = new PIXI.Graphics();
  halfEdge!: HalfEdge;
  index: number;

```

```
11    constructor(points: Array<Point> = []) {
12      const addPoints = (pointsList: Array<Point>): Array<number> => {
13        for (let i = 0; i < pointsList.length; i++) {
14          const p = pointsList[i];
15          PointStorage.addPoint(p);
16        }
17        return PointStorage.getIndexesFromPoints(pointsList);
18      };
19
20      this.pointIndexes = points.length === 0 ? [] : addPoints(points);
21      this.needsRefinement = false;
22      this.quadrantParent = null;
23      this.refFactor = -1;
24      this.selected = false;
25      this.index = -1;
26    }
27  }
```

The creation of a polygon can be from a list of Point objects, or create an empty polygon and then add the points. The relevant information for a polygon is as follows:

1. A reference to the point indexes an array.
2. A boolean that tells us whether or not a polygon needs refinement.
3. A reference to the quadrant that generated it in case it was generated from a quadtree or kd-tree. This field is null if the polygon is incorporated from a mesh already made.
4. A integer numeric factor that tells us how many times the polygon has been refined.
5. Finally an integer numeric index indicating the position of the polygon in the storage. If the index is equal to -1 it means that the point was never stored in the `PointStorage`.

5.1.3.2. Polygon Storage

Saving polygons is equal to how points were saved, using an array of Polygon objects called `PolygonStorage`. Each polygon occupies a space in the array, adopting the index it occupies as its identification property. The advantage that polygons are defined as an array of integer indices is that we avoid floating point operations when checking if a point is part of the points that make up the polygon, or if one polygon is equal to another. In the first case, it is enough to see if an integer belongs to an integer array, and in the second, if the ordered index lists of both polygons are equal.

Algorithm 20 Add polygon to PolygonStorage

```
1: Input
2:     polygon     The polygon object to be stored
3: Output
4:                 Returns the index of the stored polygon. Null if it could not be saved.

5: function AddPolygon(polygon)
6:     Ensure the polygon is well defined
7:     if not ExistPolygon(polygon) then
8:         add the polygon to the array of the PolygonStorage
9:         set the polygon index property as the index of the polygon in the array
10:        return polygon.index
11:    return null
12: end
```

A complex process for maintaining the consistency of information is the elimination of polygons. Deleting a polygon implies removing all the vertices and edges that define it, however, that can undefine other polygons that share these elements. The elimination process delegates its work to the functions of each Storage, to ensure the correct result.

Algorithm 21 Delete polygon from PolygonStorage

```
1: Input
2:     polygon     The polygon object to be deleted
3: Output
4:                 Returns true if the polygon could be deleted. False otherwise.

5: function RemovePolygon(polygon)
6:     Ensure the polygon is well defined
7:     if polygon has index equal to -1 then
8:         return false
9:     potentialDeletedEdges ← get the edges of polygon
10:    index ← get the index of polygon
11:    set array[index] of PolygonStorage to undefined
12:    foreach edge in potentialDeletedEdges do
13:        DeleteEdge(edge) from EdgeMap
14:    end
15:    RemovePoints(polygon) from PointStorage
16:    return true
17: end
```

5.1.4. Halfedge Connectivity

5.1.4.1. Halfedge definition

For the implementation of halfedge, the minimum information was saved in each instance of the objects, being these:

1. Reference to next and opposite halfedge
2. Reference to the vertex that originates the halfedge
3. Reference to the edge to which it belongs
4. Reference to the polygon in which the halfedge is located

The above described in code, looks like this:

```
export default class HalfEdge {
  constructor() {
    this.nextHalfEdge = undefined;
    // reference to the opposite halfedge associated with the edge
    this.flipHalfEdge = undefined;
    this.vertex = undefined;
    this.edge = undefined;
    this.face = undefined;
  }

  onBoundary() {
    return !this.flipHalfEdge;
  }

  isFree() {
    return !this.face;
  }
}
```

In addition, two auxiliary methods were created that allow us to ask if a halfedge is on the edge or not. For this, it is enough to ask if the opposite halfedge exists or not, since in case of being on the edge, there is no external face and therefore the opposite is not defined. In addition, there is a method that tells us if the halfedge is free or not, asking if there is an assignment of it to any polygon.

5.1.4.2. Creating halfedges for a polygon mesh

For the creation of halfedges given a mesh of polygons, we must assume that the mesh is manifold, so it will be well constituted and each arc shares a maximum of 2 polygons, there are no discrepancies between the edges of the polygons, and that they always have a vertex of beginning and another end. Under these assumptions, the function goes through each of the polygons of the mesh, and subsequently, goes through each of the points of each polygon, performing the following procedure.

Algorithm 22 Creating halfedges for a polygon mesh.

```
1: function ProcessMeshHalfedges
2:     foreach poly in Polygons of the Mesh do
3:         foreach currentPoint in Points of poly do
4:             nextPoint ← get the next point after currentPoint
5:             edge ← get Edge with points currentPoint-nextPoint
6:             hEdge ← new HalfEdge()
7:             set currentPoint as the origin vertex of hEdge
8:             set poly as the polygon of hEdge
9:             set edge as the associated edge of hEdge
10:            if edge has associated halfedge then
11:                set the opposite of hEdge to edge.halfEdge
12:                set the opposite of edge.halfEdge to hEdge
13:            else
14:                set hEdge as the associated one to the edge
15:            set hEdge as next of previousHalfEdge
16:            set hEdge as the associated one to the currentPoint
17:        end
18:        set the first halfEdge created from poly, as the associated one of poly
19:        set the first halfEdge as the next of lastHalfEdge
20:    end
21: end
```

Since the halfedges are centered on the edges, we must go through each of the points as it is done in line 3 to access the associated edges. Let us call P_i the current point and P_{i+1} the next point in counterclockwise order of the current polygon that is being reviewed, called $\mathcal{P}$. With these points we obtain the only mesh border defined by those points, called $\mathcal{E}$, on line 5, and later we create a new HalfEdge with no assignments on line 6, called *hEdge*.

Now the algorithm must assign all possible references for vertex, border, polygon, and halfedge.

1. Starting with halfedge, which we called *hEdge*, we set P_i as the origin point, $\mathcal{P}$ as its associated polygon, and edge $\mathcal{E}$ as its associated edge.

2. Subsequently, we must make the assignment for $\mathcal{E}$. If $\mathcal{E}$ already has an associated halfedge, then it means that we are visiting the edge but in an opposite direction, so we set that halfedge and *hEdge* as opposites between them. In case there is no associated halfedge, then we set the halfedge of $\mathcal{E}$ as *hEdge*.

3. Then in line 15 we update the `next` reference of the previous halfedge that was created, with the current *hEdge*.

4. Finally we assign the halfedge *hEdge* as the halfedge associated with the vertex P_i.

After processing all the points of the polygon, we must assign to $\mathcal{P}$ some halfedge of the newly created that is free (that is, without assigning a polygon). We arbitrarily assign the first halfedge created.

To end the halfedge cycle, on line 19 we assign the `next` reference of the last halfedge, with the first halfedge created. In this way, the polygon can be traversed in counterclockwise order using only the halfedges.

5.1.4.3. Applying operations to polygons

Thanks to the halfedge structure, the operation on polygons has a very well defined base structure, so it serves as a template for functions that want to obtain information or elements of a polygon. To do this, start from the halfedge associated with the polygon, and make a copy of its reference. Subsequently, a reference is created whose initial value will be the same halfedge of the associated polygon, but which will change to the halfedge corresponding to the next parameter. This process will continue until the changing reference is equal to the starting halfedge again.

Algorithm 23 Template of a function that operates on a polygon.

```
1: Input
2:     polygon     An instance of a Polygon on which to perform an operation

3: function PolygonOperation(polygon)
4:     startHalfedge ← get halfedge of polygon
5:     he ← get halfedge of polygon
6:     do
7:         <Do operation>
8:         he ← he.nextHalfedge
9:     while he is not equal to startHalfedge
10: end
```

5.1.4.4. Obtaining the neighbors of a polygon

Following the function model described in the section, we have created a function that obtains the neighbors of a polygon in constant time $\mathcal{O}(1)$. To accomplish that, we loop through the polygon halfedges and each of them, extract the polygon from the opposite halfedge (if it exists), and store its index in an array. After having visited all the halfedges, the function returns the array of polygon indexes corresponding to all the neighbors.

Algorithm 24 Obtaining the neighbors of a polygon

```
1: Input
2:     polygon     Polygon to which its neighbors are obtained
3: Output
4:                 List of neighbors indexes

5: function PolygonNeighbors(polygon)
6:     startHalfedge ← get halfedge of polygon
7:     he ← get halfedge of polygon
8:     neighbors ← [ ]
9:     do
10:        if he is not a boundary halfedge then
11:            add index of polygon associated with he.flipHalfEdge
12:        he ← he.nextHalfedge
13:    while he is not equal to startHalfedge
14:    return neighbors
15: end
```

5.1.4.5. Add a Polygon to the Mesh keeping while maintaining connectivity

The construction of halfedges for a static mesh allows us to perform operations very quickly, such as obtaining neighbors, obtaining the edges of a polygon, and obtaining all the edges that affect a vertex. This is because connectivity is kept constant and consistent. However, in our work we need to modify the polygon mesh, so adding a polygon not only implies maintaining consistency in the saved information, but also in the connectivity of the halfedges.

Algorithm 25 Adding a polygon inside a Mesh

```
1: Input
2:     mesh       Mesh of polygons
3:     polygon    Polygon to create inside the mesh
4: Output
5:                Mesh is mutated, with polygon inside it

6: function CreatePolygon(mesh, polygon)
7:     polyPoints ← get the points of polygon in CCW order
8:     foreach currentPoint in polyPoints do
9:         he ← new HalfEdge()
10:        nextPoint ← get next point of currentPoint
11:        edge ← get Edge with points currentPoint-nextPoint
12:        if edge exists then
13:            existentHalfedge ← extract the halfedge of edge
14:            if currentPoint.index is equal to existentHalfedge.vertex.index then
15:                he ← existentHalfedge
16:            else
17:                set existentHalfedge as the opposite of he
18:                set he as the opposite of existentHalfedge
19:        else
20:            edge ← new Edge with points currentPoint and nextPoint
21:            set he as the associated halfedge of edge
22:            save edge into the EdgeMap
23:        set edge as the associated edge of he
24:        set polygon as the associated polygon of he
25:        set currentPont as the associated vertex of he
26:        set next of previous halfedge as he
27:    end
28:    set the first halfedge as the associated polygon halfedge
29:    set the next first halfedge as the last halfedge created
30: end
```

This function is similar to that of creating halfedges for any polygon mesh, except for the fact that you must ask about the existence of a border that belongs to the polygon you want to add and that is already inside the polygon mesh. In this case, from line 12 to 18, the references of the halfedges are updated. Otherwise, the border must be created and added to the `EdgeMap` (which was not done in the Algorithm 22 since it assumed that all the polygons were already created and saved).

5.2. Web Application Views

The web application we created was made using the React library and the Javascript language, for the creation of the user interface. This is because React simplifies many of the tasks that concern the creation of views, and how information flows between its components.

In addition to the visualization of the mesh generation, we integrate PixiJS, which is a quick library for rendering 2D elements using WebGL. We chose this library due to its speed and capacity of drawing thousands of polygons on screen, transferring processes to the GPU, through a friendly implementation on top of WebGL.

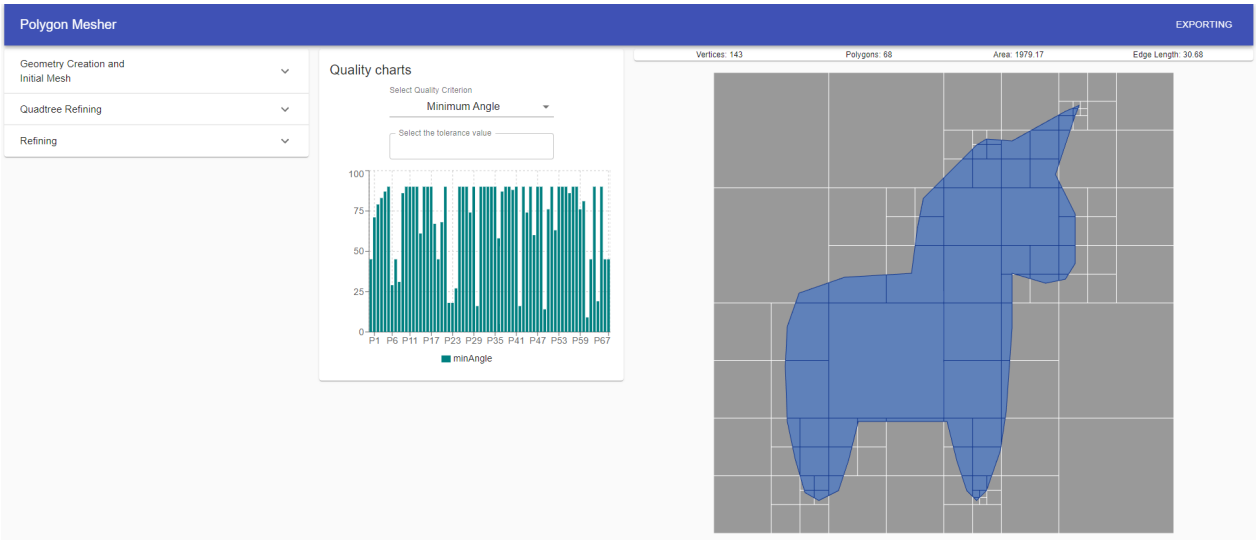


Figure 5.1: General view of the application.

The main view of the application consists of 3 main components.

1. On the left, it corresponds to the operation panels of the application, where we can select the initial creation of a polygon mesh using an .OFF file or an own creation drawing the points on the canvas, refine the mesh using a quadtree with division using the midpoint, or refine using the application's own algorithms.
2. In between, there is the graphics section, which shows us how the different quality metrics are distributed, such as the minimum and maximum angle, the minimum and maximum edge length, and the area of each polygon.
3. On the right, we find the canvas, where the polygon mesh is drawn and the polygons are displayed. Above it is a summary of basic information such as the number of vertices, polygons, and the average area of each polygon and the length of its edges.

Next, we will see how the PixiJS library was integrated, to display the canvas in an application made with the React framework.

5.2.1. Integrating PixiJS

The integration of PixiJS with React is done using a component for rendering the canvas in the web application. Using the `componentDidMount()` function, we determine what are the things that are done after the component is rendered in the application. In this particular case, what we do is create the PixiJS application, using the corresponding parameters, create a camera display layer to zoom on the canvas, and add the necessary information to the application status, such as the tree initial, and references to the application.

```
class PixiJS extends Component {
  constructor(props) {
    super(props);
    this.pixiContainer = null;
    this.app = null;
  }

  async componentDidMount() {
    const { onSetAppState, p5Props, setStateAsync } = this.props;
    const app = new PIXI.Application({
      width: 700,
      height: 700,
      transparent: false,
      antialias: true,
      backgroundColor: CONSTANTS.COLOR.BACKGROUND,
    });
    this.app = app;
    this.pixiContainer.appendChild(app.view);

    // create viewport
    const viewport = new Viewport({
      screenWidth: window.innerWidth,
      screenHeight: window.innerHeight,
      worldWidth: 700,
      worldHeight: 700,
      // the interaction module is important for wheel to work properly when renderer.view
    ↪ is placed or scaled
      interaction: app.renderer.plugins.interaction,
    });

    app.stage.addChild(viewport);
    app.stage.interactive = true;

    // activate plugins
    viewport
      .drag()
      .pinch()
      .wheel()
      .decelerate();

    // setting the pixiApp and viewport into the app state
    await setStateAsync({ pixiApp: app });
    onSetAppState({ pixiViewport: viewport });

    // Creating the boundary
    const boundary = new BoundingBox(0, 0, 700, 700);
    const tree = new QuadTree(boundary, 1, new PointDivisionAlgorithm());

    // Create the quadtree
    onSetAppState({
```

```
50        tree,
51      });
52    }
53
54    render() {
55      const { classes } = this.props;
56      return (
57        <div
58          ref={(thisDiv) => {
59            this.pixiContainer = thisDiv;
60          }}
61          className={classes.pixiCanvas}
62        />
63      );
64    }
65  }
```

Note that we have used the `ref` property of React, to obtain information about the Div that the PixiJS canvas will contain. That is why in the `componentDidMount()` function this information is known, and can be saved, for the correct creation of the PixiJS application.

5.2.2. Geometry creation and initial panel

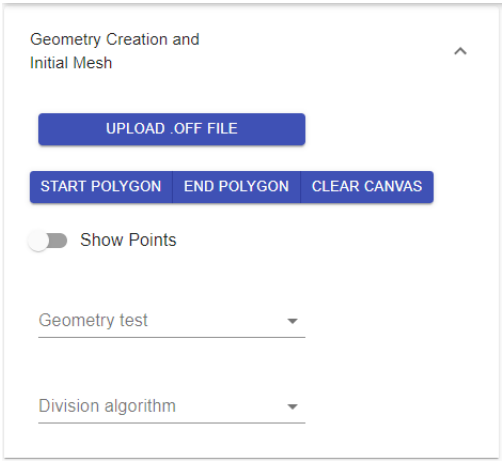


Figure 5.2: Panel to create initial meshes.

This panel is in charge of creating initial meshes on the part of the user, whose main functions are as follows:

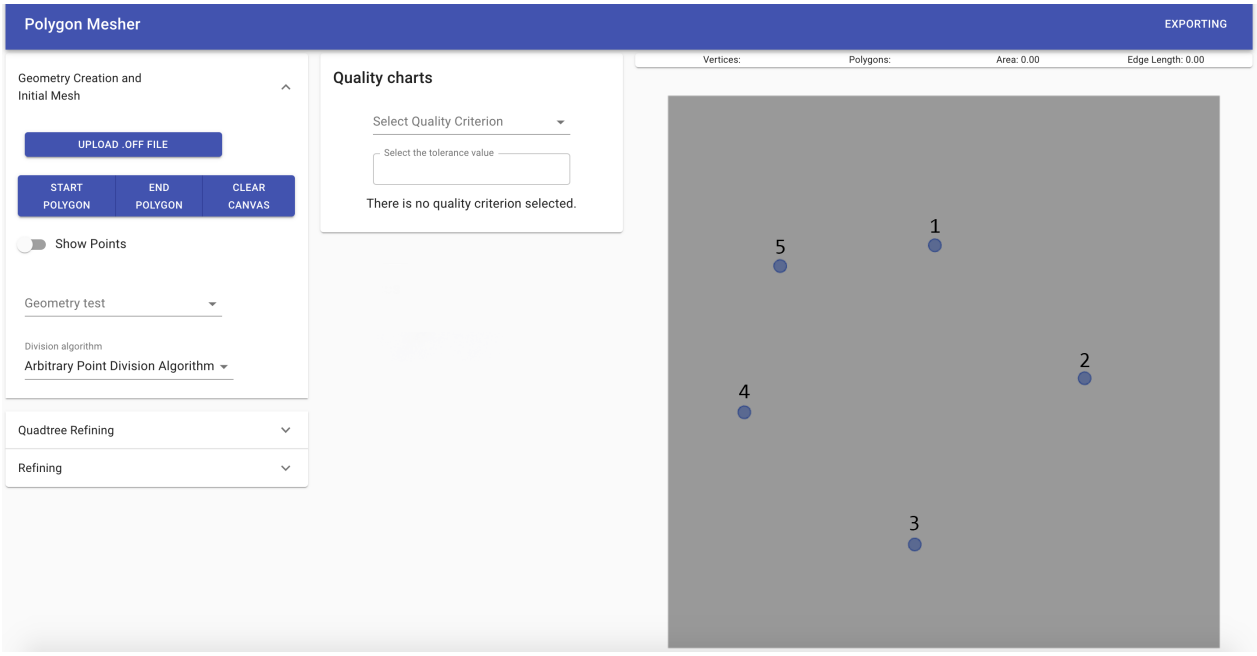
1. Upload a .off file for processing in the application
2. Create a polygon by hand, by clicking on the canvas to insert the points.
3. Use one of the tests already created to verify how the application behaves.
4. Change the type of tree that will serve as the data structure for creating meshes

If the user chooses to create a polygon from scratch (i.e. without using an .off file), he can select what type of tree the user wants for the initial mesh shaping:

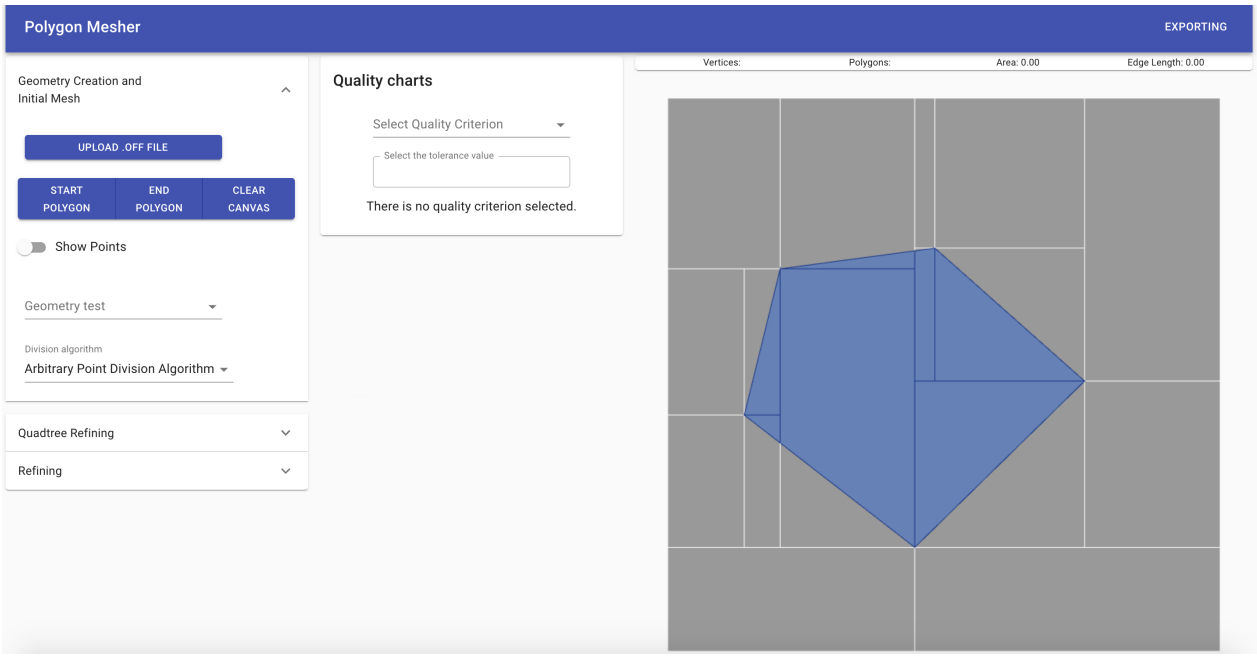
1. A quadtree that divides in half, until each point is only in its quadrant. Uses Half Point Division Algorithm.
2. A quadtree that divides using the inserted points as coordinates. Uses Arbitrary Point Division Algorithm.
3. The same algorithm as above, but with a random insertion. Uses Randomized Arbitrary Point Division Algorithm.
4. A KDtree of 2 dimensions. Uses the KD-trees own division algorithm.

The figure 5.3 shows how on the left, the user has clicked on the button `Start Polygon` to start drawing, and then has entered a set of points to build a polygon, showing on each point the order in which the user clicked for its creation. The points will be joined in the same order in which they were entered, so the user must be careful not to enter a polygon that self-enters.

Finally, the user must click on the `End Polygon` button to finish drawing, to create the polygon, and see how the initial mesh is created from the tree (quadtree or kdtree) generated by the insertion of points. In the case of the figure 5.3, a quadtree with an arbitrary insertion algorithm was used, creating an initial mesh according to the points entered by the user.



(a) Points given by the user.



(b) Creating a polygon and a Initial Mesh with user-entered points

Figure 5.3: Drawing a polygon with the given points.

5.2.3. Quadtree Refining Panel

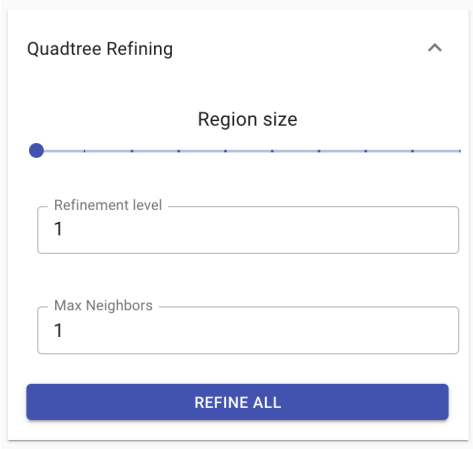
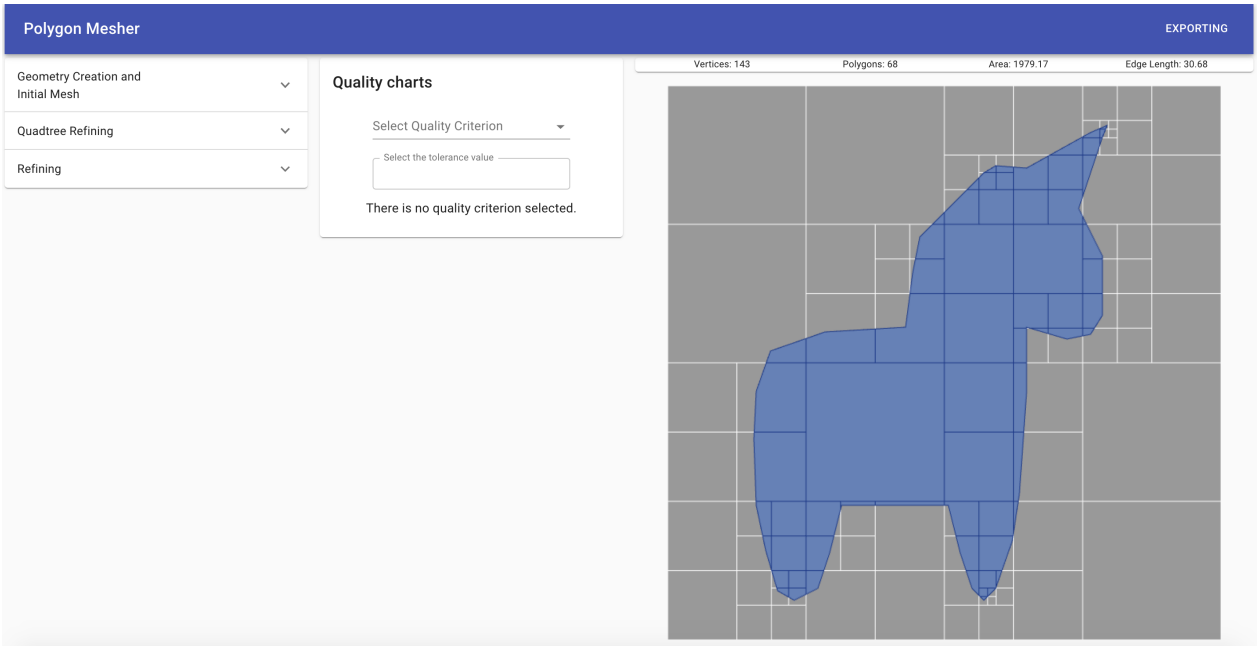


Figure 5.4: Quadtree Refining Panel.

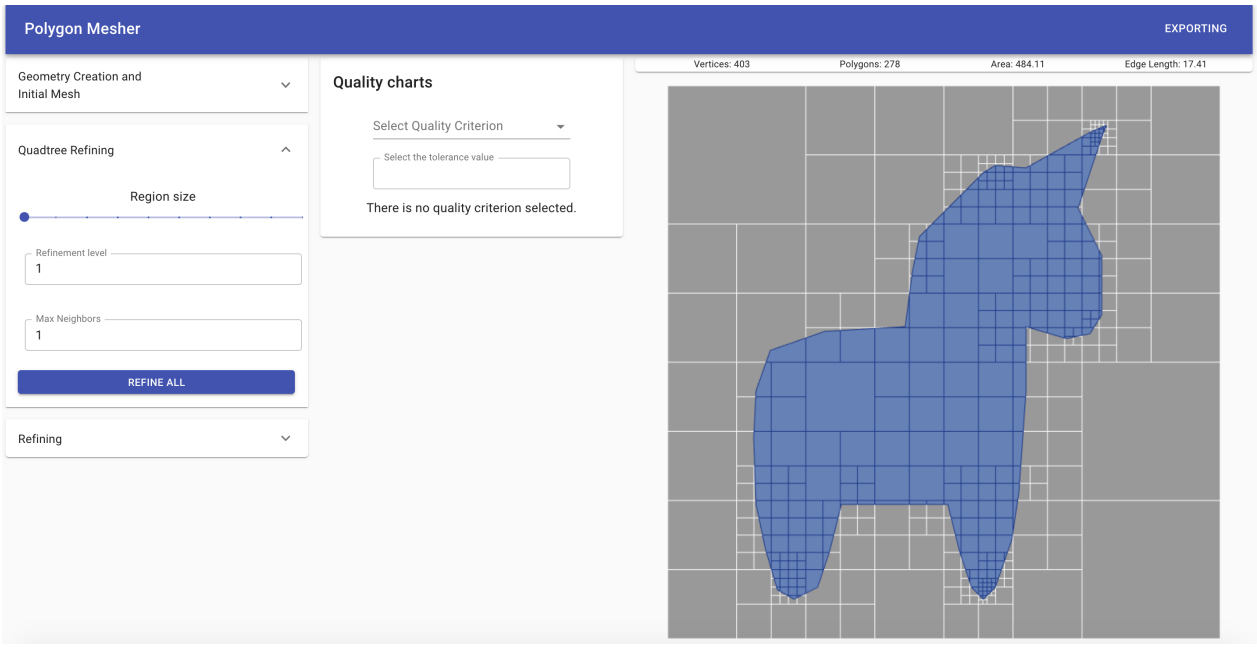
In this panel, the user can perform a deeper refinement in those generated polygons that he needs, by clicking on them, or refining all the polygons generated in the initial mesh. The refinement will be done by means of an additional subdivision in half to the quadrant that generated the polygon, which implies that it is necessary that the polygon has been cut by the application for this refinement to work. In other words, if a user enters a ready-made polygon mesh, this refinement will not be possible.

There are two possible numerical values that the user can enter. *Refinement level* corresponds to the number of times the refinement process is carried out, that is, how many times it is refined on the set of polygons created after each refinement. The other input corresponds to the *maximum refinement factor* that can exist as the difference between a polygon and its neighbors.

Figure 5.5 (a) shows how to visualize an initial mesh of a unicorn created from a preloaded test, using a quadtree with half point division algorithm. To that initial mesh, a refinement is applied using a quadtree with the same algorithm to each one of the polygons, clicking the `Refine All` button. On this example, only one refinement process was used, and with a maximum difference of refinement of one with respect to the neighbors of each polygon. The result of this process is seen in Figure 5.5 (b).



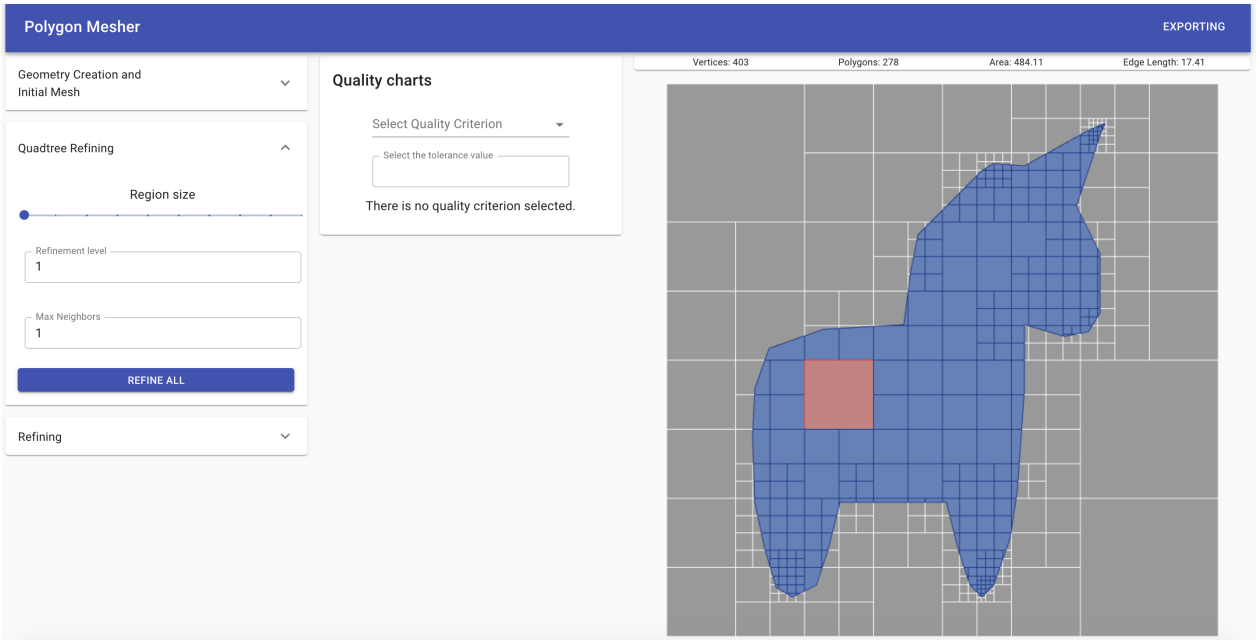
(a) Initial mesh of polygons with unicorn outline.



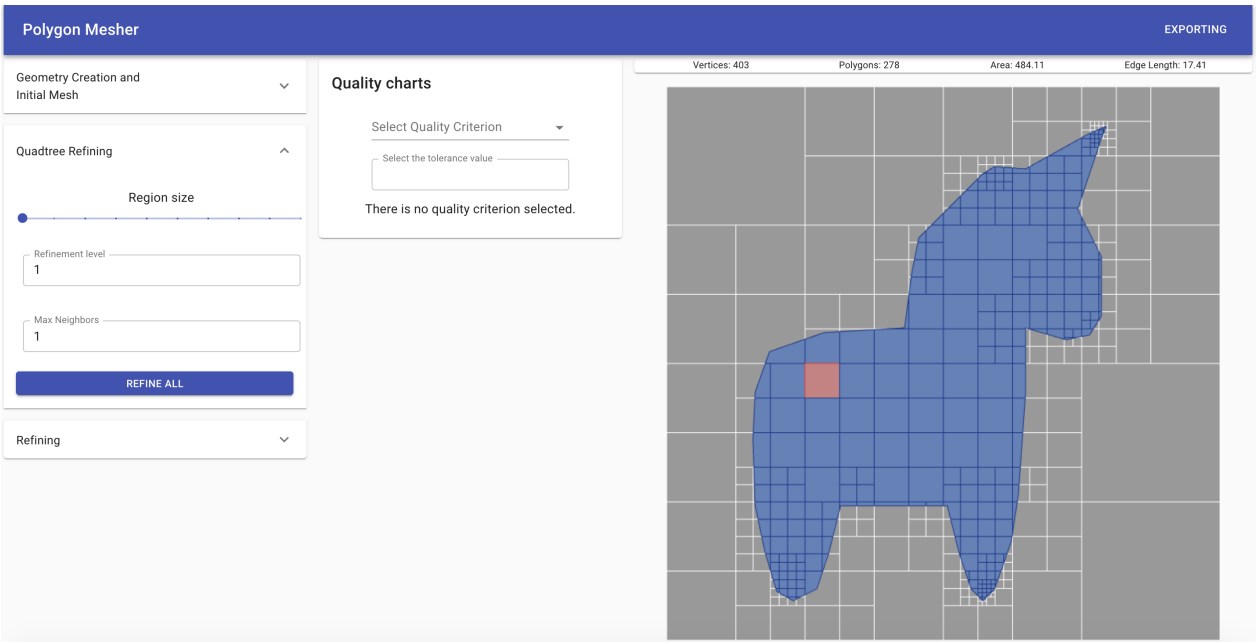
(b) Mesh generated after pressing the `Refine All` button.

Figure 5.5: Refining a polygon mesh with a quadtree.

Figure 5.6 (a) shows how the user is able to place the mouse cursor over the polygons that the user wants to refine, which are marked in red. When the user clicks the said polygon is refined, resulting in four new sub-polygons, as seen in figure 5.6 (b). Note that the parameters of amount of refinements and max neighbor factor did not change.



(a) The user places the cursor on the polygon that he wants to refine.



(b) After clicking on the polygon, it is subdivided into four new polygons.

Figure 5.6: Quadtree refinement for a polygon mesh, and subsequent user refinement.

5.2.4. Refining Panel

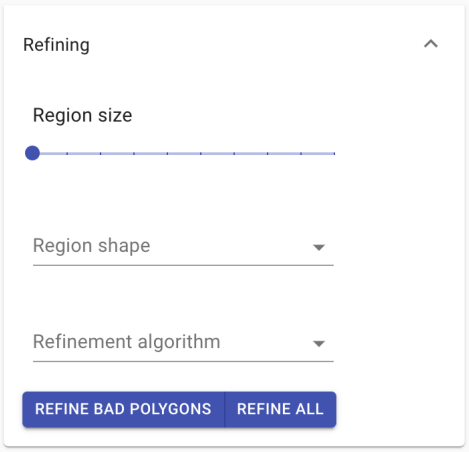


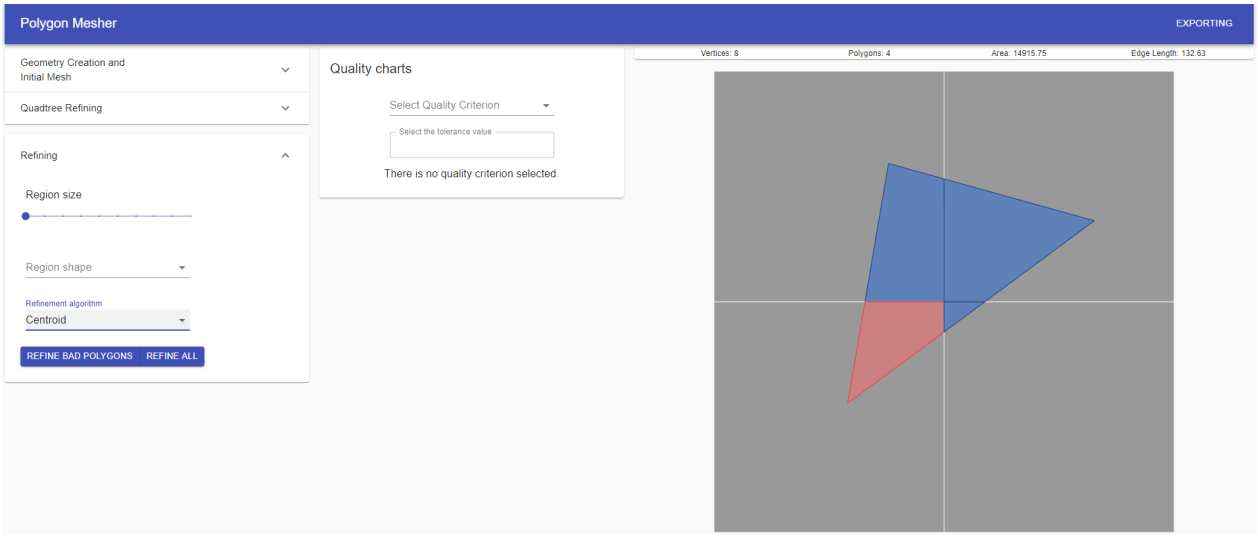
Figure 5.7: Refining Panel.

This panel is in charge of making quality refinements using the different algorithms implemented in this work, which are:

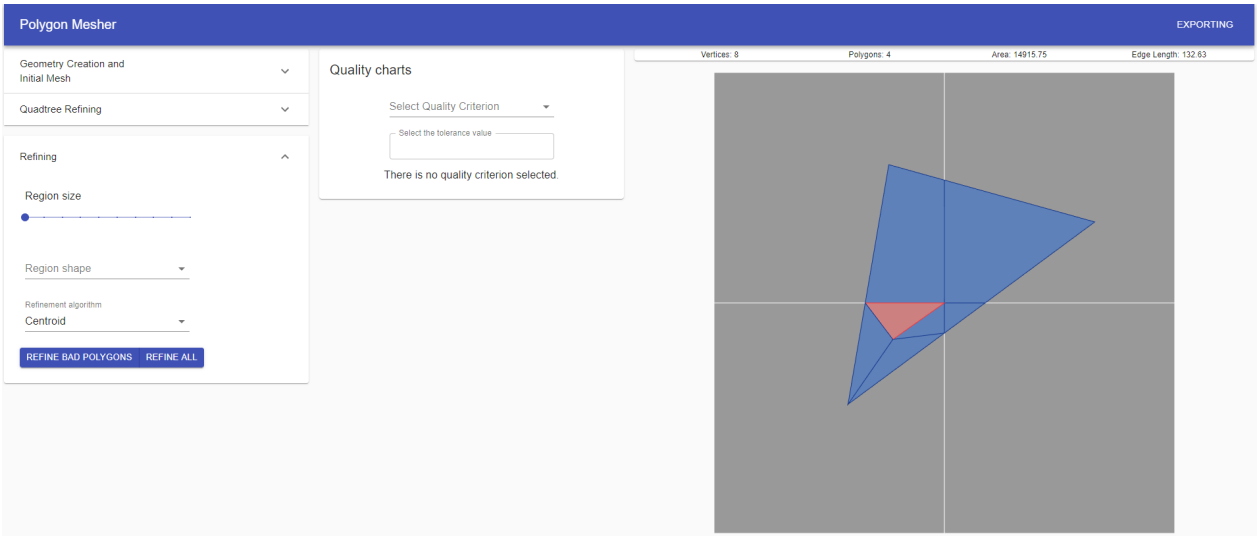
1. Centroid, detailed in Algorithm 44
2. Centroid with replicaction, detailed in Algorithm 45
3. Splitting longest edge, detailed in Algorithm 43

Refinement can be done in three possible ways. The first way is for the user to click on the polygons they want to refine, just as they did with the refinement using quadtree. The other two have to do with the buttons that appear in the figure: `Refine Bad Polygons` takes all those polygons whose *needsRefinement* flag is true according to the selected quality criteria and refines them, while `Refine All Polygons` takes all the polygons independently of whether or not they need refinement, and they are refined.

In the Figure 5.8 we can see how a user selects a polygon from a simple initial mesh of a triangle outline, and after clicking it, the polygon is refined using the centroid algorithm (note that it is the one selected in the input).



(a) The user places the cursor over the polygon that he/she wants to refine.



(b) After clicking on the polygon, it is subdivided according to the centroid algorithm.

Figure 5.8: Refining a polygon mesh with quality algorithms.

5.2.5. Quality Component

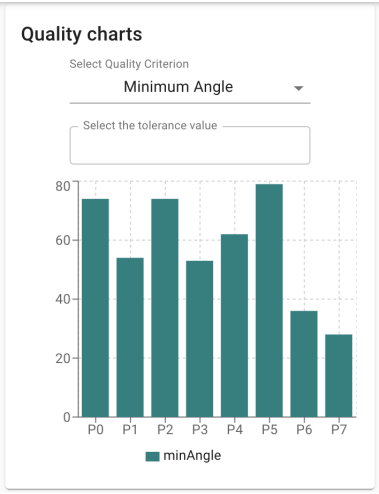


Figure 5.9: Quality Component.

This component is in charge of displaying graphics according to a certain metric, for example, the minimum length, the minimum angle and areas of the polygons of the mesh. For this, there is an input for the user, in which the metric can be chosen, and another input in which the maximum tolerance allowed for said metric can be chosen. With this, all those polygons of the mesh whose metric is above the maximum tolerance value, will be marked with the flag *needsRefinement* as true, and will be shown shown in blue on the canvas. If the polygons are correct, they will be shown in green.

The bar graph that shows the synthesized information of the polygons of the entire mesh, changes automatically as new polygons with refinements are obtained, or if a new metric is chosen.

5.3. Algorithms Implementation

Next we will see the implementation of the different algorithms used by each of the panels, to carry out their tasks. In particular, we will review the polygon cutting algorithms, the generation of a compliant mesh, the insertion of the points in the trees and the different refinement techniques.

5.3.1. Clipping Algorithms

Throughout this thesis, polygon cutting is the central algorithm used to generate smaller polygons. The cut of polygons consists of locating the area of intersection generated after superimposing the polygon that we want to cut, which we call the `subject polygon`, with another polygon, which we call the `cutting polygon`.

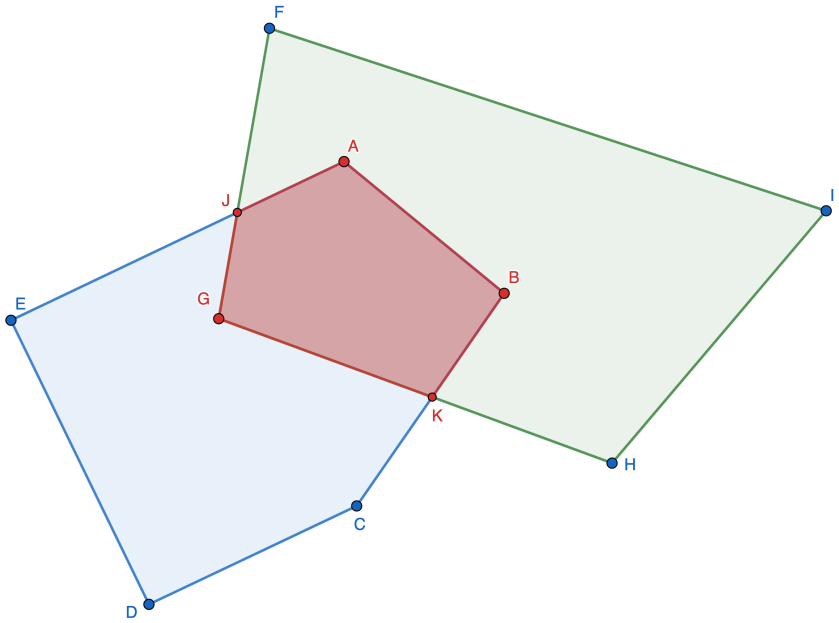


Figure 5.10: Example of cutting polygons. Suppose that the subject polygon is $ABCDE$ and the cutting polygon is $FGHI$. The cut polygon is $ABKGJ$.

For this process, it is necessary to correctly obtain the intersections between the subject polygon and the clipper polygon. In the case of the Figure 5.10, these intersections correspond to points J and K. Later, we generate the cut polygon using the appropriate points. Below, we detail the implementation of the algorithms we use in this research.

5.3.1.1. Sutherland Hodgman Algorithm

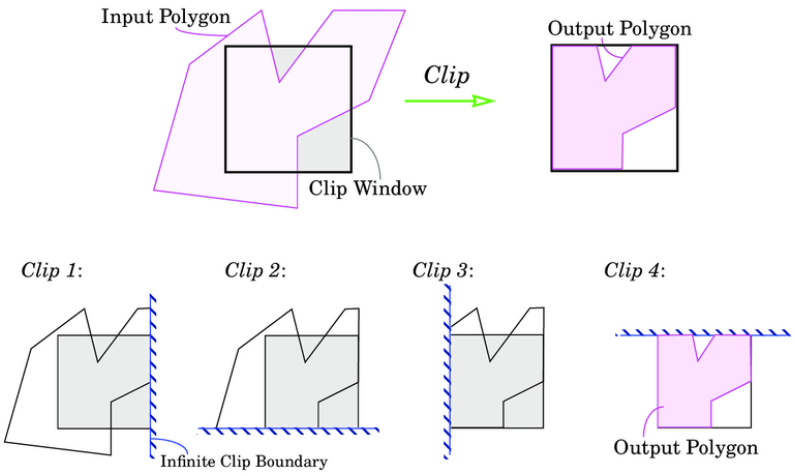


Figure 5.11: Sutherland Hodgman's algorithm example. The input polygon is cut against the clipping window. Image taken from [1]

For Sutherland Hodgman's algorithm, we need to extend in an infinite line each of the segments that conforms the cutting polygon (clipping window in the example), to look for

the intersections with the Subject polygon.

Algorithm 26 Sutherland Hodgman algorithm for clipping polygons

```
1:  Input
2:      subject     List of Points, representing the polygon to be cut
3:      clipper     List of Points, representing the cutter polygon
4:  Output
5:                  List of Points, representing the clipped polygon

6:  function Clip(subject, clipper)
7:      subjectCCW ← subject.orderedPointsToCCW()
8:      clipperCCW ← clipper.orderedPointsToCCW()
9:      clippingEdges ← clipperCCW.getCCWEdges()
10:     output ← subjectCCW
11:     foreach clipEdge in clippingEdges do
12:         input ← output
13:         output ← [ ]
14:         for i ← 0 to input.length-1 do
15:             currentPoint ← input[i]
16:             nextPoint ← input[(i + 1) mod input.length]
```

For the implementation of the Sutherland Hodgman algorithm, we must go through each of the segments that make up the cutting polygon (clipping window), considering as if they were infinite lines. To do this, we first create a reference called `output` on line 10, which will initially be identical to the subject polygon, and that will contain the partial results of the polygon cut, until its final result.

In each iteration, we create a variable called `input`, which will be equal to the previous obtained result (stored in the output variable), and reinitialize output to an empty polygon. Within the innermost cycle, we must ask each point of the current polygon built, and the point that follows, whether they are within or outside the cut region. If we are inside the subject polygon and the next one as well, then both are part of the result, otherwise if the next one is in the cut region, the current point and the intersection are part of the result. In case the current point is in the cut region, we ask if the next one is inside the subject polygon (incoming case), in this case, we add the intersection, and then the next point to the result. After having processed the current point, we continue with the next point, and so on until we have gone through all of them.

Algorithm 26 Sutherland Hodgman algorithm for clipping polygons (cont.)

```
17:             if currentPoint is inside the clipping region then
18:                 if nextPoint is inside the clipping region then
19:                     add nextPoint to output
20:                 else
21:                     currentEdge ← Edge(currentPoint, nextPoint)
22:                     intersection ← INTERSECTION(clipEdge, currentEdge)
23:                     add intersection to output
24:             else
25:                 if nextPoint is inside the clipping region then
26:                     currentEdge ← Edge(currentPoint, nextPoint)
27:                     intersection ← INTERSECTION(clipEdge, currentEdge)
28:                     if intersection is equal to nextPoint then
29:                         add intersection to output
30:                     else
31:                         add intersection to output
32:                         add nextPoint to output
33:             ensure output is Counterclockwise
34:         end
35:     end
36:     return output
37: end
```

5.3.1.2. Extended Greiner Hormann Algorithm

The implementation of Greiner Hormann's extended algorithm follows the procedures explained in the work of Foster et al. [21], adapting the instructions to our work as appropriate. The algorithm can be described in a series of steps:

1. Calculate all the intersections between both polygons
2. Classify intersections
3. Mark the chains of degenerate intersections
4. Check for crossing intersections
   - 4.1. If they exist, then we create the entry and exit lists
   - 4.2. We go through the lists, to form the cut polygons
5. If it does not exist, we check if the polygon is completely inside or outside the clipper polygon. If this is not the case, then the polygons do not intersect.

Due to the complexity of the algorithm and its stages, we decided to make a template of each operation, extracting the logic to different functions. Therefore, the `clip()` function receives the polygon to cut (Subject Polygon) and the cutter polygon (Clipper Polygon), and through different auxiliary function calls, it firstly processes the intersections between them.

The calculation of the intersections is updated in each phase of the algorithm, either by adding information flags, or by adding new points. Updates to the intersection list serve as input for each new step in the algorithm.

Algorithm 27 Extended Greiner Hormann algorithm for clipping polygons

```
1: Input
2:     subject     List of Points, representing the polygon to be cut
3:     clipper     List of Points, representing the cutter polygon
4: Output
5:                 List of Points, representing the clipped polygon

6: function Clip(subject, clipper)
7:     subjectCCW ← subject.orderedPointsToCCW()
8:     clipperCCW ← clipper.orderedPointsToCCW()
9:     subjectCircular ← new Circular(subjectCCW)
10:    clipperCircular ← new Circular(clipperCCW)
11:    intersections ← CalculateIntersections(subjectCircular, clipperCircular)
12:    intersections ← ClassifyIntersections(intersections)
13:    intersections ← markIntersectionChains(intersections)
14:    existsCrossing ← areCrossingIntersections(intersections)
```

Finally, as mentioned in the algorithm stages, we analyze if there are crossing intersections. If they exist, then a reconstruction of each of the sub-polygons is performed by going through a list of enter or exit intersections. If it does not exist, we carry out an inspection for edge cases.

Algorithm 27 Extended Greiner Hormann algorithm for clipping polygons (cont.)

```
15:     if existsCrossing is True then
16:         (entering, exiting) ← BuildEnterExitList(clipperCircular,
                                  intersections)
17:         polygons ← Traverse(subjectCircular, clipperCircular,
                        entering, exiting)
18:         return polygons
19:     else
20:         polygons ← new Circular()
21:         subjectPoly ← new Polygon(subjectCCW)
22:         clipperPoly ← new Polygon(clipperCCW)
23:         noInterSubject ← findNonIntersectionPoint(subjectCircular,
                             clipperCircular)
24:         noInterClipper ← findNonIntersectionPoint(clipperCircular,
                             subjectCircular)
25:         if noInterSubject exists and is inside clipperPoly then
26:             add subjectPoly to polygons          /*subjectPoly completely inside clipperPoly*/
27:         else if noInterClipper exists and is inside subjectPoly then
28:             add clipperPoly to polygons          /*clipperPoly completely inside subjectPoly*/
29:         else if subjectPoly is equal to clipperPoly then
30:             add subjectPoly or clipperPoly to polygons          /*Both polygons are equal*/
31:         else
32:             return polygons                      /*Then, there's no intersection between polygons*/
33:         return polygons
34: end
```

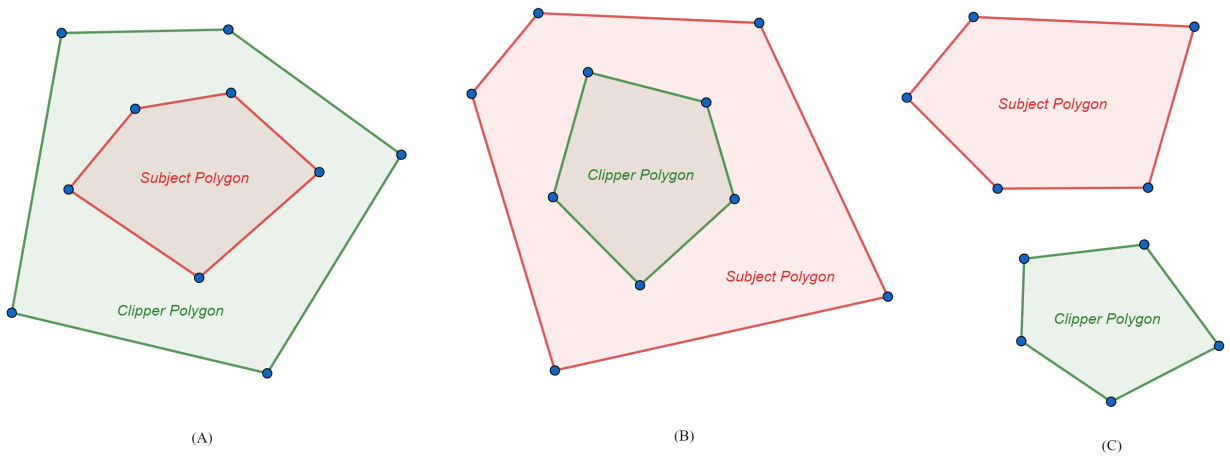


Figure 5.12: The three types of edge cases for the algorithm. (A) The Subject Polygon is completely contained within the cutter polygon. (B) Analogous case, when the cutter polygon is completely contained within the Subject Polygon. (C) There is no intersection between both polygons.

We note that the non-existence of crossing points does not necessarily imply that there

is no possible result. According to the Figure 5.12, we see that the case `(A)` and `(B)`, even when there are no intersections between their segments, the intersection between both polygons is not empty. In the case of `(A)`, the result of the algorithm must give as a result the Subject Polygon, while in `(B)`, the Clipper Polygon. Only in the case of `(C)` where there is no intersection of any kind, the polygon cut operation is empty.

The treatment of the said edge cases begins on line 19 of the algorithm. For the `FindNonIntersectionPoint(P, Q)` method, its implementation consists of finding a segment of the polygon P that is not present in the Q. We then proceed to find the midpoint of that segment, asking if the point exists and if it is inside the region bounded by Q: if that is the case then P is completely contained within Q. That procedure can be done by assigning P and Q as Subject Polygon and Clipper Polygon appropriately. On the other hand, another trivial case is when both polygons are equal, in that case it is enough to return any of them. Finally, if no previous case has occurred, then the intersection is null.

5.3.1.3. Calculating intersections

Regarding the calculation of intersections between two polygons P and Q, we performed the usual procedure of comparing each segment of P, with all the segments of Q, in search of intersections. However, we must not only find these points, but also insert them in the appropriate places in the list of points of the polygons P and Q, so that they are well formed and ordered in counterclock wise.

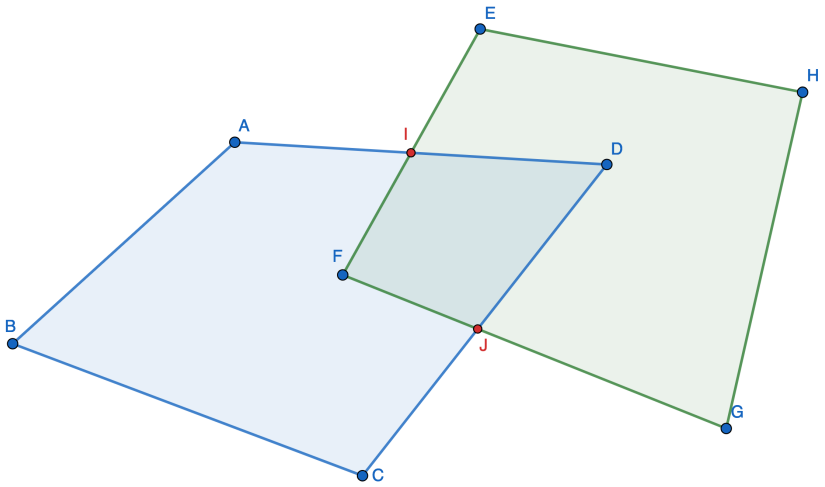


Figure 5.13: Calculation of the intersections between two polygons.

Consider the $ABCD$ and $EFGH$ polygons. The intersections that must be found between both polygons are the red points I and J, and must be inserted in the corresponding places. Finally the resulting polygons are $ABCJDI$ and $EIFJGH$.

Algorithm 28 Calculating intersections between two polygons

```
1: Input
2:     subject     Circular List of Points, representing the polygon to be cut
3:     clipper     Circular List of Points, representing the cutter polygon
4: Output
5:                 List of Points, representing the intersections

6: function CalculateIntersections(subject, clipper)
7:     currentSubject ← subject.head
8:     intersections ← [ ]
9:     do
10:        currentClip ← clipper.head
11:        do
12:            subjectEdge ← new Edge(currentSubject, currentSubject.next)
13:            clipperEdge ← new Edge(currentClip, currentClip.next)
14:            interInfo ← ExtendedEdgeIntersection(subjectEdge, clipperEdge)
15:            if interInfo exists then
16:                insert interInfo in subject list after currentSubject
17:                insert interInfo.intersection in clipper list after currentClip
18:            currentClip ← currentClip.next
19:        while currentClip is different from clipper.head
20:        currentSubject ← currentSubject.next
21:    while currentSubject is different from subject.head
```

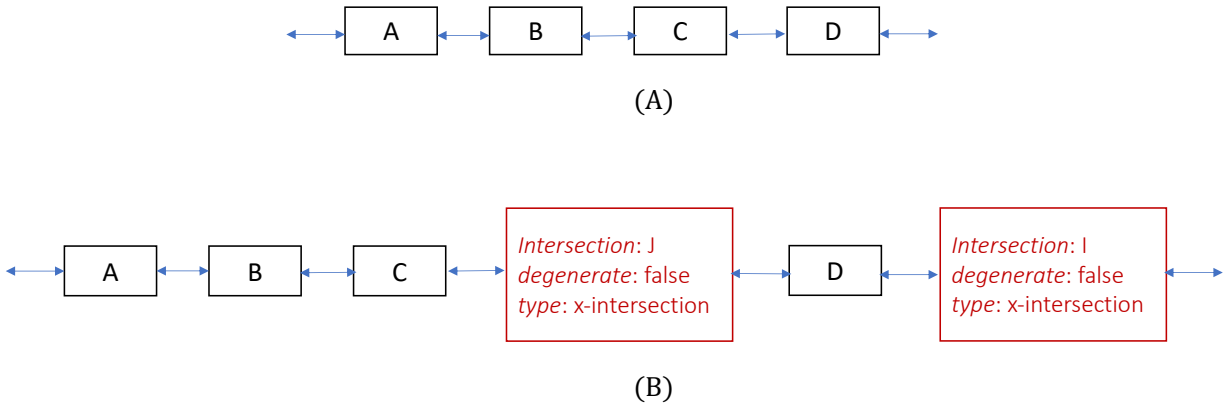


Figure 5.14: Representation of polygons.(A) Before inserting the intersections.(B) After inserting the intersection info.

The insertion of the intersections in the list of points of the polygon is done by adding nodes with additional information to the doubly linked circular list. The information consists of whether it is a degenerate intersection or not, and the type of intersection that is according to the classification of the paper [21].

Algorithm 28 Calculating intersections between two polygons (cont.)

```
22:     foreach element in subject do                    /*element can be a Point or interInfo*/
23:         if element is an inserted intersection then
24:             add element to intersections if they do not exist
25:             element ← extract intersection point from interInfo of element
26:     end
27:     foreach intersectionInfo in intersections do
28:         (prevSubj, nextSubj) ← find prev and next of the intersection in subject
29:         add (prevSubj, nextSubj) to intersectionInfo
30:         (prevClip, nextClip) ← find prev and next of the intersection in clipper
31:         add (prevClip, nextClip) to intersectionInfo
32:     end
33:     return intersections
34: end
```

Finally, we run a linear tour of the Subject Polygon's list of points on line 22 of the Algorithm 28, to insert the information nodes into a list of intersections only. Subsequently, we mutated the list so that the polygon only contains points and avoid confusion later. In line 27 of the Algorithm 28, we update the node links so that the circular list is consistent with the new inserted points. This Algorithm ends by returning the list of intersection information nodes.

5.3.1.4. Determining the orientation of polygon chains

To classify intersections, we first create an auxiliary function that allows us to obtain the orientation (left or right) of a point Q with respect to two consecutive segments of a polygon. Following the nomenclature used in [21], we define these segments as a `polygonal chain`.

Definition 9 (Polygonal Chain)**.** *A polygon chain is a sequence of two consecutive segments of a polygon.*

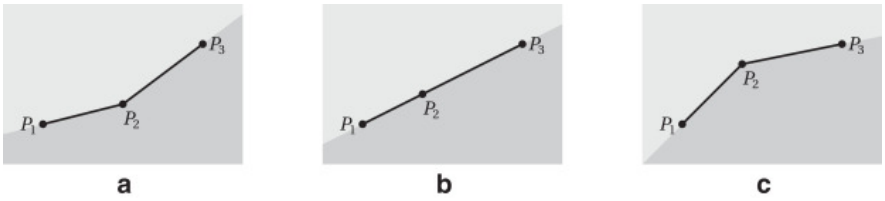


Figure 5.15: Three possible cases of orientation for a point, with respect to a polygonal chain. The light gray region is considered to be to the left of the polygon chain, while the dark gray region is considered to be to the right. Image taken from Article [1].

The main idea for the development of this function is to study the cross product considering the point Q that we want to study as the origin, and both segments in question. A third cross product is calculated considering the first point of the polygonal chain as the origin, obtaining the product with respect to the segments of the chain. With these results, calling

them S_1, S_2 and S_3, we can establish the orientation of Q with respect to the polynomial chain.

Algorithm 29 Getting which side of two consecutive edges, P_1P_2 and P_2P_3, is a point Q

```
1: Input
2:     Q          Point to be checked
3:     P1         Start Point of the first Edge
4:     P2         End Point of the first Edge, and start of the second Edge.
5:     P3         End Point of the second Edge.
6: Output
7:                'LEFT' or 'RIGHT', depending on which side is Q

8: function PointChainOrientation(Q, P1, P2, P3)
9:     S1 ← CrossProduct(Q, P1, P2)
10:    S2 ← CrossProduct(Q, P2, P3)
11:    S3 ← CrossProduct(P1, P2, P3)
12:    if S3 < 0 then
13:        if S1 < 0 and S2 < 0 then
14:            return 'LEFT'
15:        else if S1 > 0 or S2 > 0 then
16:            return 'RIGHT'
17:    else if |S3| < ε then
18:        if S1 < 0 then
19:            return 'LEFT'
20:        else if S1 > 0 then
21:            return 'RIGHT'
22:    else                                          /*Case S3 > 0*/
23:        if S1 < 0 or S2 < 0 then
24:            return 'LEFT'
25:        else if S1 > 0 and S2 > 0 then
26:            return 'RIGHT'
27: end
```

According to the Algorithm 29, we define S_3 as the cross product of the polygonal chain $P_1P_2P_3$, the result of which serves to determine if in P_2, the chain makes a left turn, continues straight, or makes a right turn.

If S_3 indicates that it is a turn to the left (line 12), we ask in relation to Q what the cross products S_1 and S_2 are like. If both are to the left, then Q is in the left region of the $P_1P_2P_3$ chain, otherwise, if either is to the right of $P_1P_2P_3$, then Q is to the right. If S_3 is close enough to 0 (line 17), we consider that $P_1P_2P_3$ are collinear, so it is enough to ask for a single cross product to see if Q is on the left or on the right. Finally, if S_3 indicates a turn to the right (line 22), it is enough to see if any of the products crosses S_1 or S_2 is to the left. If so, the point is on the left, otherwise, if both cross products go to the right, then Q is necessarily to the right of the polygon chain.

5.3.1.5. Classifying intersections

According to the previous procedures, we currently have a list of intersections where each one has three different data:

1. The `Point` object that internally has the X and Y coordinates of the intersection.
2. A `boolean` flag that tells us if the intersection is degenerate or not.
3. A string that tells us the type of intersection it is, according to the classification of the paper.

Our job now to continue the algorithm is to be able to discern whether an intersection is entering to the cut region, or exiting. If all the intersections were non-degenerate (of the x-crossing type), then an intersection that is cataloged as entering from polygon P to polygon Q, necessarily implies that the next intersection will be exiting. However, with degenerate intersections the case does not occur, because there are intersection points of P that are contained in the edge of Q or equal to some vertex of Q, so it is ambiguous to say whether they are entering or exiting intersections.

Remember that the intersections, according to the type of polygonal chain, can be:

1. Crossing
2. Bouncing
3. Left/On
4. Right/On
5. On/On
6. On/Left
7. On/Right

Therefore, the implementation goes through a series of questions about all possible cases involving intersections, both when the segments overlap, and when the intersection is over a segment of the polygon, following the procedures explained in the section. The classification will be saved as a String added to each node of the intersection list, modifying it and returning it with the changes.

Algorithm 30 Classifying Intersections

```
1: Input
2:     L_inter        List of interInfo structures
3: Output
4:                    L_inter updated with intersection types flags

5: function ClassifyIntersections(L_inter)
6:     foreach interInfo in L_inter do
7:         I ← extract intersection Point from interInfo
8:         P_1 ← extract prevSubj from interInfo
9:         P_2 ← extract nextSubj from interInfo
10:        Q_1 ← extract prevClip from interInfo
11:        Q_2 ← extract nextClip from interInfo
12:        if (P_2 = Q_2 and PointChainOrientation(Q_1, P_1, I, P_2) is 'RIGHT') or
               (P_2 = Q_1 and PointChainOrientation(Q_2, P_1, I, P_2) is 'RIGHT') then
13:            add flag 'left-on' to interInfo
14:        else if (P_2 = Q_2 and PointChainOrientation(Q_1, P_1, I, P_2) is 'LEFT') or
                    (P_2 = Q_1 and PointChainOrientation(Q_2, P_1, I, P_2) is 'LEFT') then
15:            add flag 'right-on' to interInfo
16:        else if (P_2 = Q_2 and P_1 = Q_1) or (P_2 = Q_1 and P_1 = Q_2) then
17:            add flag 'on-on' to interInfo
18:        else if P_1 = Q_1 and PointChainOrientation(Q_2, P_1, I, P_2) is 'RIGHT') or
                    (P_1 = Q_2 and PointChainOrientation(Q_1, P_1, I, P_2) is 'RIGHT') then
19:            add flag 'on-left' to interInfo
20:        else if P_1 = Q_1 and PointChainOrientation(Q_2, P_1, I, P_2) is 'LEFT') or
                    (P_1 = Q_2 and PointChainOrientation(Q_1, P_1, I, P_2) is 'LEFT') then
21:            add flag 'on-right' to interInfo
22:        else                                          /* Then, there's no overlapping */
23:            firstSide ← PointChainOrientation(Q_1, P_1, I, P_2)
24:            secondSide ← PointChainOrientation(Q_2, P_1, I, P_2)
25:            if firstSide is not equal to secondSide then
26:                add flag 'crossing' to interInfo
27:            else
28:                add flag 'bouncing' to interInfo
29:    end
30:    return L_inter
31: end
```

5.3.1.6. Marking Intersections Chains

The intersection chains correspond to those edges that overlap between the subject polygon and the cut polygon. To identify them, we know that according to the construction and classification of our intersections, the chains must start with an intersection of the type 'left-on' or 'right-on', subsequently a variable number of intersections 'on-on', and finally an intersection of type 'on-left' or 'on-right'.

Note that if the start is 'left-on' and the end is 'on-left', there was no crossing from the

subject polygon to the interior or exterior of the cut polygon, analogous situation for `'right-on'` and `'on-right'`. Therefore, all the intersections in the chain can be classified as `bouncing`, however if there is a change of direction, for example going from `'left-on'` to `'on-right'`, the last intersection should be considered of type `crossing`.

Algorithm 31 Marking Intersections Chains

```
1: Input
2:     L_inter        List of interInfo structures
3: Output
4:                    L_inter updated with intersection chains flags

5: function MarkIntersectionChains(L_inter)
6:     starters ← [ ]                                  /*Identifying start Points of every chain*/
7:     foreach interInfo in L_inter do
8:         if flag of interInfo is 'right-on' or 'left-on' then
9:             add interInfo into starters
10:    end
11:    interCircular ← new Circular(L_inter)
12:    while starters have Points inside do
13:        startingPoint ← pop element from starters
14:        finalPoint ← follow next nodes on interCircular from startingPoint, until
                        intersection with flag 'on-left' or 'on-right' is found, marking
                        intersections flags as 'bouncing' in between.
15:        if (startingPoint is 'left-on' and finalPoint is 'on-left') or (startingPoint then
               is 'right-on' and finalPoint is 'on-right')
16:            change finalPoint flag to 'bouncing'
17:        else if (startingPoint is 'left-on' and finalPoint is 'on-right') or then
                    (startingPoint is 'right-on' and finalPoint is 'on-left')
18:            change finalPoint flag to 'crossing'
19:        change startingPoint flag to 'bouncing'
20:    end
21:    return L_inter
22: end
```

To make the implementation easier, the first thing we did was obtain each of the beginnings of these chains, that is, add to an array the intersections that are `'right-on'` or `'left-on'`. Later, as long as there are intersections in the said array as it appears in line 12, we remove the first one that is in the stack, and we follow the chain until the appearance of an intersection of end of the chains (`'on-right'` or `'on-left'`). From line 15 to line 19, we mark the intersections according to the values of the beginning and the end of the chain, being `bouncing` if there was no change of direction, and `crossing` otherwise. Finally we return on line 21 the appropriately marked intersections.

5.3.1.7. Building the *Entering* and *Exiting* lists

The next step is the construction of the *Entering* and *Exiting* lists with the corresponding intersections, from the point of view of the subject polygon. For simplicity, both lists are be

doubly linked circular. The algorithm in summary is simple and follows the logic that every time we find a crossing intersection, it is because we change state, either we go from being inside to being outside the cutting polygon or vice versa.

Algorithm 32 Building *Entering* and *Exiting* Lists

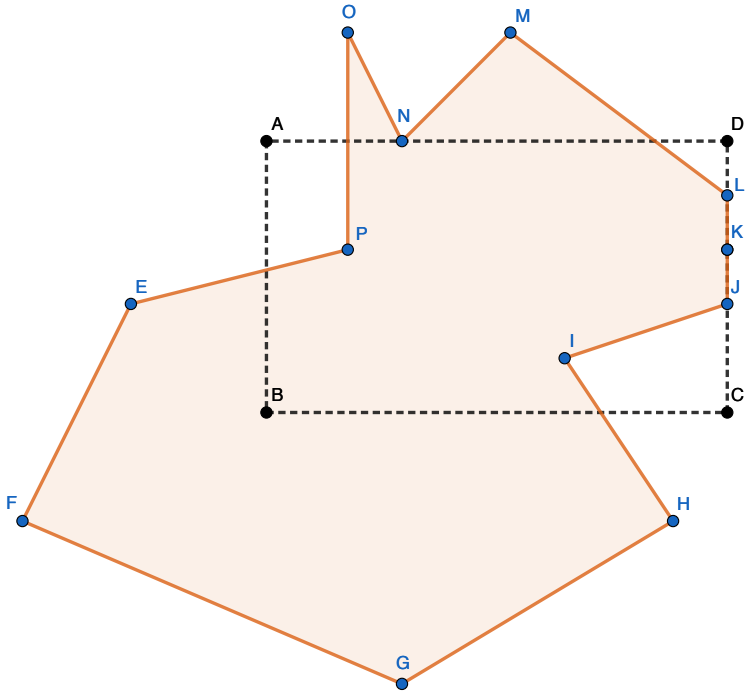
```
1:  Input
2:      clipper      Circular List of Points representing the clipper polygon
3:      L_inter      List of interInfo structures
4:  Output
5:                   L_inter updated with intersection chains flags

6:  function BuildEnterExitList(clipper, L_inter)
7:      firstCrossing ← True
8:      nextLabel ← ”
9:      entering ← new Circular()
10:     exiting ← new Circular()
11:     foreach interInfo in L_inter do
12:         if flag of interInfo is "crossing" then
13:             if firstCrossing is True then
14:                 nextPoint ← get nextSubj from interInfo        /*If next point is inside*/
                                                                   clipper polygon, then
                                                                   we are entering
15:                 inside ← Inside(nextPoint, Polygon(clipper))
16:                 borderCrossing ← Verify if nextPoint is in the list clipper
17:                 if inside is True or borderCrossing is True then
18:                     add intersection point from interInfo to entering
19:                     nextLabel ← "exiting"
20:                 else                                           /*Otherwise, we are getting out of the*/
                                                                   clipper polygon
21:                     add intersection point from interInfo to exiting
22:                     nextLabel ← "entering"
23:                 firstCrossing ← False
24:             else
25:                 if nextLabel is "entering" then
26:                     add intersection point from interInfo to entering
27:                     nextLabel ← "exiting"
28:                 else
29:                     add intersection point from interInfo to exiting
30:                     nextLabel ← "entering"
31:     end
32:     return (entering, exiting)
33: end
```

For the implementation in lines 9 and 10 we create the empty circular lists. Later on line 11 we go through each of the intersections that we have (with their respective additional information that we have been adding). From line 13 to line 23, special treatment is made for the first crossing intersection that the algorithm finds, since depending on which region

the next point of that intersection on the subject polygon is located, it will be the list that will be entered: If the next point is inside the cut polygon, it means that the subject polygon is entering the cut section, so we added this intersection to the *Entering* list. Otherwise, the polygon leaves the cut section, so it is added to the *Exiting* list.

Subsequently, depending on the value of the first intersection, the following crossing intersections will alternate the *Entering* and *Exiting* values as shown from line 25 to line 30, because the other intersections are of the bouncing type necessarily due to construction and chain management of intersections previously made. Finally, on line 32, both lists are returned.



Subject Polygon
E
F
G
H
I
J
K
L
M
N
O
P

Clipping Polygon
A
B
C
D

Figure 5.16: Two polygons example.

Let us consider the figure 5.16, to illustrate our algorithm so far. It shows two polygons: one orange that will be our subject polygon and the other with a crossed line that will be our cut polygon, represented by the ABCD points. This is the initial situation, where the intersections have not been calculated or inserted in each polygon, nor have they been classified.

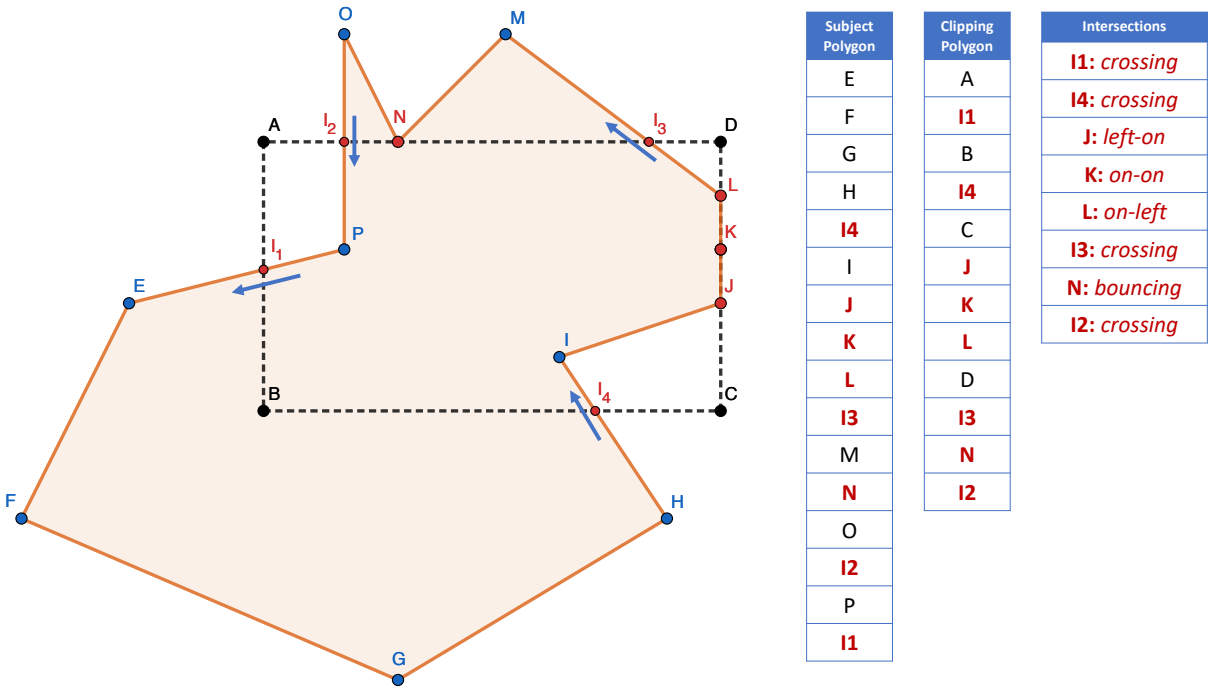


Figure 5.17: Classifying intersections and building lists.

In the figure 5.17, we can see that the intersections have been calculated and inserted in the corresponding places in the polygon lists. In the lists, intersections I_1, I_2, I_3 and I_4 are crossing, while the rest are degenerate intersections. Note that JKL intersections form a chain of type `left-on`, `on-on`, and `on-left`, and since the starting direction matches the ending (left), the intersections are bouncing and do not involve a crossing type.

Regarding the creation of the *Entering* and *Exiting* lists, we have the following. The first intersection in the list is I_4, which is of type crossing. We examine the point before this intersection, in our case it is H, and since it is outside the cut polygon, the intersection I_4 must necessarily go in the *entering* list. Therefore, the algorithm alternates between *entering* and *exiting* as appropriate, obtaining as results for crossing intersections: I_3 is *exiting*, I_2 is *entering* and I_1 is *exiting*. The said behavior is seen visually in the figure 5.17 by arrows that enter the cutting polygon and others that exit.

5.3.1.8. Traversing the lists

Finally, after building the two lists *Entering* and *Exiting*, we can go through both the subject polygon and the cut polygon, in order to create the cut polygons. The algorithm consists of going through each one of the intersections that are in the Entering list, locating its place in the list of points of the subject polygon, and going forward until we find an Exiting intersection. At that moment, we must change the list to the list of points of the cut polygon, and continue going through its list. This process of jumping between lists is performed until the algorithm returns to the intersection with which it started.

The implementation of this part was done through two functions. The `Traverse` function goes through each of the intersections of the *Entering* list, and invokes the `TraverseList`

function with the list of points of the corresponding polygon (subject or clipper).

Algorithm 33 Traversing *Entering* and *Exiting* lists to obtain the cut polygons

```
1:  Input
2:      subject     Circular List of Points representing the subject polygon
3:      clipper     Circular List of Points representing the clipper polygon
4:      entering    Circular List of Points representing entering intersections
5:      exiting     Circular List of Points representing exiting intersections
6:  Output
7:                  Circular list with the new cut polygons

8:  function TRAVERSE(subject, clipper, entering, exiting)
9:      polygons ← new Circular()                          /*Contains the new cut polygons*/
10:     L_current ← subject
11:     while entering has points do
12:         polygon ← new Circular()
13:         start ← get intersection point from entering.head
14:         transitionNode ← entering.head
15:         count ← 0
16:         while transitionNode is not Null and (count = 0 or (count > 0 and start is not
        equal to transitionNode value)) do
17:             (transitionNode, poly) ← TRAVERSELIST(L_current, entering, exiting, polygon,
        transitionNode, start)
18:             if L_current is equal to subject list then
19:                 L_current ← clipper
20:             else
21:                 L_current ← subject
22:             count ← count + 1
23:             polygon ← poly
24:         end
25:         cutPolygon ← new Polygon(polygon)
26:         order cutPolygon to CounterClockwise
27:         add cutPolygon to polygons
28:     end
29:     return polygons
30: end
```

Showing the implementation in more detail, on line 10 we save a variable with the current list that initially corresponds to that of the subject polygon. On line 11 we start a cycle for each of the *entering* intersections that exist in the list of the same name. Within the cycle, we make a new cycle on line 16 for the construction of the polygon, whose term condition is that the transition node is null (that is, that we have reached the same node with which we started). Then we call the `TraverseList` function on line 17, obtaining the current polygon and the transition node from which the jump was made in the current list. After that, we must alternate the current list to the list of points of the cut polygon or to the subject polygon as appropriate.

Finally, at the end of the innermost cycle, a cut polygon has been obtained, which is saved in the polygons list. At the end of the outermost cycle, corresponding to the entering intersections, we will have all the possible polygons cut in the polygons list, so we return their value on line 29.

Algorithm 34 Helper function to traverse just one list at time

```
1:  Input
2:      L_current   Circular List of Points. Can be subject or clipper polygon
3:      entering    Circular List of Points representing entering intersections
4:      exiting     Circular List of Points representing exiting intersections
5:      polygon     Circular List that will contain the new polygon
6:      tNode       Circular node from where the traverse started
7:      start       Start Point from where the traverse started
8:  Output
9:                  Next transition node for moving to the other list

10: function TraverseList(L_current, entering, exiting, polygon, tNode, start)
11:     Ensure value of tNode is inside L_current
12:     if value of tNode is inside entering list then
13:         remove the value of tNode from entering, mutating it
14:     listNode ← get Node from L_current that has the same value as tNode
15:     do
16:         add Point value from listNode to polygon
17:         listNode ← listNode.next
18:         if Value of listNode is equal to the start point then
19:             return null
20:     while listNode is not null and entering and exiting don't
                include the value of listNode
21:     return (listNode, polygon)
22: end
```

The function that goes through a particular list, first makes sure that the transition node exists in the current list, which should always happen since by construction, if they are intersections, they must belong to both lists (subject polygon list and clipper polygon list). Then on line 12 we remove the intersection from the *entering* list to reach the cycle condition. After line 15 to line 20, we go through the current list adding all the points to the *polygon* creation list, bearing in mind that if we reach the initial node, we return null as a value. Otherwise, we must return the polygon that we have created so far, and the transition node on line 21.

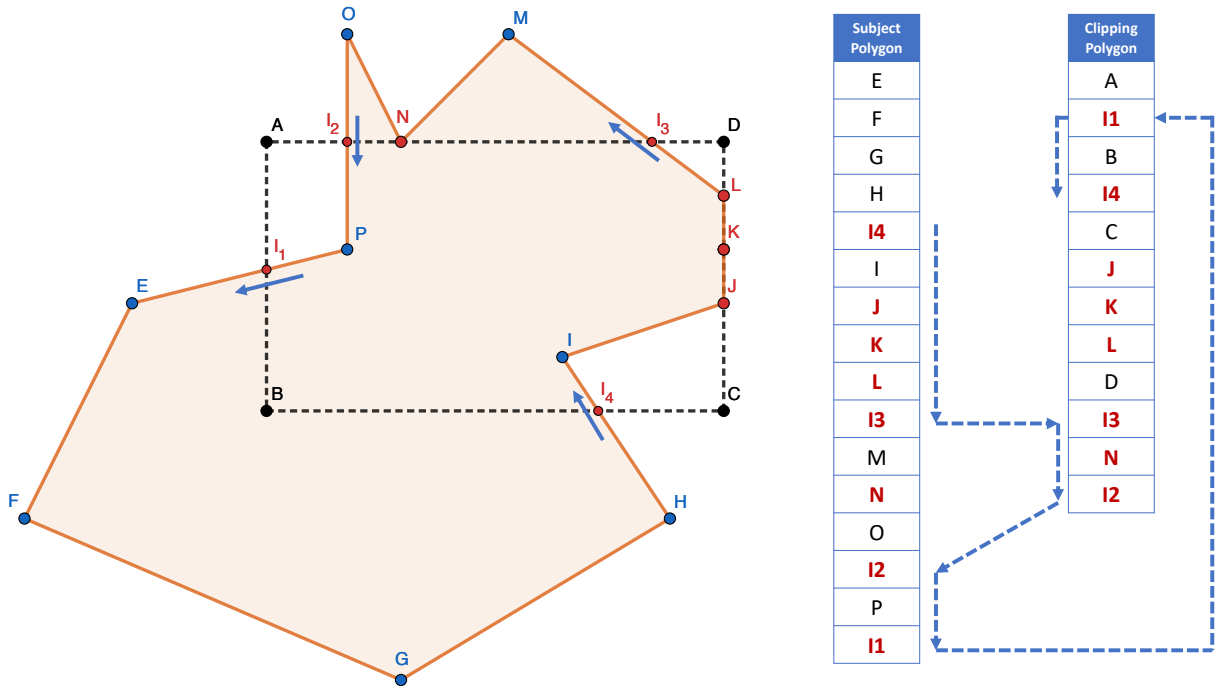


Figure 5.18: Example of constructing the cut polygons from both lists.

To conclude with the example presented, we see how in the figure 5.18, the polygon lists are traversed starting from the first intersection of entering, which is I_4. We go through the subject polygon list until we reach the next crossing intersection, which is I_3. In this case, we must change the list and go through the clipping polygon list. We keep going through each of the lists, jumping between the crossing intersections, until we reach the initial intersection I_4. Note that when visiting the intersection I_2, which is in *Entering* list, it is deleted from the list in line I_2 and I_3, so the algorithm ends since the *Entering* list is empty. At that moment, we gather all the points we visit, making up the only polygon cut as the result of the cut algorithm.

5.3.1.9. Complexity Analysis

To analyze the complexity of the clipping algorithm, we analyzed each stage separately according to our implementation, and then saw if it corresponded to the theoretical results. Each of the stages has the following analysis. Suppose we have a subject polygon of n_s points and a clipper polygon of n_c points. Then:

1. **Intersections**: we perform a comparison for each edge of subject with the n_c arcs of clipper. This process is done n_s times, so this operation is $\mathcal{O}(n_c \cdot n_s)$. Let n_i be the number of intersections found, then we perform a selection sweep of intersections inserted in the copy list of subject which takes $\mathcal{O}(n_s + n_i)$, and then we obtain the neighborhood of intersections in subject and clipper, a process that takes $\mathcal{O}(n_c) + \mathcal{O}(n_s)$ time. Therefore the dominance is $\mathcal{O}(n_c \cdot n_s)$.

2. **Classify intersections**: For each of the intersections found, we perform a classification according to the calculations made with the cross products. As we have already pre-

calculated the points, obtaining the previous and next point is $\mathcal{O}(1)$, so the whole process runs in $\mathcal{O}(n_i)$.

3. **Mark Intersections**: To obtain those intersections that initiate a degenerate chain on a polygon, just go through the list of intersections, which takes $\mathcal{O}(n_i)$. Then we perform a second iteration, considering as start the intersections obtained previously to categorize them, a process that again takes $\mathcal{O}(n_i)$.

4. **Entering and Exiting**: Forming the list of intersections that are classified as entering or exiting, corresponds to a sweep through all intersections, which takes $\mathcal{O}(n_i)$ time.

5. **Traverse**: For each of the intersections marked as entering, we start traversing the clipper and subject polygons, exchanging the lists. This process takes at most $\mathcal{O}(n_s + n_c)$.

The analysis of our implementation corresponds to the theoretical result presented in [23], which shows that polygon cutting grows in the order of $\mathcal{O}(nm)$, where n and m are the number of edges that polygons have. It is concluded in the same work that 80% of the time spent by any polygon cutting algorithm is consumed by the computation of intersections, having as lower bound the order $\mathcal{O}(nm)$. In the work presented in [21], it is proposed that the extension to the algorithm involves a labeling phase of $\mathcal{O}(k)$, where k is the number of intersections, which corresponds to the result obtained in our analysis.

5.3.2. Point insertion in Tree Data Structures

If we look at the algorithm shown in Algorithm 9, it assumes that the insertion of points in a quadtree is done at the leaf level so that there is always a maximum of one point for each quadrant. This is accomplished through a division of each quadrant into four sub regions, using the midpoint of each quadrant as the vertical and horizontal axis of division.

To generalize this behavior, the algorithm was modified allowing a quadtree to use objects corresponding to a family of division and insertion strategies. With this, the insertion of points will be determined as a strategy to follow that depends on how the `insert` method is implemented.

```
1  export default class QuadTree {
2    constructor(boundary, capacity, divisionAlgorithm) {
3      /** @type {QuadTree} */
4      this.northWest = null;
5      /** @type {QuadTree} */
6      this.northEast = null;
7      /** @type {QuadTree} */
8      this.southWest = null;
9      /** @type {QuadTree} */
10     this.southEast = null;
11
12     this.boundary = boundary;
13     this.capacity = capacity;
14     this.divisionAlgorithm = divisionAlgorithm;
```

```
15      this.isLeaf = true;
16      this.points = [ ];
17      this.parent = null;
18    }
19
20    // inserts a point inside the quadtree
21    insert(point) {
22      this.divisionAlgorithm.insert(this, point);
23    }
```

In this case, we can see how we delegate the entire insertion process to the insertion algorithm. Next, we will detail the insertion algorithm using divisions in the middle, and divisions at the insertion points.

5.3.2.1. Half Point Division Algorithm

The algorithm is similar to the one shown in Algorithm 9, except for the following details. First, we add a parent parameter, to have reference to the quadrant that originated the current quadrant, which allows us to hierarchically scale until we reach the root of the tree.

Furthermore, we explicitly detail that division into four quadrants always by using the midpoint of each quadrant.

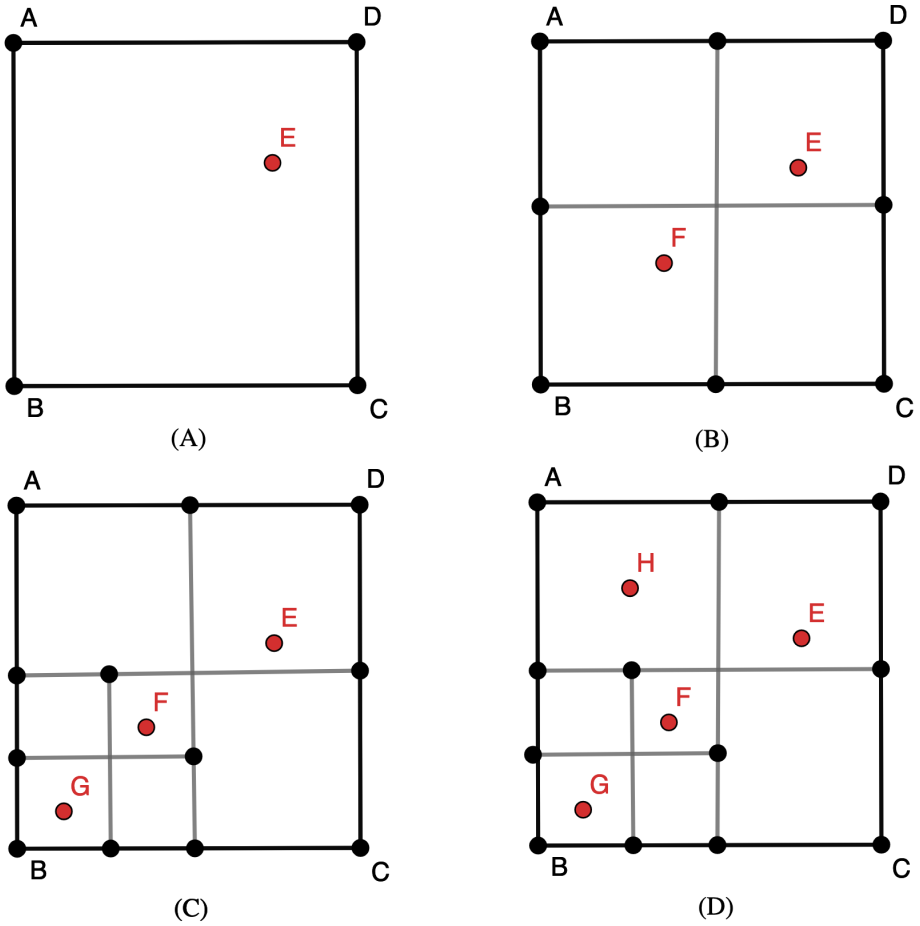


Figure 5.19: Example of Half Division Algorithm. (A) Insertion of point E. (B) Insertion of point F. (C) Insertion of point G. (D) Insertion of point H.

Algorithm 35 Half Point Division Inserting

```
1: Input
2:     Q          quadtree
3:     p          point to be inserted in the tree
4: Output
5:                True if the point was inserted. False otherwise.

6: function INSERT(Q, p)
7:     if current quadrant does not contain p then
8:         return False
9:     if p is already inserted on Q then
10:        return False
11:    if Q is a leaf then
12:        Q_points ← current points stored in Q
13:        if capacity of Q is enough to handle p then
14:            add p to Q_points
15:            return True
16:        else
17:            mark Q as non leaf
18:            DIVIDE(quadtree)
19:            reinsert every point of Q_points into the quadtree
20:            return SUBDIVIDEDINSERTION(Q, p)
21:    else
22:        return SUBDIVIDEDINSERTION(Q, p)
23: end
```

The `SubdividedInsertion` used is the same as the one defined in Algorithm 10. Now we will detail how the divide method implemented in this algorithm is.

Algorithm 36 Half Point Division Dividing

```
1: Input
2:     Q          quadtree
3: Output
4:                Mutates the quadtree, dividing it into four quadrants

5: function DIVIDE(Q)
6:     box ← Q_boundary
7:     midPoint ← get mid point from box
8:     (Q_nw, Q_ne, Q_sw, Q_se) = divide Q using midPoint
9:     Q.northWest ← Q_nw
10:    Q.northEast ← Q_ne
11:    Q.southWest ← Q_sw
12:    Q.southEast ← Q_se
13:    assign Q as parent to Q.northWest, Q.northEast, Q.southWest and Q.southEast
14: end
```

5.3.2.2. Arbitrary Point Division Algorithm

This algorithm differs from the previous one, mainly in how the subregions are divided according to the point to be inserted. Note that there is no reinsertion because there is no storage capacity for points in the quadrants, since strictly speaking, the points will only serve to subdivide the space.

The above indicates that the insert algorithm is faster, however it is subject to more edge cases that were not presented in the previous algorithm. These cases correspond to when it falls on a corner point or, on a segment of the edge of the quadrant. In those cases, there will be quadrants that will not exist, and therefore, they must be treated in a particular way.

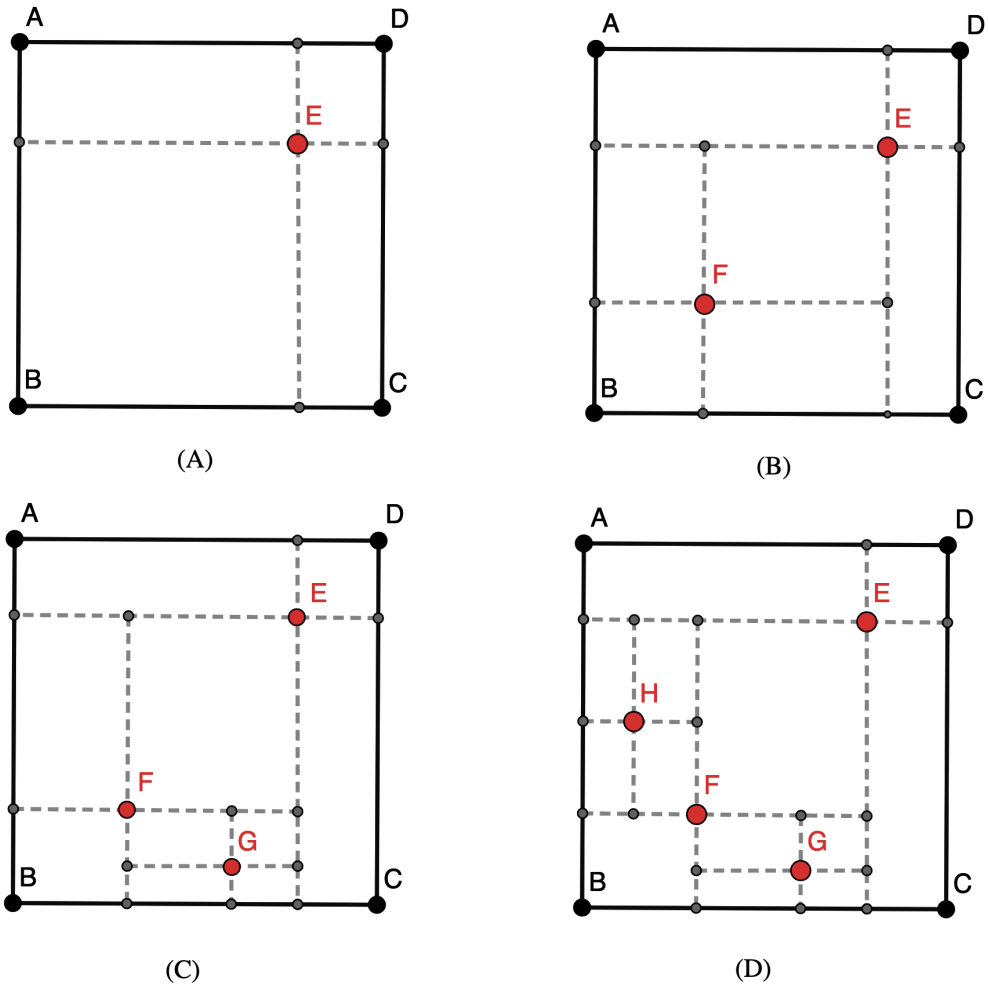


Figure 5.20: Example of Arbitrary Point Insertion Algorithm. (A) Insertion of point E. (B) Insertion of point F. (C) Insertion of point G. (D) Insertion of point H.

Algorithm 37 Arbitrary Point Division Inserting

```
1: Input
2:     Q          quadtree
3:     p          point to be inserted in the tree
4: Output
5:                True if the point was inserted. False otherwise.

6: function INSERT(Q, p)
7:     if Q is null then
8:         return False
9:     if current quadrant does not contain p then
10:        return False
11:    if p is already inserted on Q then
12:        return False
13:    if Q is a leaf then
14:        mark Q as non leaf
15:        DIVIDE(quadtree, point)
16:    else
17:        SUBDIVIDEDINSERTION(Q,p)
18:    return false
19: end
```

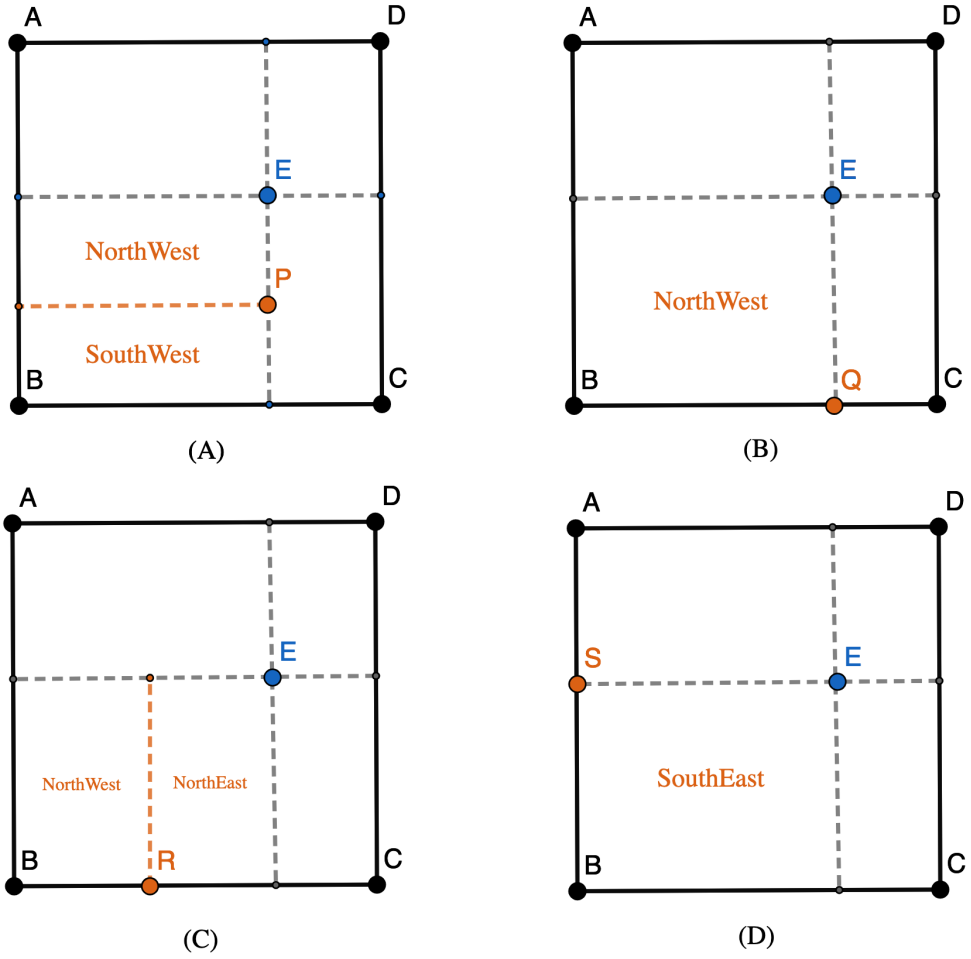


Figure 5.21: Example of Arbitrary Point Insertion Problems. (A) Insertion of point P in right edge of quadrant. (B) Insertion of point Q in bottom right corner of quadrant. (C) Insertion of point R in bottom edge of quadrant. (D) Insertion of point S in top left corner of quadrant.

Algorithm 38 Arbitrary Point Division Dividing

```
1: Input
2:     Q          quadtree
3:     p          point to be inserted in the tree
4: Output
5:                True if the point was inserted. False otherwise.

6: function DIVIDE(Q, p)
7:     box ← Q_boundary
8:     boxEdges ← get the edges from box
9:     boxCorners ← get the corner points from box
10:    (Q_nw, Q_ne, Q_sw, Q_se) = divide Q using p coordinates
11:    Q.northWest ← Q_nw
12:    Q.northEast ← Q_ne
13:    Q.southWest ← Q_sw
14:    Q.southEast ← Q_se                    /*Consider that some quadrants are not*/
                                             valid. Now we will overwrite those of the
                                             corners.
15:    if p is equal to top left of boxCorners then
16:        set all quadrants except Q.southEast to null
17:    else if p is equal to the top right of boxCorners then
18:        set all quadrants except Q.southWest to null
19:    else if p is equal to the bottom left of boxCorners then
20:        set all quadrants except Q.northEast to null
21:    else if p is equal to the bottom right of boxCorners then
22:        set all quadrants except Q.northWest to null   /*If not, then over-*/
                                                          write those of the
                                                          point over edge.
23:    else if p is over the left edge of boxEdges then
24:        set Q.northWest and Q.southWest to null
25:    else if p is over the top edge of boxEdges then
26:        set Q.northWest and Q.northEast to null
27:    else if p is over the right edge of boxEdges then
28:        set Q.northEast and Q.southEast to null
29:    else if p is over the bottom edge of boxEdges then
30:        set Q.southEast and Q.southWest to null   /*If no overwrite hap-*/
                                                     pened, then the point
                                                     divided the quadrant
                                                     in four.
31: end
```

5.3.3. Generating the Initial Mesh

Taking into consideration each of the algorithms shown in this work, in particular Greiner Hormann's Extended polygon cutting algorithm, we can explain how each of the mesh polygons is generated. The three important processes in its generation are detailed below: Creation of the polygons using the cutting algorithm, Obtaining the problem points and finally

the arrangement of the consistency of the polygons.

5.3.3.1. Generating new polygons from a contour geometry

Algorithm 39 Building the initial mesh

```
1: Input
2:     polygon     Polygon to be clipped. Can be drawn by the user, or uploaded by a file
3: Output
4:                 Mutate the storage of Polygons, adding the new clipped polygons

5: function GenerateNewPolygons(polygon)
6:     remove polygon from the polygon storage
7:     problemPoints ← new Circular()
8:     newPolygons ← [ ]
9:     foreach quadrant in leaves from the quadtree do
10:        boundary ← get boundary points from the quadrant
11:        clippedPolygons ← new GreinerHormannAlgorithm().clip(polygon, boundary)
12:        if exists polygons in clippedPolygons then
13:            newProblemPts ← GetProblemPoints(polygon, clippedPolygons, problemPoints)
14:            add all points from newProblemPts to problemPoints, without repetition
15:            foreach newPoly in clippedPolygons do
16:                add quadtrantParent reference to newPoly, pointing to quadrant
17:                add newPoly to newPolygons
18:            end
19:    end
20:    FixPolygons(newPolygons, problemPoints)
21: end
```

The polygon mesh generation algorithm receives an initial polygon, which corresponds to the geometry contour we want to mesh, either created by the user or delivered in an `OFF` file. When starting the application, the polygon immediately generates a quadtree or kd-tree that generates quadrants, so it is enough to take each of them and apply the Extended Greiner Hormann Algorithm of cutting polygons to generate the new polygons, such as It is shown in line 11. Based on the new polygons obtained and the original polygon, we invoke the `GetProblemPoints` function to obtain those problem points and store them in a list as in line 14. From line 15 to 18 , we save the new generated polygons, and set the quadrant that generated each polygon as property. Finally, after all the polygons are generated, we invoke the `FixPolygons` function, using the problem points obtained and the new polygons, to have a consistent mesh.

5.3.3.2. Obtaining the possible problematic points

Given a starting polygon $\mathcal{P}$, we say that a point is problematic if there is a point that belongs to some polygon created from P by means of a cutting algorithm, and is not part of the original

polygon. According to this definition, we can easily find all the problematic points by going through each of the points of the polygons generated by the cutting algorithm, and verify if they exist in the original polygon.

Algorithm 40 Getting the problematic points after clipping

```
1: Input
2:     polygon     Polygon to be clipped. Can be drawn by the user, or uploaded by a file
3:     clipped     List of lists of new polygons, generated by the clipping algorithm
4:     probPts     Circular List of problematic points
5: Output
6:                 Circular List of new problematic points

7: function GetProblemPoints(polygon, clipped, probPoints)
8:     newProbPoints ← new Circular()
9:     foreach polygon in clipped polygons do
10:        foreach point in polygon.points do
11:            if point is neither inside polygon nor probPoints then
12:                add point to newProbPoints, with no repetitions
13:        end
14:    end
15:    return newProbPoints
16: end
```

The natural implementation corresponds to a nested cycle, and for each one of the polygons generated by the cutting algorithm, we iterate through all its points again. If the point in question does not exist in the original polygon, we add it to a list that contains the result, as seen on line 12. Something that is not explicit is that verifying if a point exists among those that make up a polygon, is equivalent to verifying if an integer exists inside a list of integers. This is because the points are represented by their index, and the polygons keep within themselves a list of indexes representing their points.

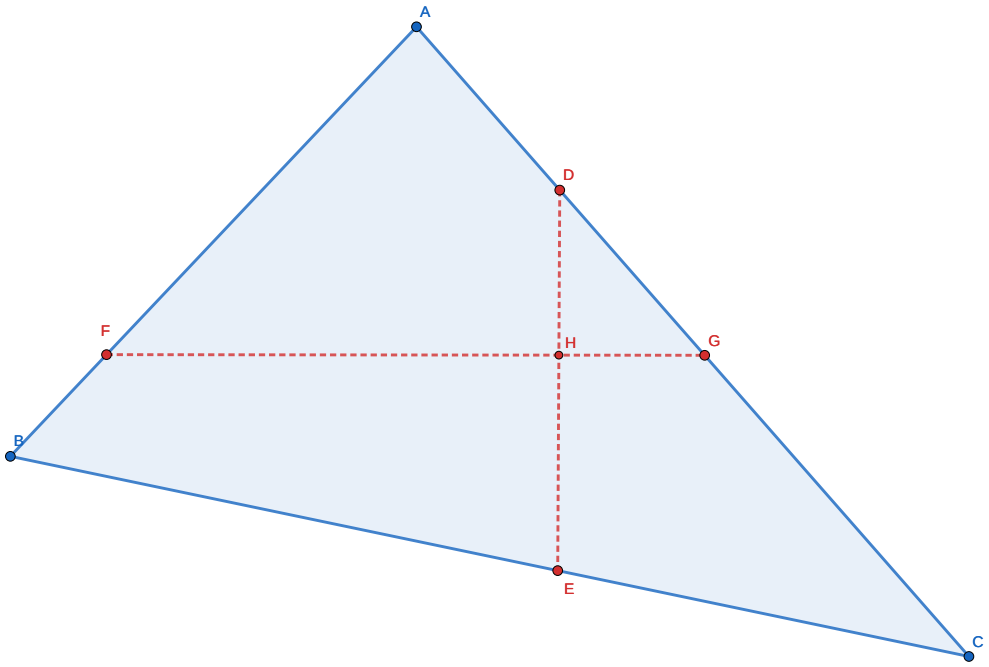


Figure 5.22: Getting problematic points after clipping.

Let us consider the figure 5.22. The BCA triangle has been cut into four new sub-polygons according to the midpoint division algorithm, generating the $FHDA$ and $BEHF$ quadrilaterals, and the HGD and $ECGH$ triangles. Of the original points of the BCA triangle, there are now new $DEFGH$ points that did not exist before, which is why these points say they are called problem points.

5.3.3.3. Fixing polygons consistently with problem points

Finally, to fix the polygons, what we do go through each one of the created polygons, and for each one of the problematic points, we check if they exist between any of their edges, without considering the extreme points. If the conditions are met, the point is inserted in the corresponding place.

Algorithm 41 Fix the polygons with the new generated points

```
1:  Input
2:      clipped     List of lists of new polygons, generated by the clipping algorithm
3:      probPts     Circular List of problematic points
4:  Output
5:                  Circular List of new problematic points

6:  function FixPolygons(clipped, probPoints)
7:      foreach polygon in clipped do
8:          circularPoly ← new Circular(polygon.points)
9:          foreach p in probPoints do
10:             currentNode ← circularPoly.head
11:             do
12:                 currentEdge ← new Edge(currentNode.value, currentNode.next.value)
13:                 if p is inside currentEdge and is not equal to any of its endpoints then
14:                     add p after currentNode.value in circularPoly
15:                 currentNode ← currentNode.next
16:             while currentNode is different from circularPoly.head
17:         end
18:         assign the circularPoly points as the new ones of polygon
19:         save polygon into the storage
20:     end
21: end
```

We can see that in the implementation, for each edge without its ends, it is asked if the point in question is in the middle of it or not, as seen in line 13 and 14. Finally, the list of points must be updated, mutating the polygon and overwriting the one that is stored in the `PolygonStorage` as it is done in lines 18 and 19.

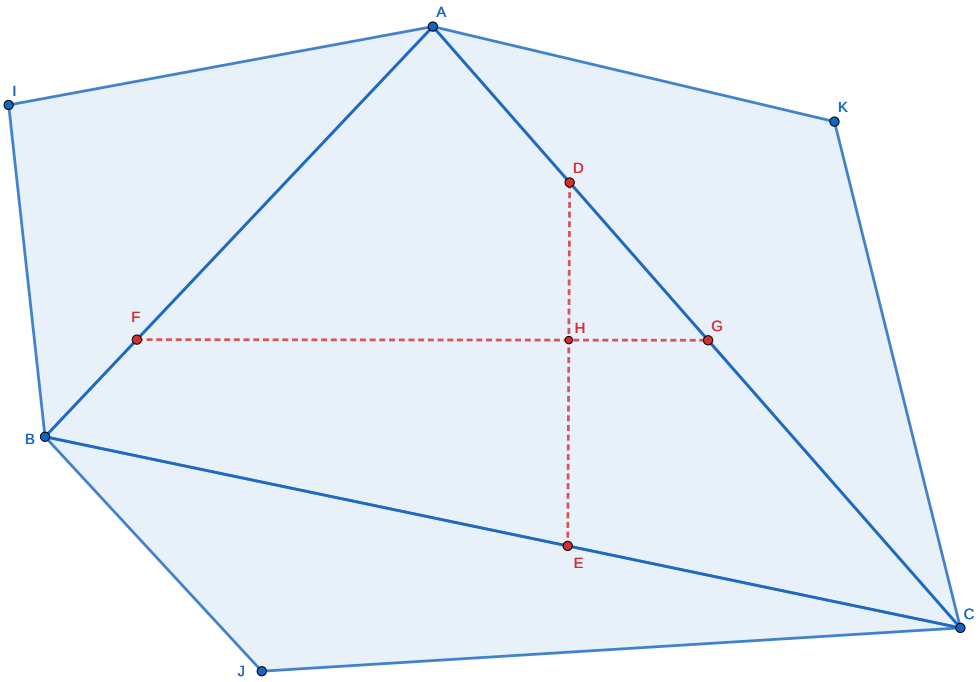


Figure 5.23: Fixing polygons after clipping.

Consider the neighbors of the triangle shown above, shown in the figure. These correspond to the BAI, BJC, and ACK triangles. Note that when adding the $DEFGH$ points to the initial triangle, the neighbors are left with problems of agreement, since, for example, without applying a correction, the segment AB of the BAI triangle is not aware of the new point F. This is why the `FixPolygons` function check each of the neighbors to locate those edges that need the intersection of a problem point.

5.3.4. Refinement Algorithms

5.3.4.1. Tree Refinement

To perform a refinement through a quadtree, it is necessary that the mesh has been generated through the application, and not imported from outside. This is because when importing a mesh already created, it implies not having a generating quadtree to refer to, and therefore subdivide the polygons.

For those polygons that were created from quadtrees, allowed the user to select by means of clicks, which are the polygons that they are interested in refining. When this is done, the polygon gets its generating quadtree by means of a reference saved in the `Polygon` object, and proceeds to perform a division into four new regions.

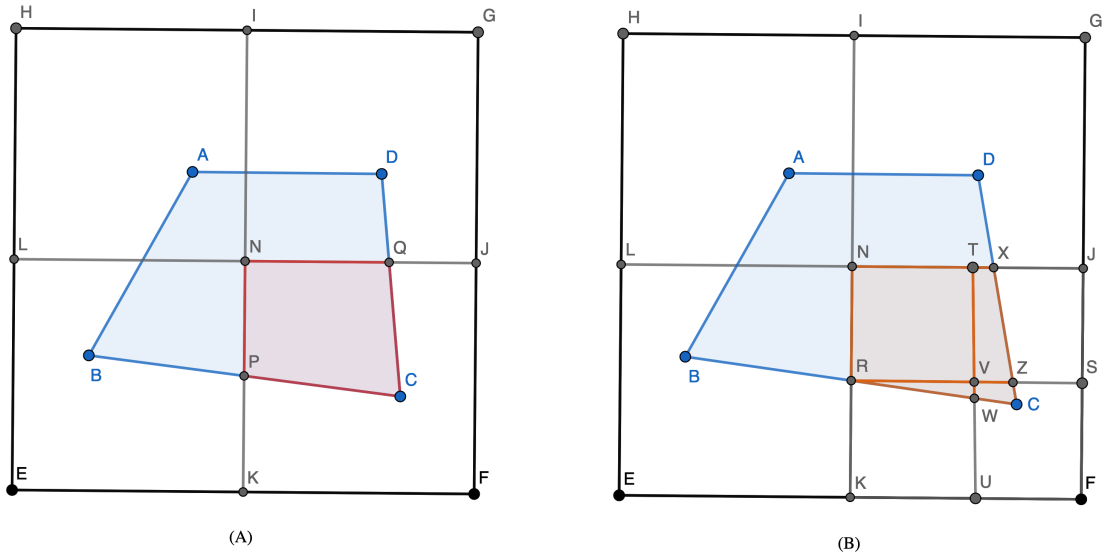


Figure 5.24: Refinement of a polygon using quadtrees. (A) The user clicks on the $PCQN$ polygon. (B) The quadrant that generated the polygon is subdivided into four equal quadrants. Then, with the new quadrants, the polygon is cut into four new ones.

As an example of the procedure, an example is shown in the figure. The $ABCD$ polygon is cut into four sub-polygons using a quadtree. If we assume that a user clicks on the $PCQN$ polygon, then the algorithm accesses the quadtree that generated the polygon (in this case $NFKJ$) and it is divided into four equal quadrants.

With these quadrants, the $PCQN$ polygon is subdivided into four sub-polygons, each one referencing the quadrants that generated them.

Algorithm 42 Tree Refinement

```
1: Input
2:     polys      intersected polygons to be refined
3:     mesh       a reference to the mesh being constructed
4: Output
5:                A list of the new generated polygons.

6: function RefineByQuadtree(polys, mesh)
7:     newPolygons ← [ ]
8:     foreach polygon in polys do
9:         neighbors ← get neighbors for polygon
10:        quad ← polygon.quadrant
11:        divide quad in four quadrants using the Algorithm 36.
12:        set quad as a non leaf quadrant
13:        newLeaves ← get the four new leaves from the divided quadtree
14:        cut polygon using Greiner Hormann clip method
15:        fix the neighborhood of polygon
16:        add all the cut polygons (if they exists) to newPolygons
17:        delete polygon from PolygonStorage (if it was cut)
18:    end
19:    return newPolygons
20: end
```

5.3.4.2. Splitting Longest Edge

The algorithm for dividing the longest segment of a polygon consists of a generalization of that defined for triangles. First, the segment with the longest length is searched. Subsequently, the midpoint of said segment is obtained, and the areas of all the polygons formed by the cut segment that joins the midpoint and a point inside the polygon, are inspected.

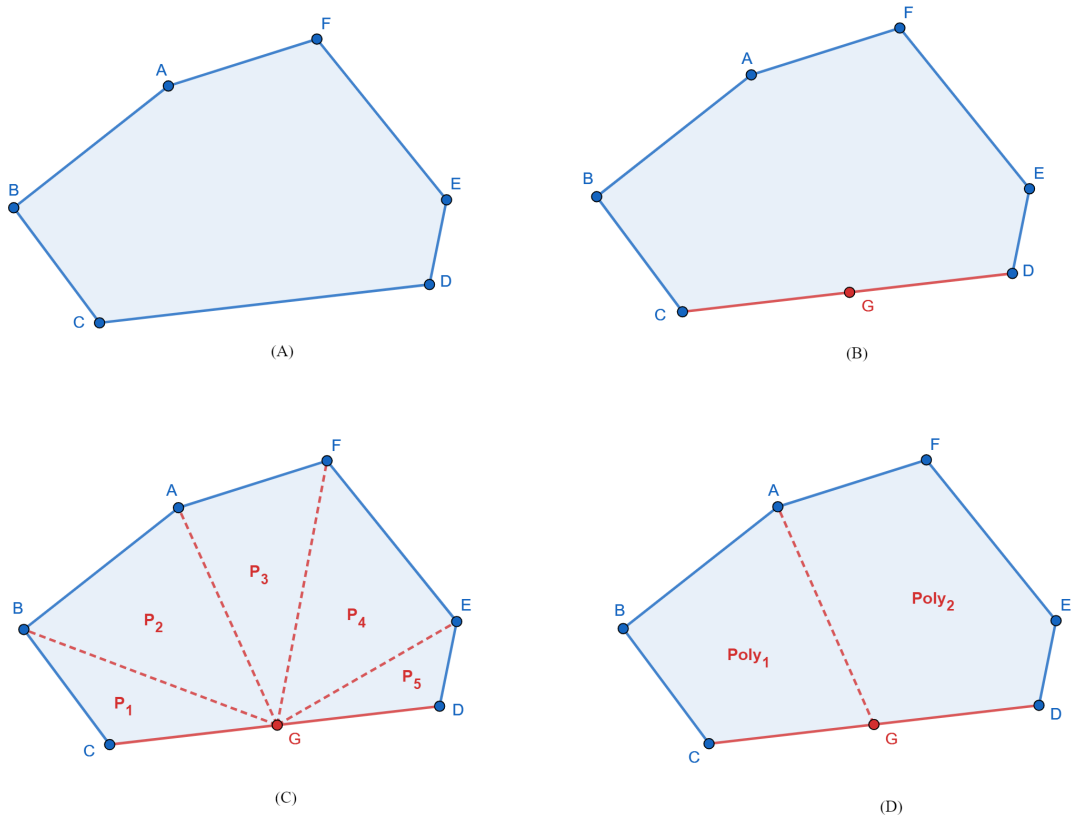


Figure 5.25: Example of how a polygon is refined using the longest edge splitting algorithm. (A) is the polygon to be refined. (B) select the longest segment $\overline{CD}$ and search for the midpoint G (C) inspect the area of the possible polygons (D) select the partition that has the smallest difference between its areas.

For example, in the figure if we consider the cut segment as $\overline{BG}$, it generates two polygons: BCG and $BGDEFA$. However, the difference between their areas is not the best, since there are partitions that further minimize this difference.

When we find the segment that cuts the polygon with the minimum difference in areas (in our example, that segment is $\overline{AG}$), we proceed to cut the polygon and return the two that are generated (in our example $Poly_1$ and $Poly_2$).

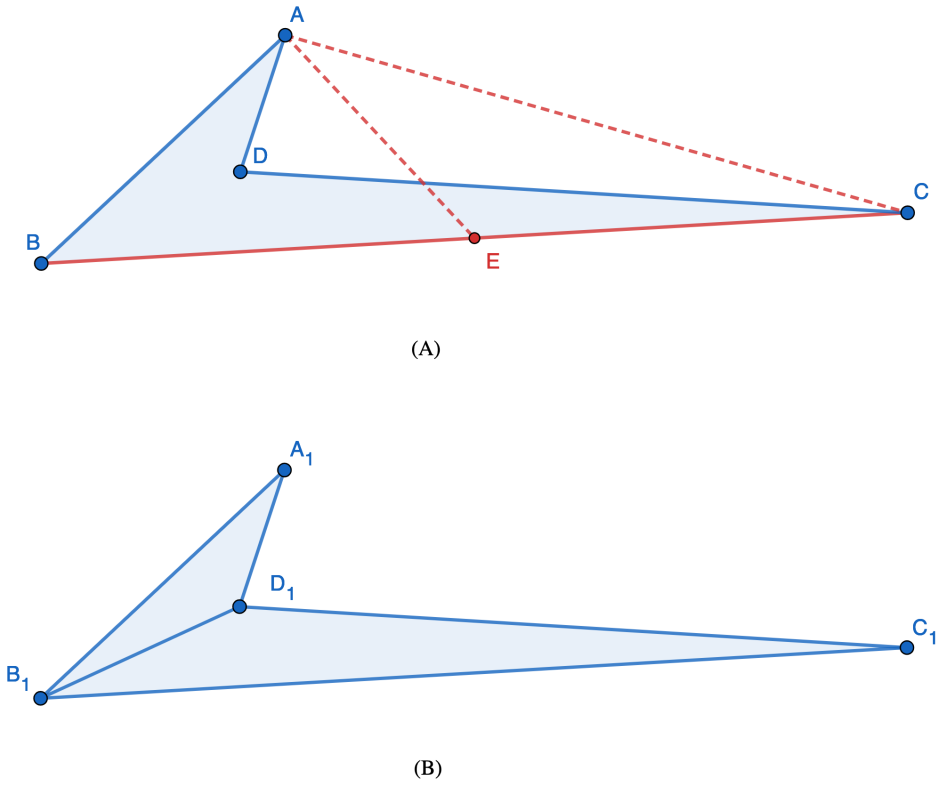


Figure 5.26: Algorithm problem for non-convex polygons. (A) The algorithm determines that segment $\overline{AE}$ generates the best difference between polygons ABE and AEC. However, these polygons are non-existent. (B) The solution is to triangulate the polygon when it is not convex.

However, the algorithm suffers from a problem when polygons are non-convex, because the segments between the midpoint of the longest side and the interior points do not always lie inside the polygon (Figure 5.26). These non-contained diagonals inside the polygon generate non-existent polygons as seen in the Figure 5.26, giving erroneous results.

To avoid this, the polygon is triangulated, so that at a later stage, the algorithm can be applied to the generated triangles in case more refinement is needed. The implementation of the algorithm explained above is detailed below.

Algorithm 43 Splitting Longest Edge

```
1: Input
2:     polys      intersected polygons to be refined
3:     mesh       a reference to the mesh being constructed
4: Output
5:                A list of the new generated polygons.

6: function REFINE(polys, mesh)
7:     newPolygons ← []
8:     problemPoints ← []
9:     foreach polygon in polys do
10:        if polygon is not convex then
11:            triangles ← triangulate polygon
12:            add every triangle in triangles to newPolygons
13:        else
14:            maxEdge ← get max edge from polygon
15:            midPoint ← get mid point from maxEdge
16:            add midPoint to PointStorage
17:            add midPoint to problemPoints
18:            insert index of midPoint into polygon
19:            (poly_1, poly_2) ← GETBESTPARTITION(maxEdge, midPoint)
20:            save poly_1 and poly_2, fixing mesh
21:            fix the neighborhood of polygon in mesh
22:            remove polygon from PolygonStorage
23:    end
24:    return newPolygons
25: end
```

5.3.4.3. Centroid

This algorithm consists of calculating the centroid of the polygon, to subsequently join each interior point with the said point. The result for convex polygons is a collection of triangles, which all share the centroid as a common point.

This algorithm also has problems if the polygon is not convex, due to the existence of segments between centroid and point not completely contained within the polygon. In these cases, the polygon is triangulated, without using the centroid.

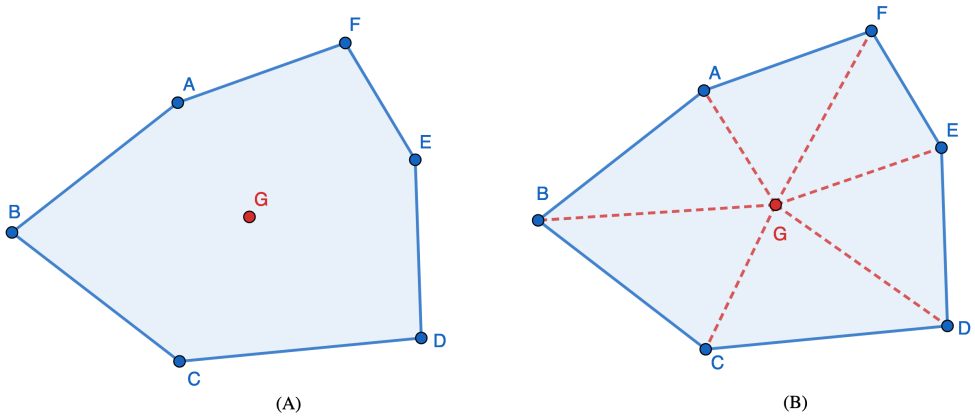


Figure 5.27: Example of refinement of a polygon using the centroid. (A) obtaining the centroid point G (B) triangles are obtained by joining each internal point, with the centroid G.

Algorithm 44 Centroid Refinement

```
1:  Input
2:      polys       intersected polygons to be refined
3:      mesh        a reference to the mesh being constructed
4:  Output
5:                  A list of the new generated polygons.

6:  function REFINE(polys, mesh)
7:      newPolygons ← [ ]
8:      foreach polygon in polys do
9:          if polygon is not convex then
10:             triangles ← triangulate polygon
11:             add every triangle in triangles to newPolygons
12:         else
13:             centroid ← FINDCENTROID(polygon)
14:             foreach point in polygon.points do
15:                 auxPoints ← [point, point.next, centroid]
16:                 create new Polygon from auxPoints
17:                 add the new Polygon to newPolygons
18:             end
19:             remove polygon from PolygonStorage
20:     end
21:     return newPolygons
22: end
```

5.3.4.4. Centroid with Replication

This algorithm consists of using the centroid, to create a new polygon smaller than the original one at the center of it, and new polygons in between. For this, we take the midpoints of each

of the segments that join the internal points and the centroid. Then, the inner points of the larger polygon are joined to the corresponding points of the inner polygon, and finally, the inner points are joined together. This procedure forms a polygon identical to the original in the middle of it, and quadrilaterals between them.

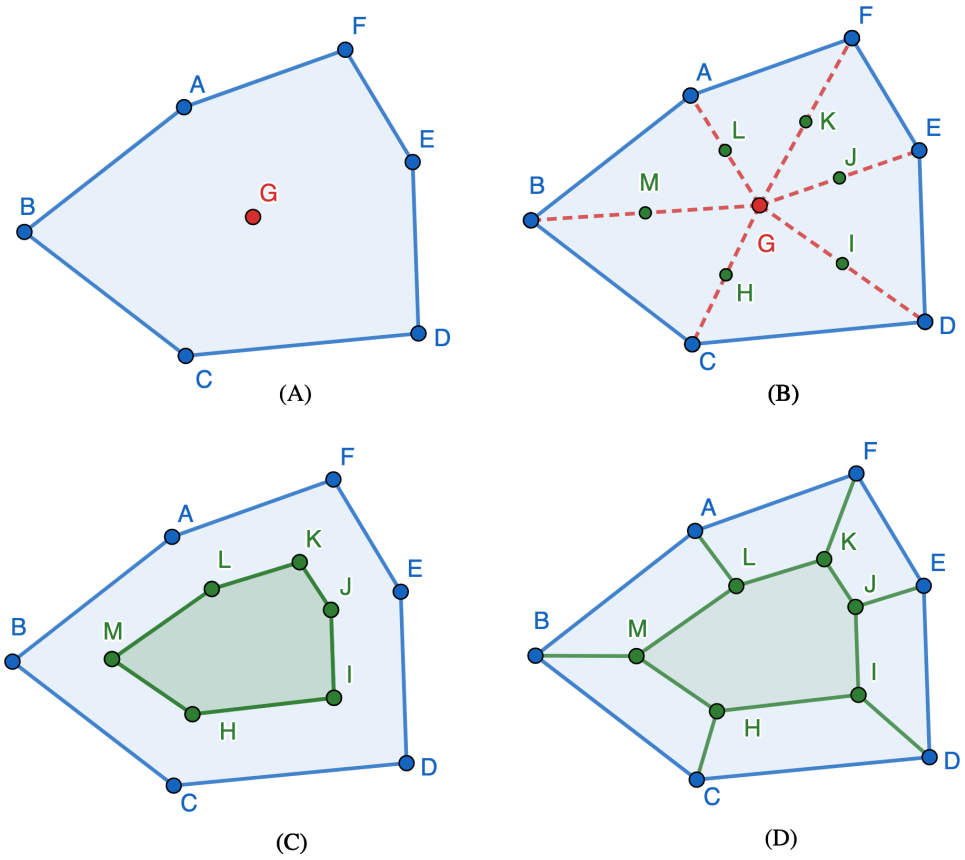


Figure 5.28: Refinement example using centroid with replication. (A) obtain the centroid G of a polygon (B) obtain the midpoints H, I, J, K, L, M of each segment that joins the internal points with the centroid (C) form the internal polygon $HIJKLM$ (D) form the internal quadrilaterals.

This algorithm has the same problems for non-convex polygons, so they triangulate instead of using this refinement. The implementation of this strategy is detailed below.

Algorithm 45 Centroid with Replication

```
1: Input
2:     polys       intersected polygons to be refined
3:     mesh        a reference to the mesh being constructed
4: Output
5:                 A list of the new generated polygons.

6: function REFINE(polys, mesh)
7:     newPolygons ← [ ]
8:     foreach polygon in polys do
9:         if polygon is not convex then
10:            triangles ← triangulate polygon
11:            add every triangle in triangles to newPolygons
12:        else
13:            centroid ← FINDCENTROID(polygon)
14:            interiorPoints ← [ ]
15:            foreach p in polygon.points do
16:                intPoint ← get middle point from p to centroid
17:                add intPoint to interiorPoints
18:            end
19:            create polygon using interiorPoints
20:            save polygon to PolygonStorage
21:            add polygon to newPolygons
22:            for i ← 0 to polygon.points.length do
23:                P1 ← polygon.points[i]
24:                P2 ← polygon.points[i + 1  mod polygon.points.length]
25:                P3 ← interiorPoints[i + 1  mod interiorPoints.length]
26:                P4 ← interiorPoints[i]
27:                create new polygon from P1, P2, P3, P4
28:                save the new polygon to PolygonStorage
29:                add the new polygon to newPolygons
30:            end
31:            remove polygon from PolygonStorage
32:    end
33:    return newPolygons
34: end
```

Chapter 6

Results

In this section we show the results obtained in the experiments and tests we performed with the polygon mesh generator. We divide the results into the following categories: (1) Time analysis for the polygon clipping algorithm and the initial mesh creation algorithm for both quadtree (in both its half division and point division strategies) and kdtree. (2) Characteristics of the initial meshes generated by quadtrees and kdtrees. (3) Application of refinement algorithms to improve the quality of the meshes (4) Successive refinements to all those polygons that do not meet a certain quality condition (5) Analysis of quality metrics and finally (6) Comparison of the meshes generated by our application contrasted with those obtained by Triangle.

6.1. Time Analysis

To analyze time, we focused on measuring the duration of the process of cutting polygons against quadrilaterals, thereby simulating the generation of a polygon based on the grid of a quadtree. We then analyze the time it takes to generate an initial mesh using quadtrees and kdtrees. The results of the experiments are presented below.

6.1.1. Extended Greiner Hormann Algorithm

The timing performance results of our implementation of the extended Greiner Hormann algorithm are shown below.

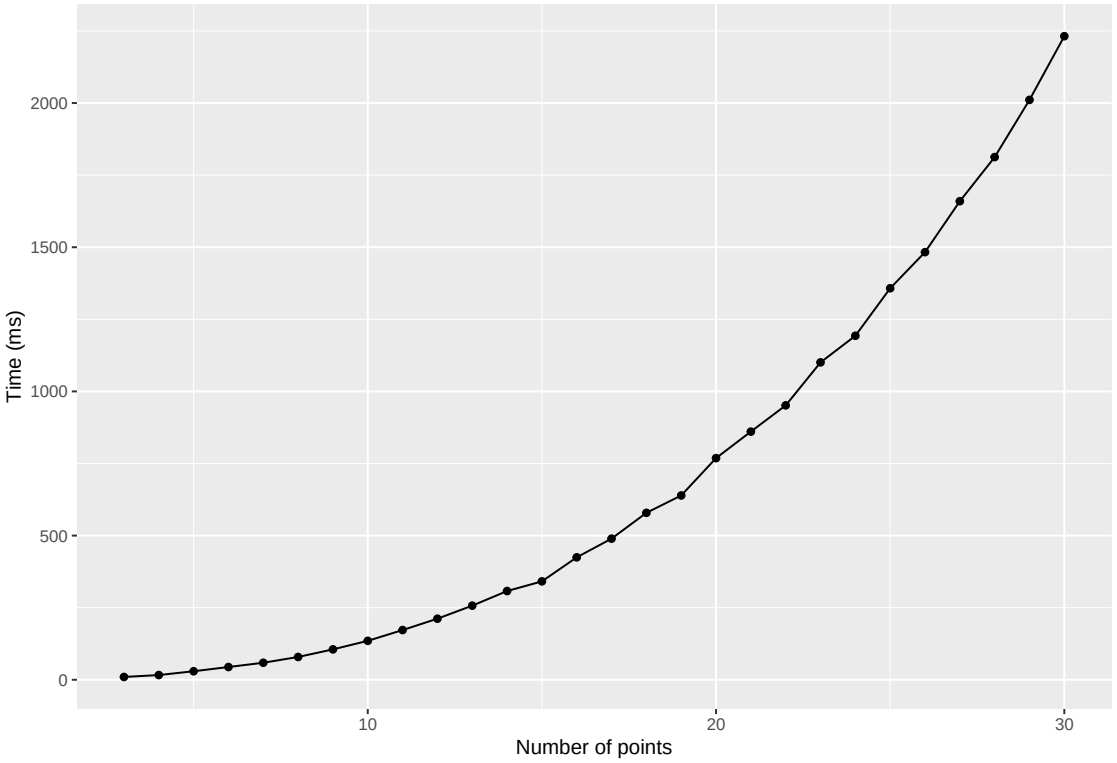


Figure 6.1: Time analysis for clipping algorithm.

From the obtained graph and the obtained fit, we obtained as a result that our implementation runs in time $\mathcal{O}(n^2)$, where n is the initial number of points of the polygon that is cut by a quadrilateral. The fit has an R^2 of 0.9981, so we consider that the trend line fits appropriately to the curve obtained.

Table 6.1: Trendline function for clipping algorithm

Trendline function	R^2
$3.5801x^2 - 40.731x + 154.13$	0.9981

6.1.2. Initial meshes creation time

To compare the initial mesh generation times, we generated a random polygon with a given number of points, and created a mesh using a quadtree (with the strategy of dividing with the midpoint and using the points that make up the polygon) and a kdtree. The results obtained from measuring the time for each strategy are as follows.

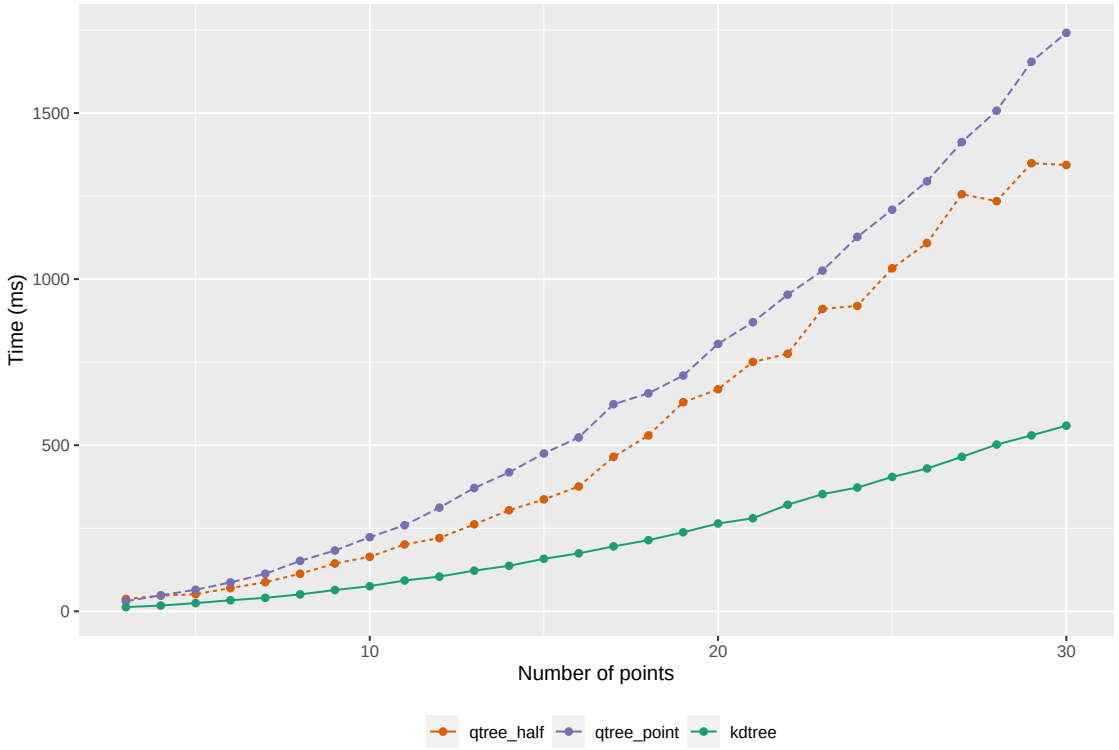


Figure 6.2: Time analysis for initial meshes.

The results obtained indicate that the algorithms run in $\mathcal{O}(n^2)$ time, differing from each other only in the constant accompanying the quadratic term. According to the graph, the best performance is obtained by the kdtree algorithm for the generation of initial meshes, followed by the division using the midpoint, and as last place the strategy of using arbitrary points.

Table 6.2: Trendline function for initial time mesh generation.

Algorithm	Trendline function	R^2
Quadtree Half Division	$1.5422x^2 + 1.7949x - 0.5408$	0.994
Quadtree Point Division	$1.7776x^2 + 4.3276x + 1.9348$	0.9995
KDtree	$0.5642x^2 + 1.9096x + 0.846$	0.9996

The explanation we found for the better performance obtained by a kdtree is the optimization it makes at the moment of ordering the points and choosing in which direction to divide the quadrant (according to the X or Y direction). However, the performance of obtaining an initial mesh is determined by the quadratic time it takes to cut the initial geometry with respect to the grid generated by the kdtree. This is why the improvement is seen only at the level of constants, and not in the order of performance of the initial mesh generation algorithm.

6.2. Initial Mesh Generation

We show now the results of creating an initial mesh for the geometry shown in Figure 6.3. This geometry is interesting to analyze because it has a non-convex angle, so it is important to see how it behaves at the level of shaping an initial mesh of polygons, as well as further refinement using a quadtree with a division strategy using the midpoint of each quadrant.

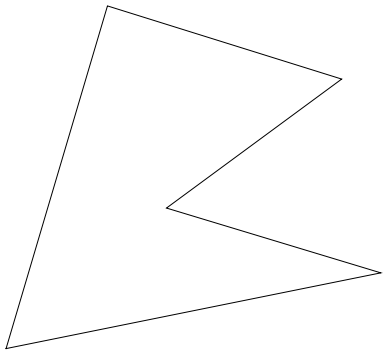
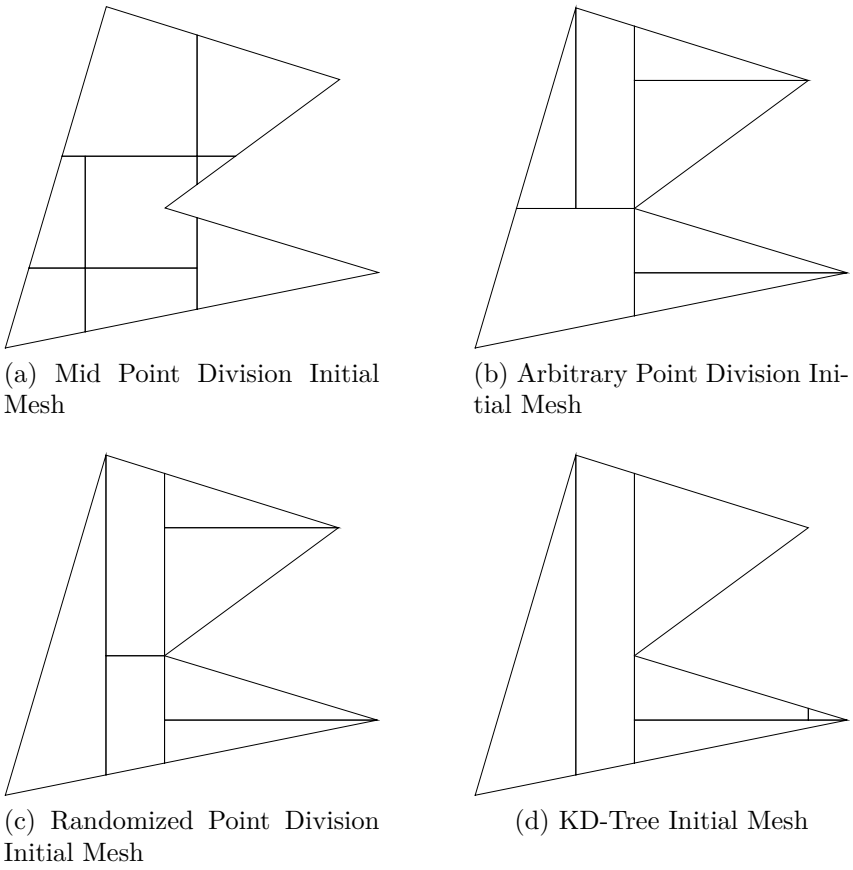


Figure 6.3: Geometry used for the creation of an initial mesh, and the subsequent analysis of different refinements.

6.2.1. Initial Meshes

Four initial meshes were obtained using two types of trees. The first tree consists of a quadtree, to which different criteria of insertion of points and division of quadrants have been applied. These are: Division using the midpoint, Division using an arbitrary point, and a random insertion of points. The fourth initial mesh is the one generated using a kd-tree, the structure we want to evaluate in this thesis in the context of polygon mesh generation.



(a) Mid Point Division Initial Mesh

(b) Arbitrary Point Division Initial Mesh

(c) Randomized Point Division Initial Mesh

(d) KD-Tree Initial Mesh

Figure 6.4: Four different strategies to obtain an initial mesh of polygons.

The first three strategies take up a quadtree for creating an initial mesh. We can see that even when the data structure is the same, there are differences in the result that is delivered, depending on the insertion order of the points. The last strategy, which uses a two-dimensional KD-Tree, alternates the X and Y coordinates to make horizontal and vertical cuts respectively, using a recursive ordering looking for the median of the points, for a better result.

Table 6.3: Main study characteristics of the initial meshes.

Division Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
Half Point Division	8	17	131.29	57.4	18861.31	66.96
Arbitrary Point Division	7	11	205.65	33.31	21555.79	52.55
Randomized Arbitrary Point Division	8	13	172.97	43.95	18861.31	39.72
KD-Tree	6	11	210.5	40.7	25148.42	22.29

According to the results obtained in table 6.3, we can see that the creation time of the initial meshes depends on the cutting complexity between the quadrant and the initial polygon. For example, we see that the number of generated polygons by the midpoint algorithm is equal to that of the random insertion criterion, however, their times differ by approximately 40%. This is attributable to the fact that the cuts made in the mid point algorithm are more

complex and require a greater number of points compared to the random insertion algorithm.

The best build time is the KD-tree algorithm. We attribute this to the lower polygon cut processing that needs to be done when choosing only one cut direction (vertical or horizontal), compared to the arbitrary point algorithm, which always performs cross cut. Consequently, the number of polygons and points in the initial mesh is less and, therefore, the generation time. The comparison is made with this algorithm because its nature is similar, demonstrating this in that the similar number of points and polygons are obtained.

Regarding the minimum average angle, the best result has the mesh obtained using the mid point algorithm (The largest average of small angles). We attribute this to the fact that due to the cuts made by the quadrants of the quadtree they are not at the point level, so there are numerous right angles, unlike the other strategies.

6.3. Quality improvements

In this section we discuss about the quality refinements implemented in this work, applied on a polygon mesh. Initially we apply a quality refinement, that is, the refinement algorithm using a centroid, the replication algorithm using the centroid, and the cut of the polygon according to the longest side, to each polygon of the initial meshes shown in the figure 6.3 in order to inspect the different characteristics of the resulting mesh.

Later we are interested in how each of the algorithms behave to converge to a mesh in which all the polygons meet a certain condition. For this analysis we start as before with an initial polygonal mesh, on which we carry out successive refinements on those polygons that do not meet a certain quality condition, such as, for example, have a greater area or a side greater than a certain limit value. After that, we examine the important results for our investigation, such as the execution time, the minimum angle, the length minimum of one segment of the polygon, and the number of points and polygons generated.

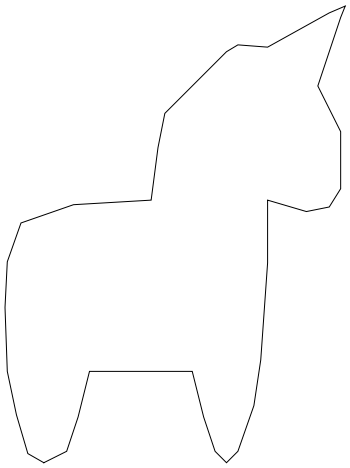
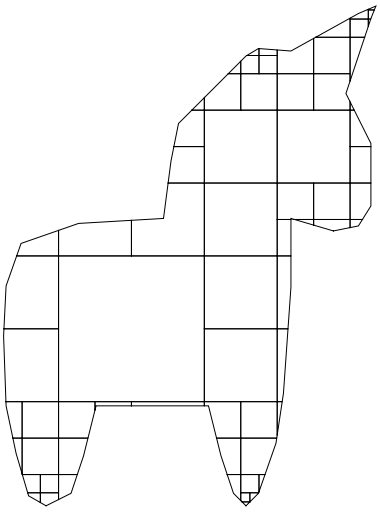


Figure 6.5: Contour of a unicorn geometry used in quality refinements.

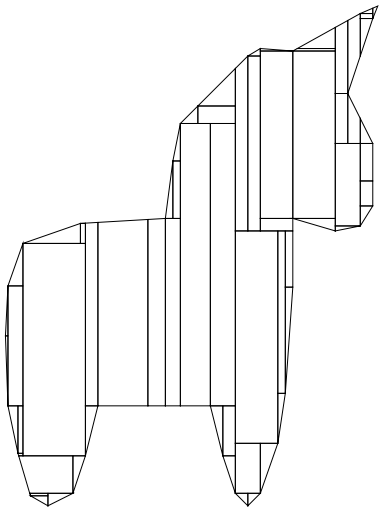
For this section we use a unicorn-shaped geometry shown in figure 6.5, which has 37 points and multiple non-convex angles.

6.3.1. Initial Meshes

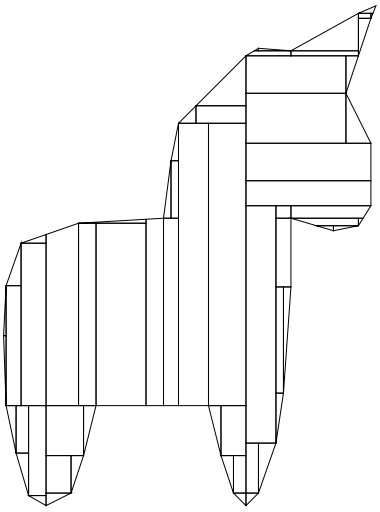
The first thing we did was generate the four types of initial meshes created from the unicorn geometry, shown in the figure 6.5, on which we applied the different quality refinements. As before, four different processes were applied: The use of a quadtree with three different strategies of insertion of points and division of quadrants, and the use of a two-dimensional KD-Tree.



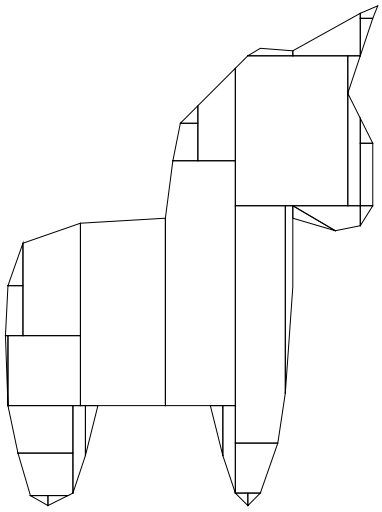
(a) Mid Point Division Initial Mesh



(b) Arbitrary Point Division Initial Mesh



(c) Randomized Point Division Initial Mesh



(d) KD-Tree Initial Mesh

Figure 6.6: Four different initial meshes of the unicorn geometry.

Table 6.4: Main study characteristics of the initial meshes of the unicorn geometry.

Division Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
Half Point Division	68	143	30.68	71.49	1979.17	1772.53
Arbitrary Point Division	66	101	47.32	51.34	2039.15	913.83
Randomized Arbitrary Point Division	67	102	42.27	51.27	2008.71	902.04
KD-Tree	35	67	50.2	46.46	3845.25	334.97

In relation to the number of elements, we see that the largest number of points and polygons generated is produced by the division algorithm using the midpoint, this is because there are no cuts in the points of the polygon contour, a greater amount is produced of quadrilaterals or figures that have at least a right angle. The least amount of elements is produced by the KD-Tree algorithm, as a consequence of the division at each point through only one axis. We also see that unlike arbitrary point algorithms, randomized or not, the KD-Tree algorithm generates fewer problematic quadrilaterals, that is, those in which the difference between the longest and smallest segment is very large.

Regarding the shape of the polygons obtained on average, we note that the largest figures are found in the mesh generated by the KD-Tree algorithm, obtaining on average the longest minimum segment length and the largest area. As for the minimum angle, on average the Half Point algorithm has the highest average value, attributed to the fact that there are a large number of right angles that increase the average and make it tend towards said value.

In relation to time, we can see that the KD-Tree algorithm is approximately 3 times faster than quadtree's arbitrary point algorithms, and almost 6 times faster than the mid point algorithm. We attribute this behavior to the fact that the number of calculated intersections (and consequently the generation of quadrants and generated polygons) is less than in the other algorithms, so the processing time is lower.

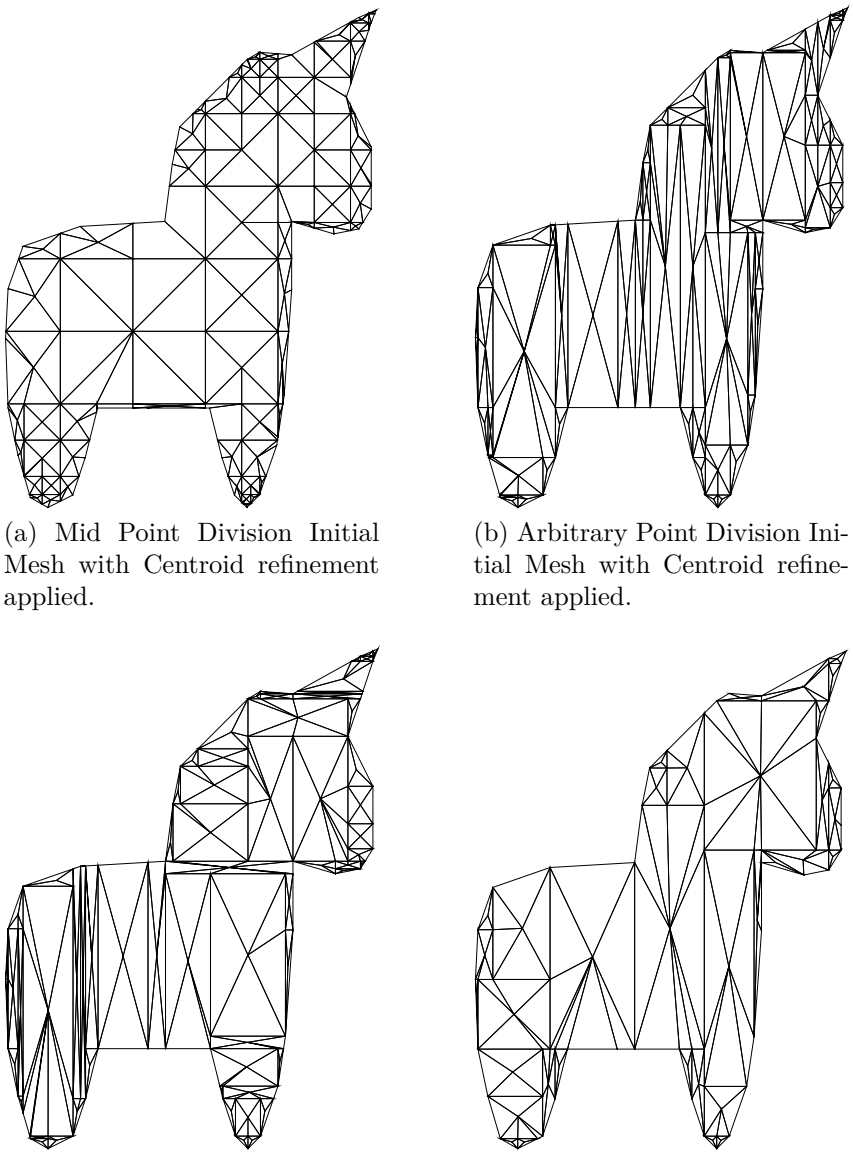
6.3.2. Quality Refinements to Bad Polygons

We can also perform a quality refinement on a mesh. Here we seek to further refine those polygons that belong to a sector of our interest, or those that do not meet the quality criteria provided by the user. The quality refinement algorithms implemented in this work are as follows: Join each of the points with the centroid of the polygon (`Centroid Algorithm`), creating a replica of the polygon having as its center the centroid of the original polygon (`Centroid Replication Algorithm`), and finally the union of the midpoint that divides the largest segment of the polygon, with some of its points, maintaining a similarity between the areas of the resulting polygons (`Splitting Longest Edge`).

In the next subsection we see how these quality refinements behave when applied to each of the initial meshes generated by each of the different strategies. The quality refinements initially are applied to each one of the polygons, to obtain a better understanding of the relevant study parameters.

6.3.2.1. Centroid refinement to Initial Meshes

The refinement using the centroid explained in the section 5.3.4.3 involved the calculation of the centroid, and the union of each point of the polygon with it. This procedure was possible only if the polygon is convex, due to degenerate diagonals that can appear in non-convex polygons. As a solution, the non-convex polygons are polygonized, instead of applying the centroid algorithm. The results obtained after applying the refinement centroid algorithm to each of the different initial meshes generated by our application are as follows:



(a) Mid Point Division Initial Mesh with Centroid refinement applied.

(b) Arbitrary Point Division Initial Mesh with Centroid refinement applied.

(c) Randomized Point Division Initial Mesh with Centroid refinement applied.

(d) KD-Tree Initial Mesh with Centroid refinement applied.

Figure 6.7: Applying centroid refinement to the different initial meshes.

Analyzing the four different results, we can see graphically in the figure 6.7 that the meshes generated on the initial meshes of the mid point division algorithm and that of kd-

tree show better results. Using the arbitrary point algorithm in normal and random insertion, we obtained elongated triangles as a result due to the creation of rectangles in which the ratio between their length and width is much greater than one. When there are points close to the contour of the unicorn, or that have their X and Y coordinates close, these elongated rectangles are produced, which lead to geometries with a very large maximum angle.

Table 6.5: Main study characteristics of the initial meshes of the unicorn geometry using Centroid refinement.

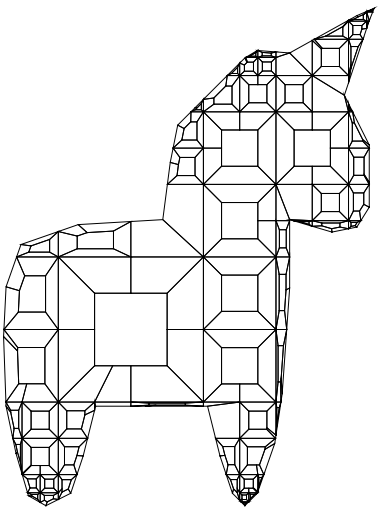
Division Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
Half Point Division	302	204	29.01	32.8	445.64	133.16
Arbitrary Point Division	280	167	48.08	17.13	480.66	63.32
Randomized Arbitrary Point Division	267	158	47.22	20.37	504.06	61.74
KD-Tree	160	102	50.87	22.22	841.15	41.15

Regarding the statistical results obtained, we see that the initial mesh that takes less time to produce is that generated by the KD-tree algorithm, being approximately 3 times less than the time taken by the Half Point algorithm. The KD-tree algorithm produces geometries with a larger area than all other algorithms and with a greater average minimum length. Regarding the minimum angle, the Mid Point algorithm produces the highest value, and regarding the number of elements, the KD-tree algorithm produces approximately half of the elements than the other algorithms. We can see that the KD-tree algorithm produces very good results compared to the other algorithms, however, we must consider that the initial mesh generated by a KD-Tree naturally has fewer elements, so when applying the centroid algorithm, fewer polygons and a shorter time are generated accordingly. A better comparison would be with an initial mesh generated with a KD-Tree with a similar amount of elements to the other strategies, but that is beyond the scope of this work.

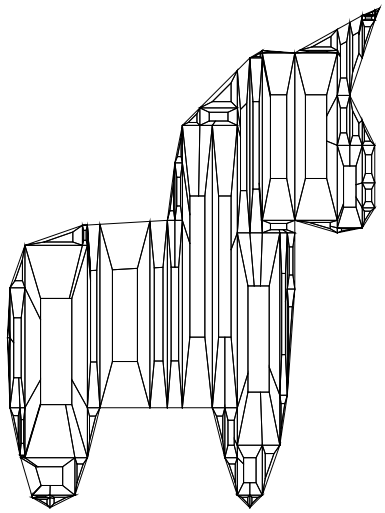
6.3.2.2. Centroid Replication refinement to Initial Meshes

In relation to the replication centroid algorithm explained in section 5.3.4.4 applied to each of the initial meshes, it has the same considerations as the centroid algorithm, that is, those polygons that are non-convex are poligonized to avoid degenerate diagonals. The results are the following.

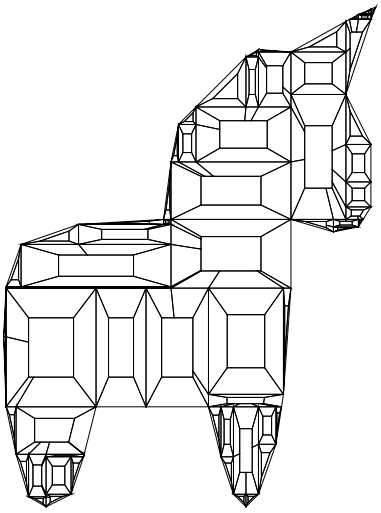
Regarding the shape of the geometries obtained in the meshes as we see in the figure 6.8, the pattern is repeated with respect to the refinement centroid algorithm, since in the divisions using arbitrary points, elongated rectangles appear that lead to the creation of extended triangles and large maximum angles. We think that the generation of elongated elements is due to the same explanation of close coordinate points, since it occurs to a greater extent only in the algorithms of division of arbitrary points. However, a difference between the two methods is that the initial mesh in the random case can be generated from a distribution of points such that there are few cases where the points are close to each other. This explains the reason for the improvement in the polygons obtained in the random insertion compared to the sequential insertion, dividing the quadrants according to the points to insert.



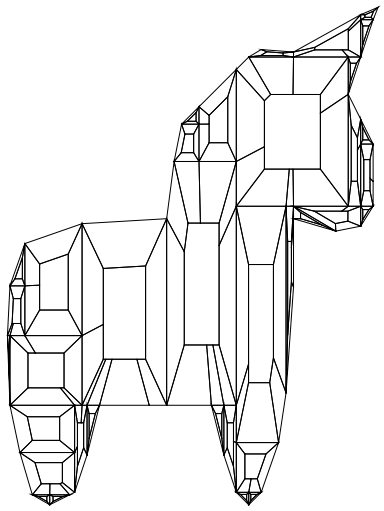
(a) Mid Point Division Initial Mesh with Centroid replication refinement applied.



(b) Arbitrary Point Division Initial Mesh with Centroid replication refinement applied.



(c) Randomized Point Division Initial Mesh with Centroid replication refinement applied.



(d) KD-Tree Initial Mesh with Centroid replication refinement applied.

Figure 6.8: Applying centroid replication refinement to the different initial meshes.

Table 6.6: Main study characteristics of the initial meshes of the unicorn geometry using Centroid Replication refinement.

Division Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
Half Point Division	363	428	18.5	45.42	370.75	620.16
Arbitrary Point Division	346	381	28.62	38.34	388.97	366.34
Randomized Arbitrary Point Division	345	380	29.58	36.81	390.1	305.57
KD-Tree	195	227	30.32	37.84	690.17	162.67

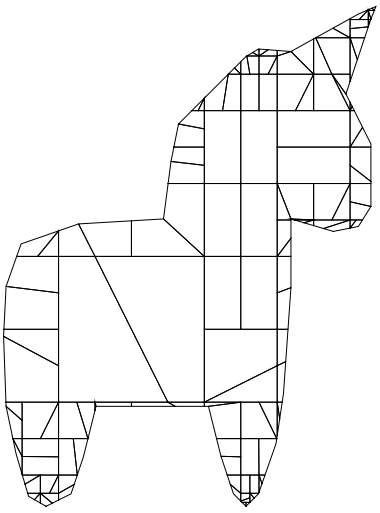
Regarding the computation time of the initial meshes, again KD-Tree is the fastest algorithm of all, however it is the one that produces fewer elements, approximately half of the other algorithms, so it makes fewer cuts between polygons. We reached the same conclusions mentioned above regarding the optimal level of comparison of a KD-Tree mesh with those of a quadtree. Even so, the time used by the algorithm of division by the midpoint takes a great amount of time, being approximately 5 times greater than that used in the KD-Tree and 2 times greater than in those algorithms for inserting arbitrary points. This is because the division by midpoint algorithm produces more intersections of the polygons in the quadrants, so the number of cuts that must be made is much greater, and therefore smaller polygons are produced that are subjected to the algorithm.

6.3.2.3. Splitting Longest Edge refinement to Initial Meshes

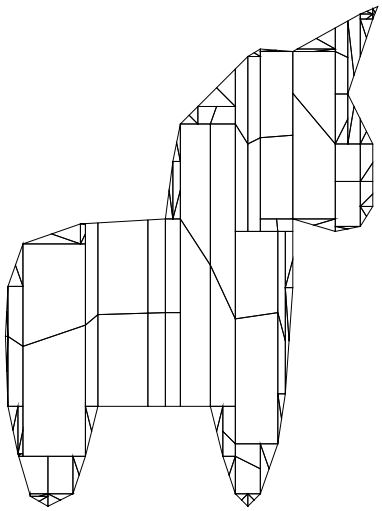
Finally, in the refinement algorithm using splitting longest edge explained in the section 5.3.4.2, the same procedure is done for the generated polygons that are not convex like the two previous refinement algorithms. The results are the following.

We can see in relation to the shape of the generated elements in figure 6.9 that the tendency to have elongated elements in the algorithms of insertion of arbitrary points is maintained, so that regardless of the refinement method, said characteristic is maintained and is inherent to the geometry. It follows then that those geometries that have close coordinate points produce similar results even if the insertion is randomized or not.

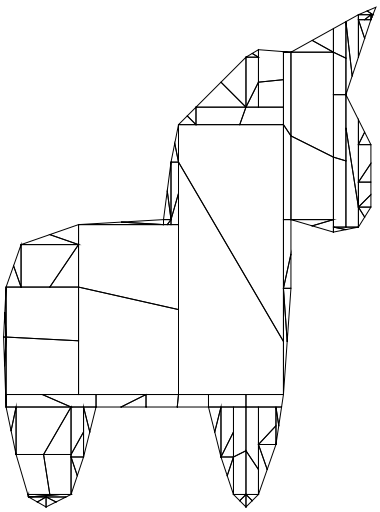
And in relation to the data obtained for different elements, we see that the trend continues in relation to computing time, with the mid-point algorithm being the one that takes the longest time, but the one that produces smaller edges and higher minimum angles. The algorithm that takes the least time is the KD-Tree algorithm, also obtaining the best result in terms of number of elements, achieving the fewest number of polygons and vertices inserted in the initial mesh, but due to the little natural refinement of the algorithm, they produce very large elements, having the highest average area compared to all other algorithms. The same considerations as previously studied are then maintained with respect to an appropriate comparison for the results obtained by a KD-Tree.



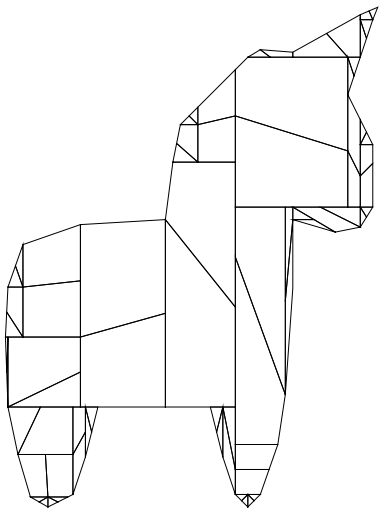
(a) Mid Point Division Initial Mesh with splitting longest edge refinement applied.



(b) Arbitrary Point Division Initial Mesh with splitting longest edge refinement applied.



(c) Randomized Point Division Initial Mesh with splitting longest edge refinement applied.



(d) KD-Tree Initial Mesh with splitting longest edge refinement applied.

Figure 6.9: Applying splitting longest edge refinement to the different initial meshes.

Table 6.7: Main study characteristics of the initial meshes of the unicorn geometry using Splitting Longest Edge refinement.

Division Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
Half Point Division	139	204	26.23	51.59	968.23	705.96
Arbitrary Point Division	132	167	33.51	38.03	1019.57	529.6
Randomized Arbitrary Point Division	130	164	34.2	38.31	1035.26	526.11
KD-Tree	70	102	40.46	40.24	1922.63	149.23

6.4. Successive quality refinements

In this section we show the results obtained after making successive quality refinements on those polygons that do not meet a certain condition. In particular, we study the initial mesh generated using the midpoint algorithm, shown in the figure 6.6 and how this varies as we apply refinements to arrive at a mesh that meets the quality criteria. For the analysis we focus on two usual quality criteria for polygons, marking as "bad" those polygons that exceed a certain area, or that have a longer side at a certain value.

6.4.1. Declaring an upper limit to the area of the polygons

The following analysis consists of applying the three quality refinements to the initial polygon mesh obtained from applying the division using the midpoint on the unicorn geometry. We consider for the experiment the value of the average area of the entire mesh as the upper threshold, consequently all the polygons that have a greater area will be considered as bad polygons.

After that, we consider different fractions of said value, to be more strict and mark more polygons as bad, reaching 1/10 of the average area of the initial mesh as the acceptable area limit.

6.4.2. Upper limit equal to average mesh area

The calculation of the average area of the polygons of the initial mesh gives an approximate 1979 square units of area, which marks 10 polygons that must be refined. Below we show which are the polygons to be refined marked with a red hue, and later, the results after applying each of the different quality refinements.

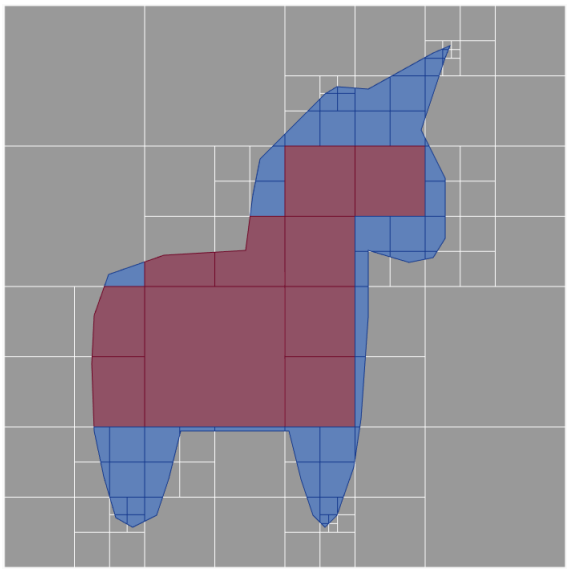
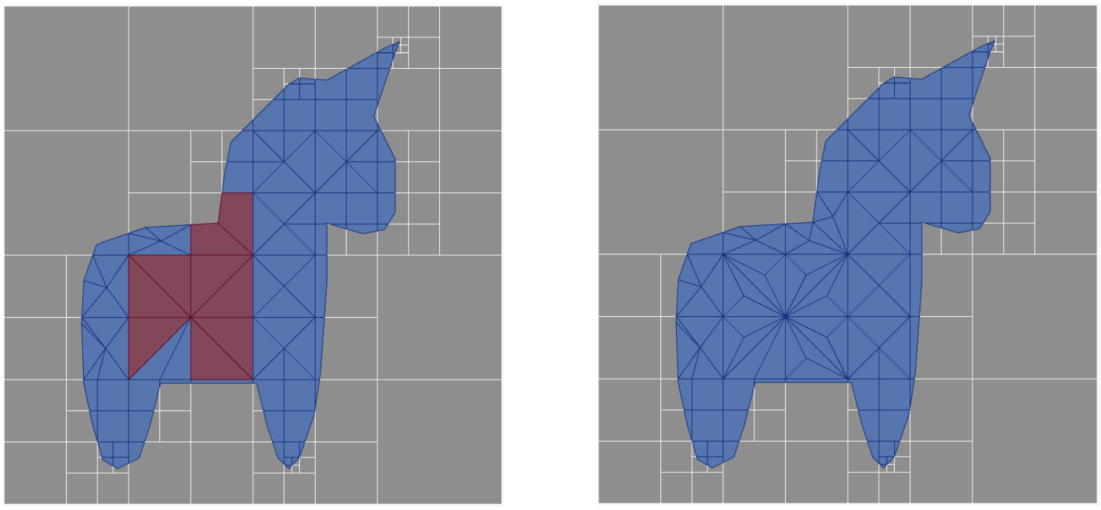


Figure 6.10: Unicorn geometry with bad polygons (Mean area).

6.4.2.1. Results obtained by centroid algorithm



(a) First refinement

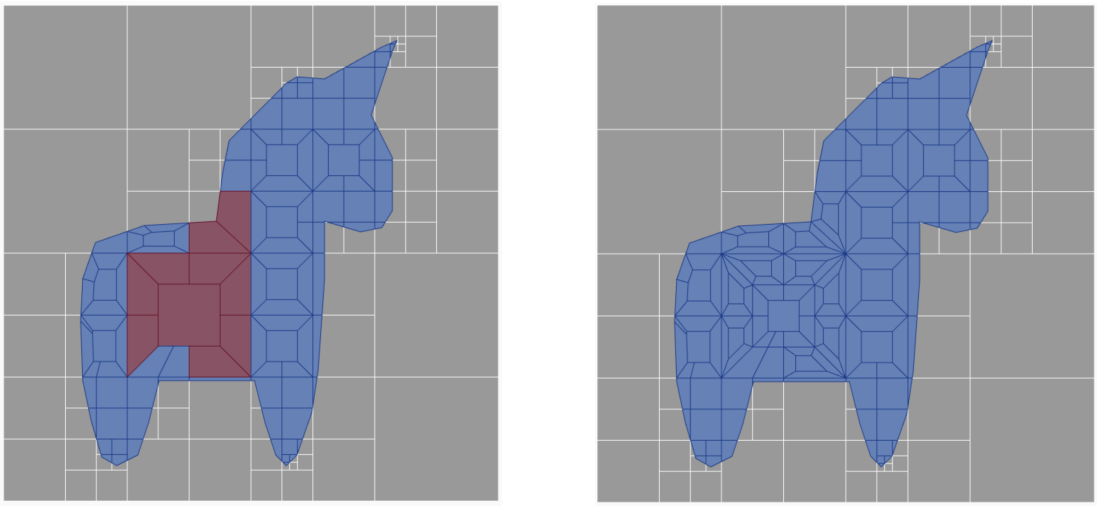
(b) Second refinement

Figure 6.11: Succesive Centroid refinement to bad polygons (Mean area).

Table 6.8: Main study characteristics after successive Centroid refinements to bad polygons (Mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	112	152	38.55	55.57	1201.64	28.01
2	132	161	40.24	50.01	1019.57	17.95

6.4.2.2. Results obtained by centroid replicate algorithm



(a) First refinement

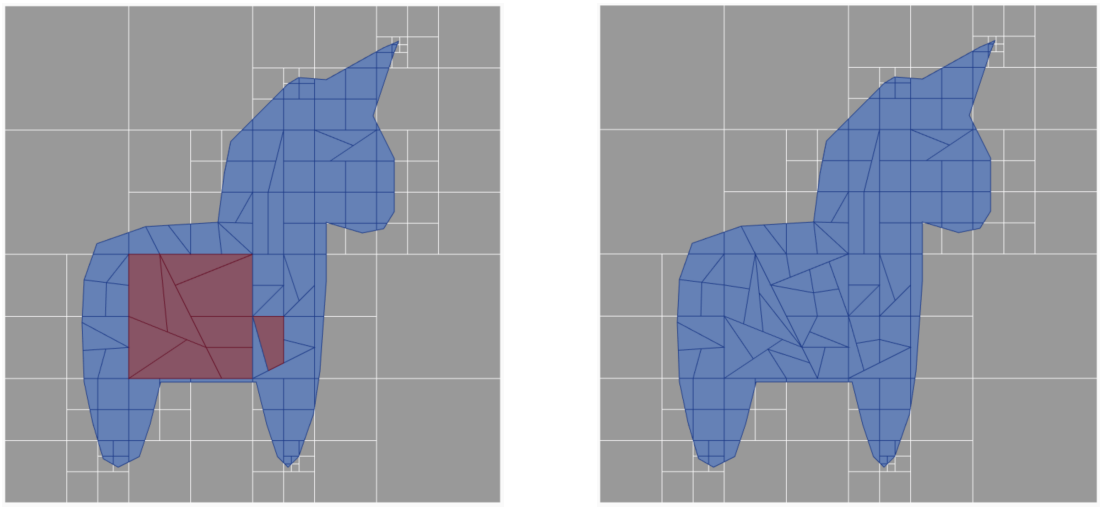
(b) Second refinement

Figure 6.12: Succesive Centroid Replicate refinement to bad polygons (Mean area).

Table 6.9: Main study characteristics after successive Centroid Replicate refinements to bad polygons (Mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	121	195	31.15	59.54	1112.26	74.36
2	166	240	29.51	53.33	810.75	79.9

6.4.2.3. Results obtained by Splitting Longest Edge algorithm



(a) Third refinement

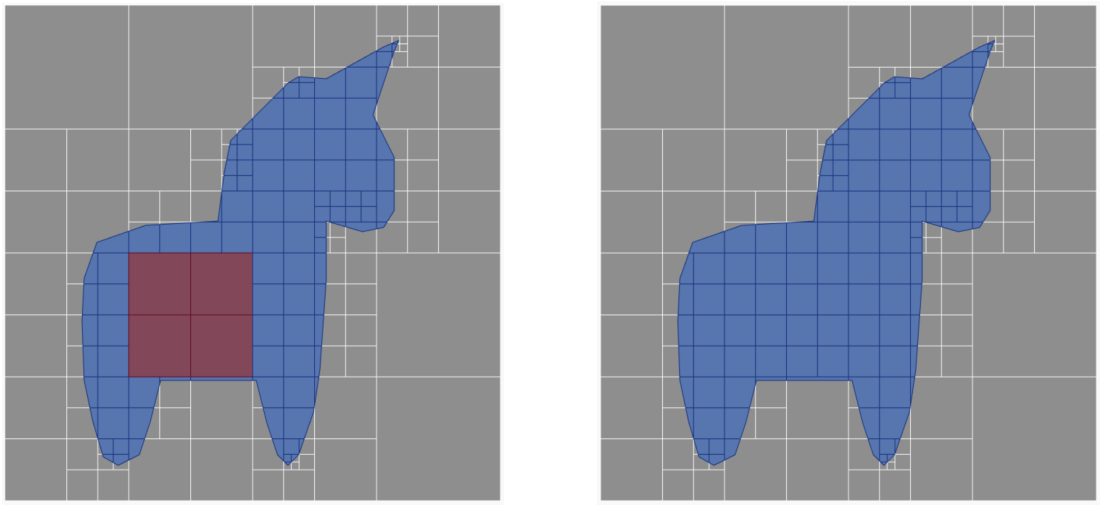
(b) Sixth refinement

Figure 6.13: Succesive Splitting Longest Edge refinement to bad polygons (Mean area).

Table 6.10: Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (Mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	78	152	32.38	69.35	1725.43	53.55
2	95	169	32.49	65.82	1416.67	98.99
3	107	181	32.63	63.57	1257.79	60
4	116	190	32.47	62.55	1160.2	39.39
5	121	195	32.37	61.27	1112.26	17.83
6	122	196	32.26	61.42	1103.15	3.75

6.4.2.4. Results obtained by Quadtree Refining algorithm



(a) First refinement

(b) Second refinement

Figure 6.14: Succesive Quadtree Refinement to bad polygons (Mean area).

Table 6.11: Main study characteristics after successive Quadtree Refinement to bad polygons (Mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	111	194	28.22	77.26	1212.47	505.32
2	123	203	28.61	78.5	1094.18	119.73

6.4.3. Upper limit equal to one tenth of the average area

One tenth of the average area is about 198 square units, thereby marking 51 polygons as bad polygons. Below we show which are the polygons to be refined marked with a red hue, and later, the results after applying each of the different quality refinements.

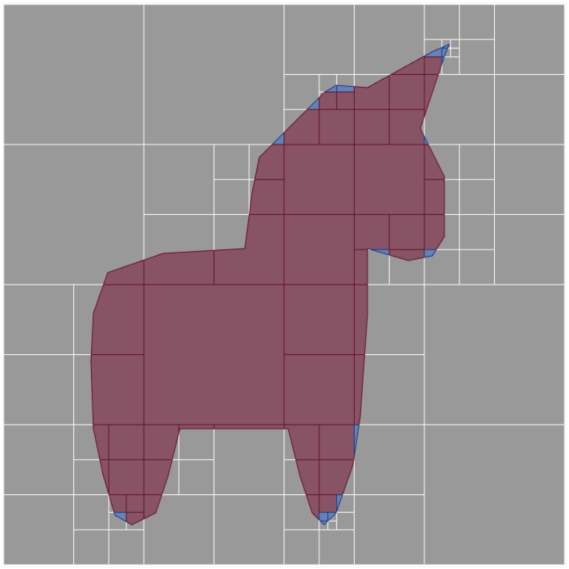
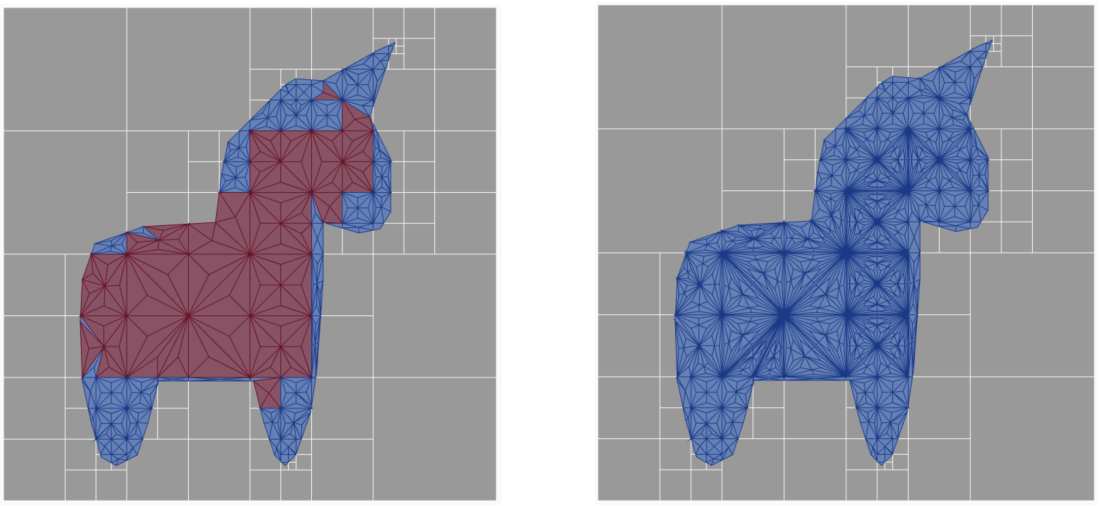


Figure 6.15: Unicorn geometry with bad polygons (One tenth of the mean area).

6.4.3.1. Results obtained by centroid algorithm



(a) Second refinement

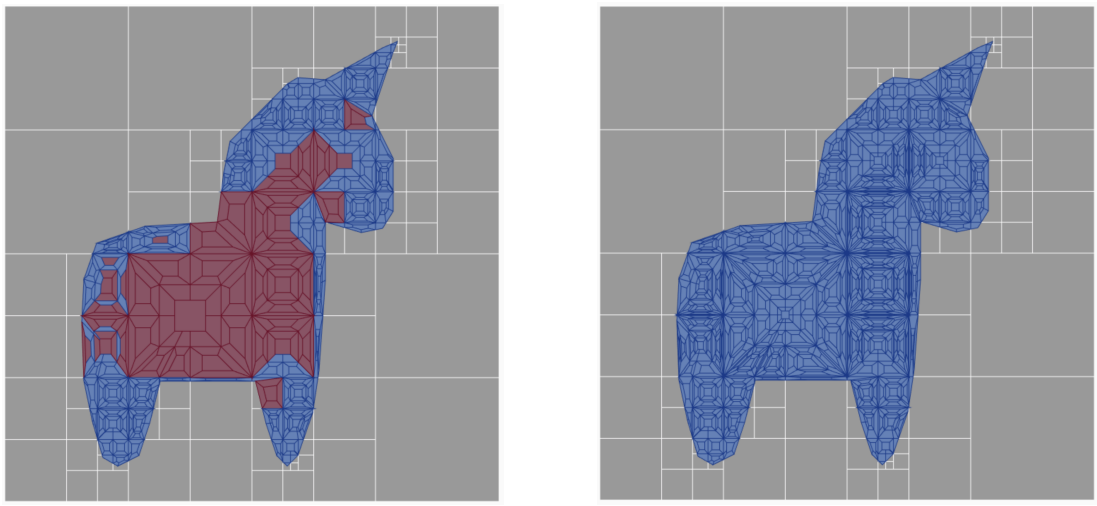
(b) Fourth refinement

Figure 6.16: Succesive Centroid refinement to bad polygons (One tenth of the mean area).

Table 6.12: Main study characteristics after successive Centroid refinements to bad polygons (One tenth of the mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	257	188	32.15	35.76	523.67	108.41
2	530	320	29.2	25.04	253.93	327.75
3	868	489	27.72	20.22	155.05	620.83
4	1300	705	26.14	17.92	103.53	1055.39

6.4.3.2. Results obtained by centroid replicate algorithm



(a) Second refinement

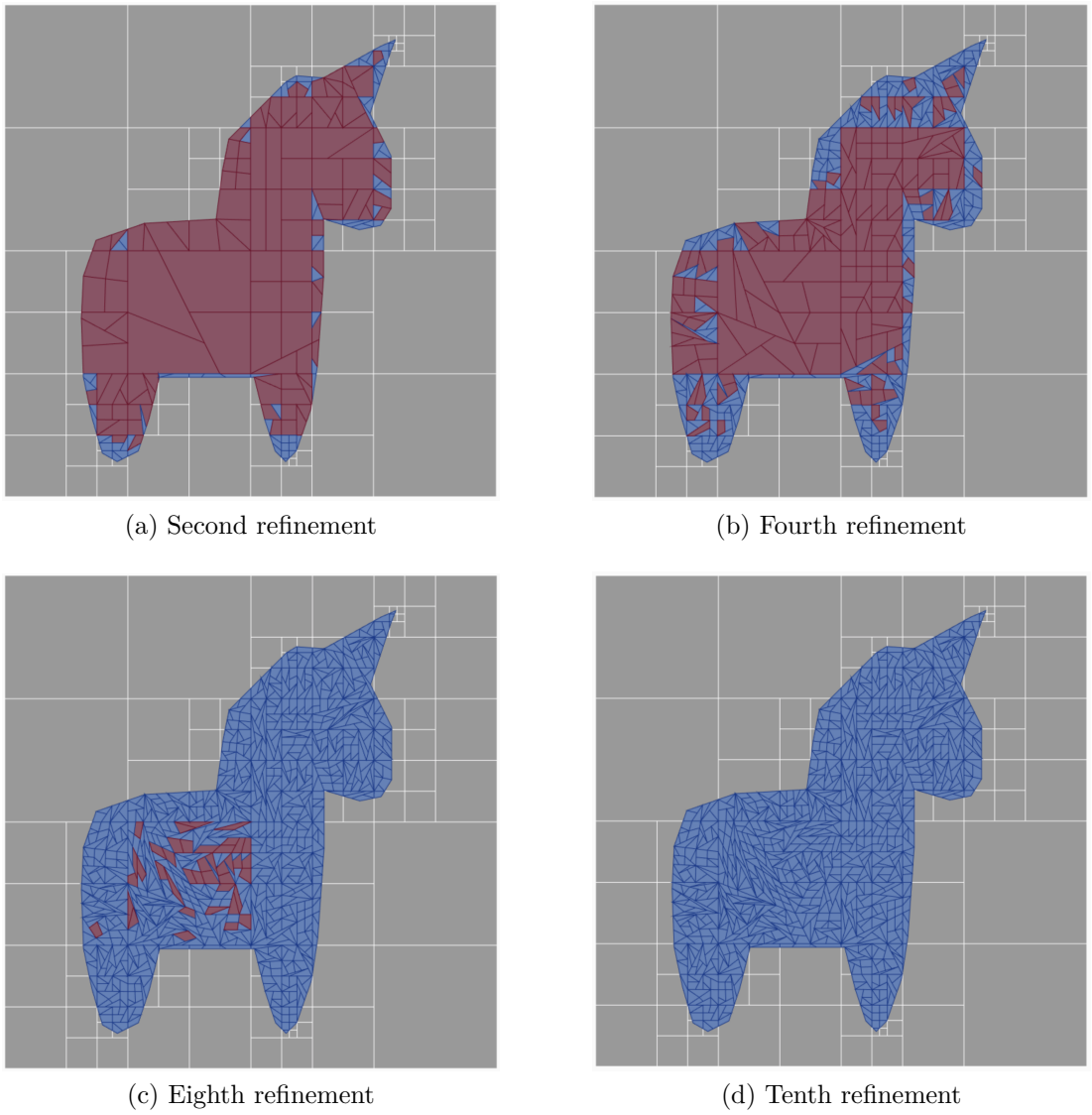
(b) Fourth refinement

Figure 6.17: Succesive Centroid Replicate refinement to bad polygons (One tenth of the mean area).

Table 6.13: Main study characteristics after successive Centroid Replicate refinements to bad polygons (One tenth of the mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	302	368	21.09	48.29	445.64	457.71
2	910	976	15.65	37.85	147.89	2595.05
3	1678	1744	13.64	32.37	80.2	6291.69
4	1687	1753	13.6	32.45	79.78	88.89

6.4.3.3. Results obtained by Splitting Longest Edge algorithm



(a) Second refinement

(b) Fourth refinement

(c) Eighth refinement

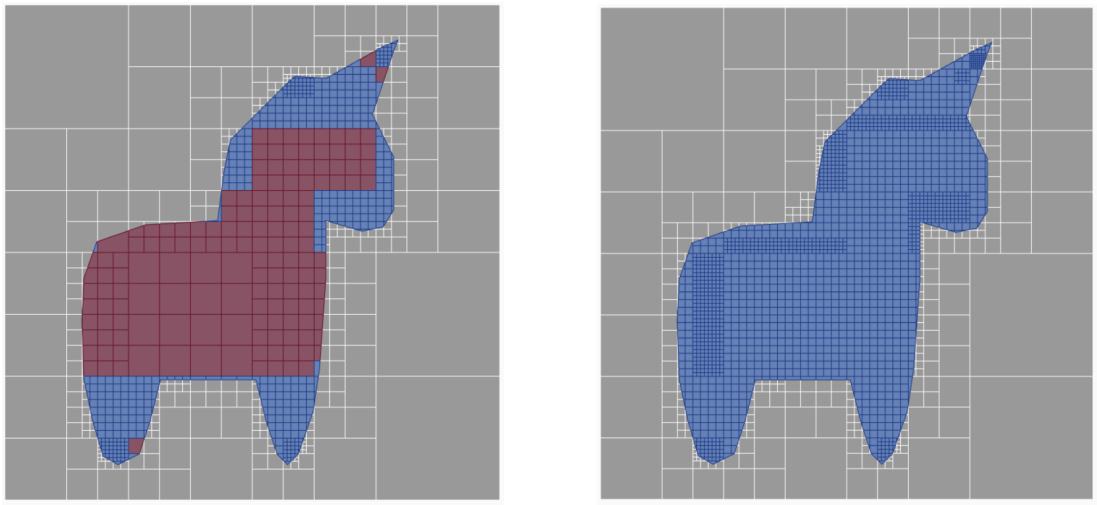
(d) Tenth refinement

Figure 6.18: Succesive Splitting Longest Edge refinement to bad polygons (One tenth of the mean area).

Table 6.14: Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (One tenth of the mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	122	188	28.34	54.45	1103.15	450.68
2	209	275	23.75	49.59	643.94	1252.74
3	330	396	21.06	44.63	407.83	2626.45
4	481	547	18.72	41.92	279.8	4133.18
5	657	723	17.36	38.23	204.85	5602.9
6	806	872	16.36	37.18	166.98	4417.65
7	926	992	15.75	36.26	145.34	3229.33
8	1013	1079	15.29	35.92	132.86	2198.12
9	1053	1119	15.07	35.7	127.81	671.64
10	1056	1122	15.05	35.68	127.45	109.2

6.4.3.4. Results obtained by Quadtree Refining algorithm



(a) Second refinement

(b) Fourth refinement

Figure 6.19: Succesive Quadtree Refining to bad polygons (One tenth of the mean area).

Table 6.15: Main study characteristics after successive Quadtree Refining to bad polygons (One tenth of the mean area).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	202	312	19.83	79.27	666.33	2132.99
2	667	848	11.58	83.7	201.8	10063.5
3	1477	1718	8.43	86.54	91.13	26918.14
4	1963	2230	7.58	87.39	68.57	25736.54

6.4.4. Declaring an upper limit to the maximum edge length of the polygons

Another common type of analysis is to perform quality refinements until all the polygons in a given polygon mesh have no side longer than a certain value, provided by the user. Later, we perform the analysis for half of that value, with the corresponding algorithms.

Something important to note is that the Centroid and Centroid with Replication refinement algorithms do not converge to a solution, because in each refinement, they maintain the shape of each initial polygon, and in no case is the length of the segments of the elements shortened. Therefore, the only analyzed methods are Splitting longest edge and Quadtree Refining.

6.4.5. Upper limit equal to average edge length

The calculation of the average edge length of the polygons of the initial mesh gives an approximate 30 units, which marks 43 polygons that must be refined. Below we show which are the polygons to be refined marked with a red hue, and later, the results after applying each of the different quality refinements.

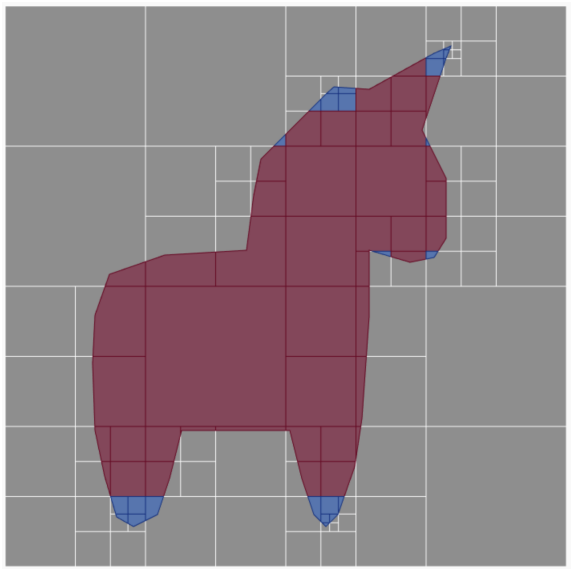
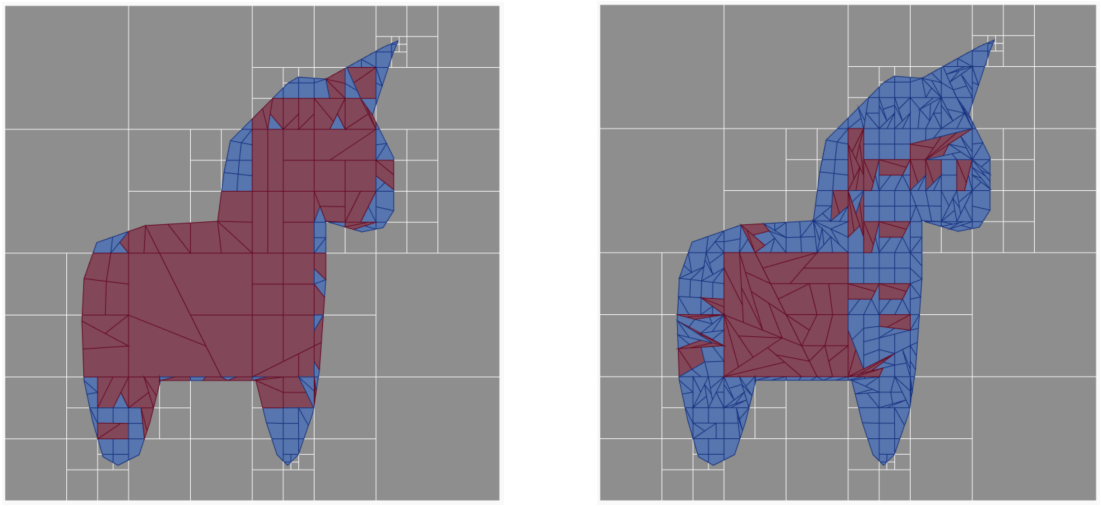


Figure 6.20: Unicorn geometry with bad polygons (Average edge length).

6.4.5.1. Results obtained by Splitting Longest Edge algorithm



(a) Second refinement

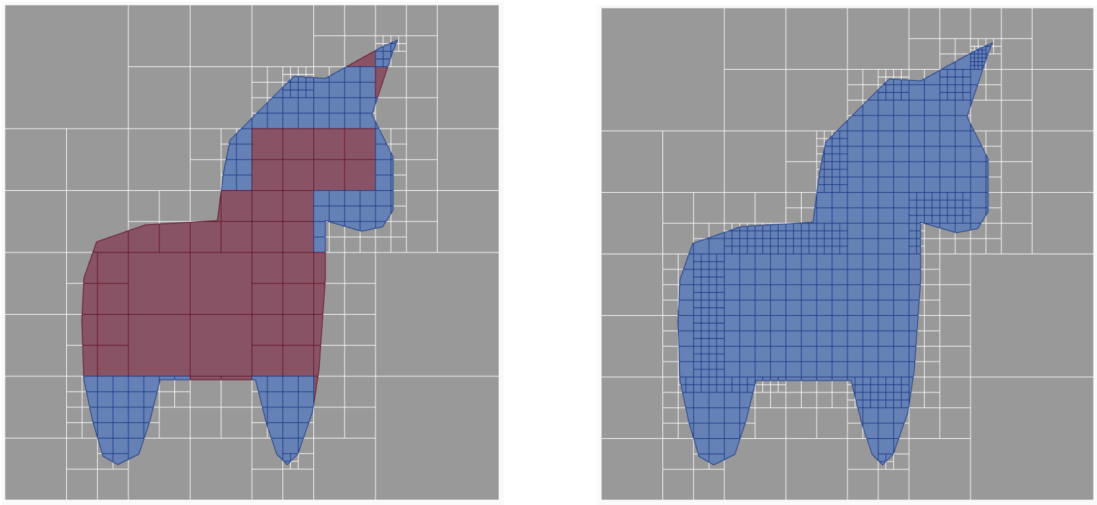
(b) Fifth refinement

Figure 6.21: Succesive Splitting Longest Edge refinement to bad polygons (Average edge length).

Table 6.16: Main study characteristics after successive Splitting Longest Edge refinements to bad polygons (Average edge length).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	114	180	29.32	56.15	1180.56	310.76
2	187	253	25.25	51.16	719.7	976.96
3	281	347	22.8	46.26	478.95	1595.22
4	385	451	20.81	44.88	349.57	2190.13
5	475	541	20.07	41.48	283.33	1870.07
6	565	631	19.07	39.64	238.2	1999.23
7	636	702	18.38	37.98	211.61	1387.24
8	687	753	17.92	36.82	195.9	873.35
9	725	791	17.49	35.68	185.63	595.98
10	735	801	17.37	35.31	183.11	106.34
11	739	805	17.33	35.14	182.12	39.86

6.4.5.2. Results obtained by Quadtree Refining algorithm



(a) First refinement

(b) Third refinement

Figure 6.22: Succesive Quadtree Refining to bad polygons (Average edge length).

Table 6.17: Main study characteristics after Quadtree Refining to bad polygons (Average edge length).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	181	284	21.55	79.26	743.64	1629.77
2	419	563	15.14	83.76	321.24	4054.19
3	566	720	13.7	85.1	237.81	2790.32

6.4.6. Upper limit equal to one half of the average edge length

One half of the average edge length of the polygons is about 15 units, which marks 59 polygons that must be refined. Below we show the polygons to be refined marked with a red hue, and later, the results after applying each of the different quality refinements.

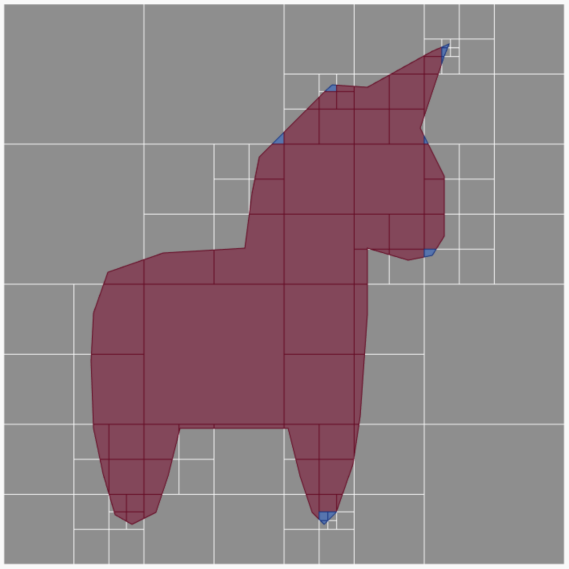
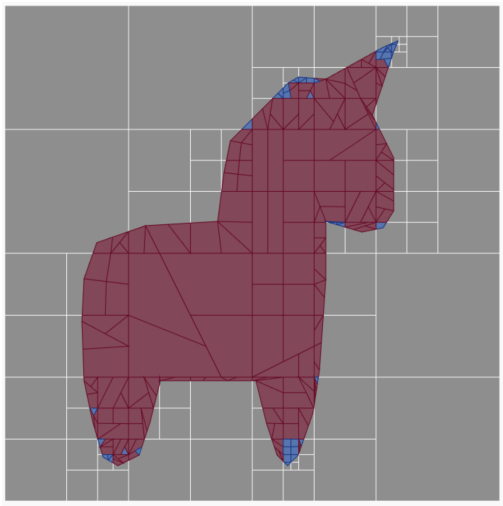
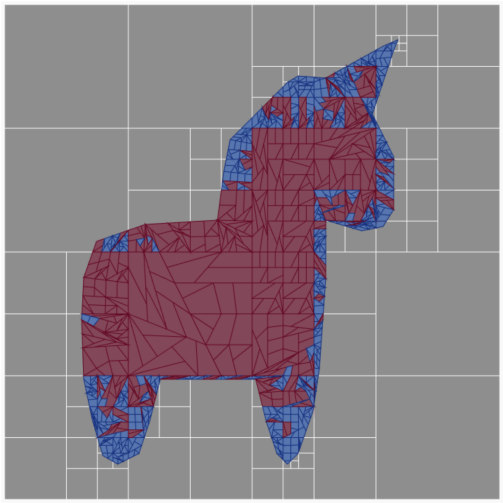


Figure 6.23: Unicorn geometry with bad polygons (One half of average edge length).

6.4.6.1. Results obtained by Splitting Longest Edge algorithm



(a) Second refinement



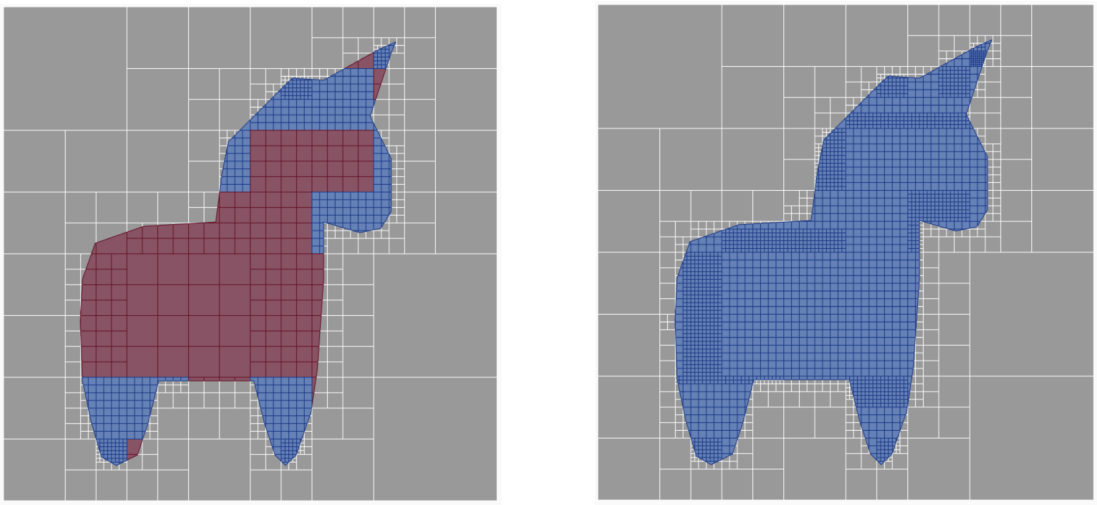
(b) Fifth refinement

Figure 6.24: Succesive Splitting Longest Edge refinement to bad polygons (One half of the average edge length).

Table 6.18: Main study characteristics after successive of Splitting Longest Edge refinements to bad polygons (One half of the average edge length).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	130	196	27.4	53.02	1035.26	564.3
2	242	308	21.69	47.56	556.13	1895.42
3	438	504	17.79	41.45	307.27	5564.14
4	748	814	14.66	38.17	179.92	15208.71
5	1173	1239	12.8	34.55	114.73	31600.35
6	1687	1753	11.45	32.78	79.78	48701.09
7	2299	2365	10.43	30.37	58.54	73139.26
8	2903	2969	9.66	28.97	46.36	78343.87
9	3391	3457	9.17	27.54	39.69	58845.49
10	3792	3858	8.83	26.37	35.49	46852.58
11	4052	4118	8.59	25.47	33.21	24088.05
12	4189	4255	8.48	24.85	32.13	9657.58
13	4277	4343	8.4	24.4	31.47	5115.46
14	4321	4387	8.38	24.16	31.15	2291.16
15	4357	4423	8.37	23.96	30.89	1757.97
16	4394	4460	8.34	23.76	30.63	1791.38
17	4418	4484	8.32	23.63	30.46	1148.22
18	4434	4500	8.31	23.55	30.35	748.79
19	4445	4511	8.3	23.49	30.28	531.41

6.4.6.2. Results obtained by Quadtree Refining algorithm



(a) Second refinement

(b) Fourth refinement

Figure 6.25: Succesive Quadtree Refining to bad polygons (One half of the Average edge length).

Table 6.19: Main study characteristics after successive Quadtree Refining to bad polygons (One half of the average edge length).

Refinement Level	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
1	202	312	19.83	79.27	666.33	2077.83
2	670	851	11.55	83.73	200.89	10342.6
3	1650	1926	7.86	86.72	81.58	33974.16
4	2234	2529	7.07	87.53	60.25	28500.06

6.5. Comparison of quality metrics between different levels of refinement

In this section we study the quality metrics on the geometric meshes of the refinements made previously. The analysis was carried out on the geometric meshes that fulfilled the condition of Area and Maximum length, after multiple refinements using the different algorithms. Our purpose is to obtain the best algorithm according to the metrics, and thus compare it with Triangle.

Remember that the metrics are independent of the polygon scale, and they're defined in a certain range and with a certain trend. For the purposes of a better analysis we summarize this information below:

- The *Circle Ratio (CR)* metric is defined in a range of $[0, 1]$ and the greater and closer to 1 is, the better.
- The *Edge Ratio (ER)* metric is defined in a range of $(0, 1]$ and the greater and closer to 1 is, the better.
- The *Normalized Point Distance (NPD)* metric is defined in a range of $(0, 1]$ and the greater and closer to 1 is, the better.
- The *Perimeter Area Ratio (PAR)* metric is defined in a range of $(0, \infty)$ and the smaller and closer to 0, the better.

6.5.1. Results of imposing a upper limit to the maximum area

6.5.1.1. Limit equal to the average area of the geometric mesh

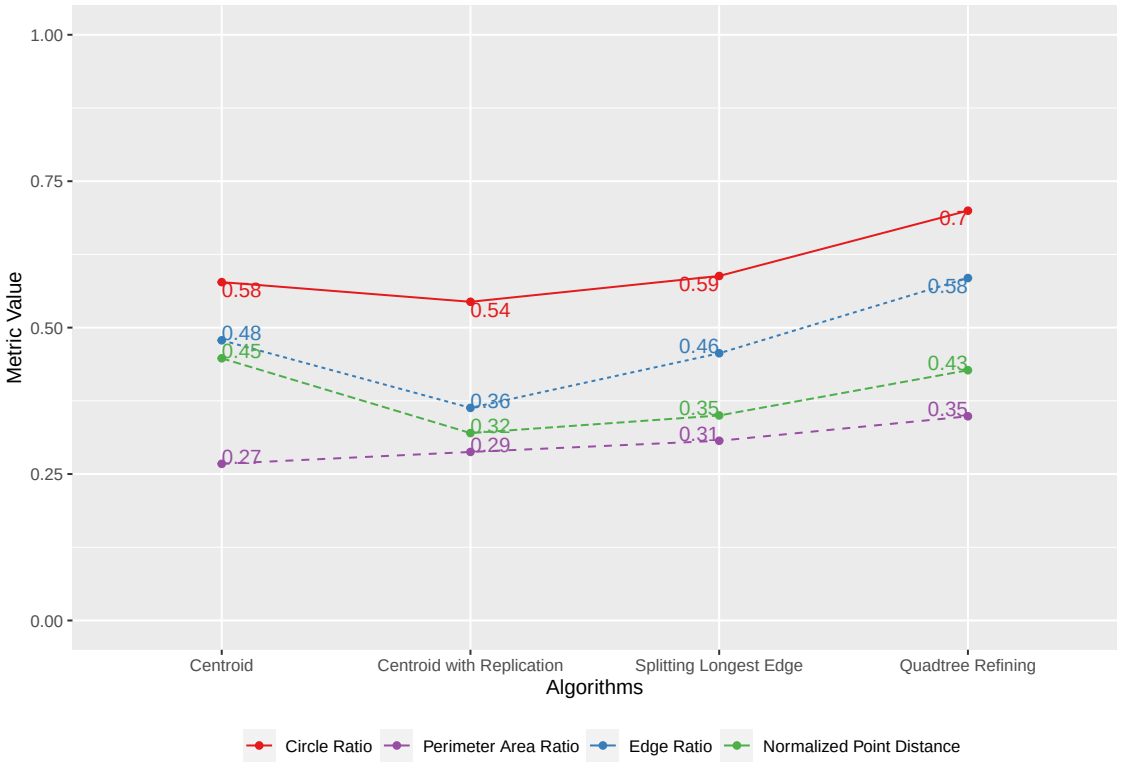


Figure 6.26: Metrics for Average Area.

6.5.1.2. Limit equal to $\frac{1}{10}$ of the average area of the geometric mesh

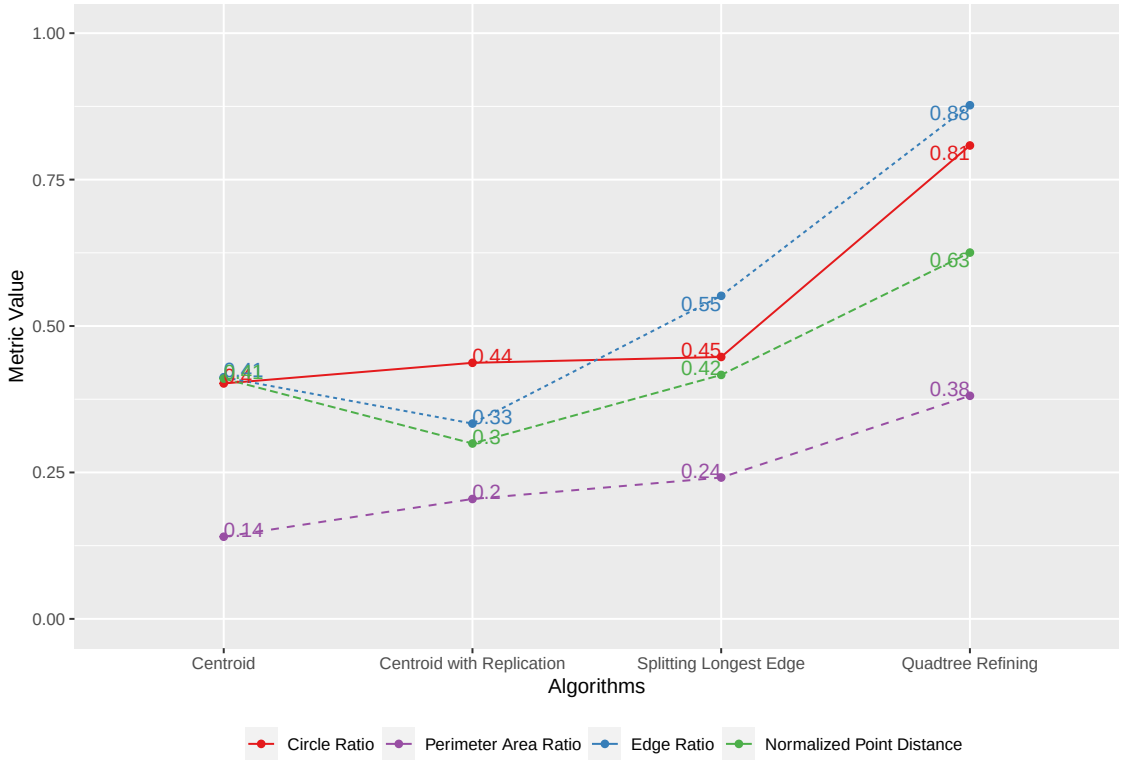


Figure 6.27: Metrics for one tenth of the Average Area.

6.5.1.3. Analysis of metric results

The analysis of the results obtained, and the comparison between them for each of the metrics, is as follows:

1. **Circle Ratio (CR)**: The trend in this measure is downward between both limit scenarios, with the exception of the Quadtree Refining algorithm, which goes from 0.7 to 0.84, being the only one that has an upward trend.

2. **Edge Ratio (ER)**: In this case, there are two refinement algorithms that increase their value when going from one scenario to the other, which are Longest Splitting Edge and Quadtree Refining. The biggest change between the two is the Quadtree Refining algorithm, going from 0.58 to 0.88, so we conclude that this is the best algorithm for this metric.

3. **Normalized Point Distance (NPD)**: In this metric, we again attribute Quadtree Refining as the best algorithm, going from 0.43 to 0.63, thus obtaining better results in deeper refinements.

4. **Perimeter Area Ratio (PAR)**: Finally, in this metric that has the best result the closer it is to 0, the best algorithm is the Centroid Algorithm, falling from 0.27 to 0.14. In this case, it should be noted that the Centroid Refining algorithm did not perform well, even maintaining an upward trend, which implies that it obtained a worse result.

6.5.2. Results of imposing a upper limit to the maximum length

It is important to remember that in relation to getting shorter and shorter lengths, the only refinement strategies that converge to such a mesh are Splitting Longest Edge and Quadtree Refining. The analysis of the results obtained, and the comparison between them for each of the metrics, is as follows:

6.5.2.1. Limit equal to the average length of the geometric mesh

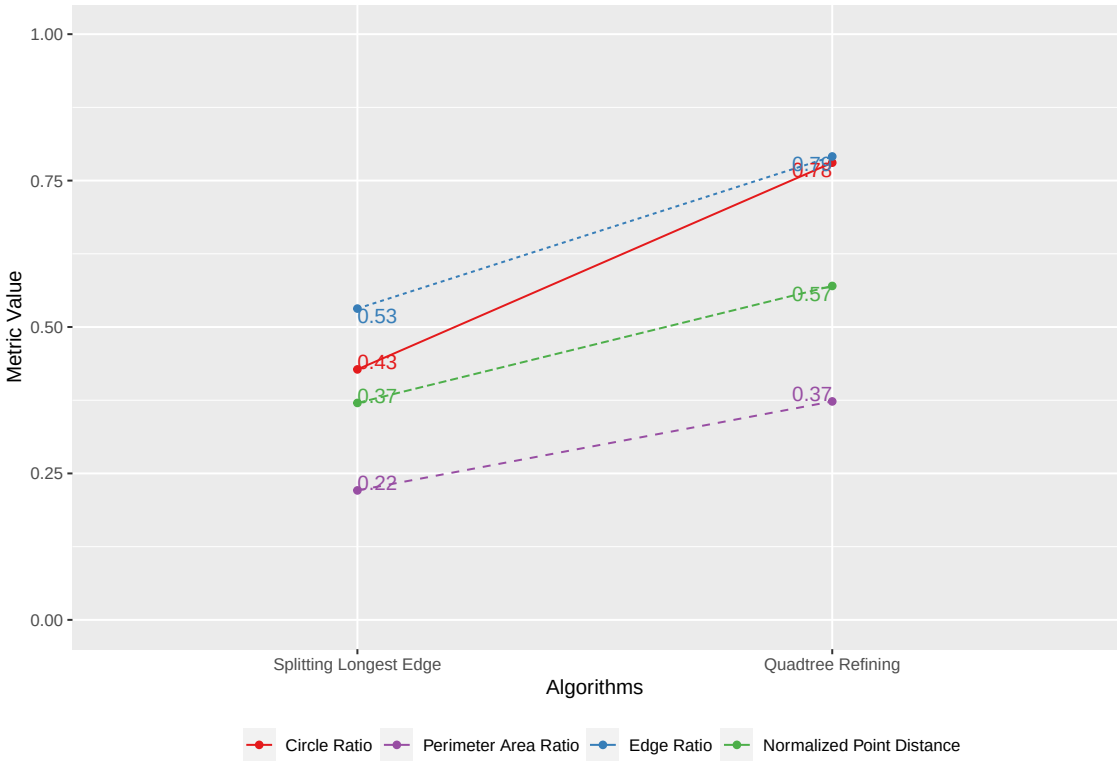


Figure 6.28: Metrics for Average Length.

6.5.2.2. Limit equal to $\frac{1}{2}$ of the average length of the geometric mesh

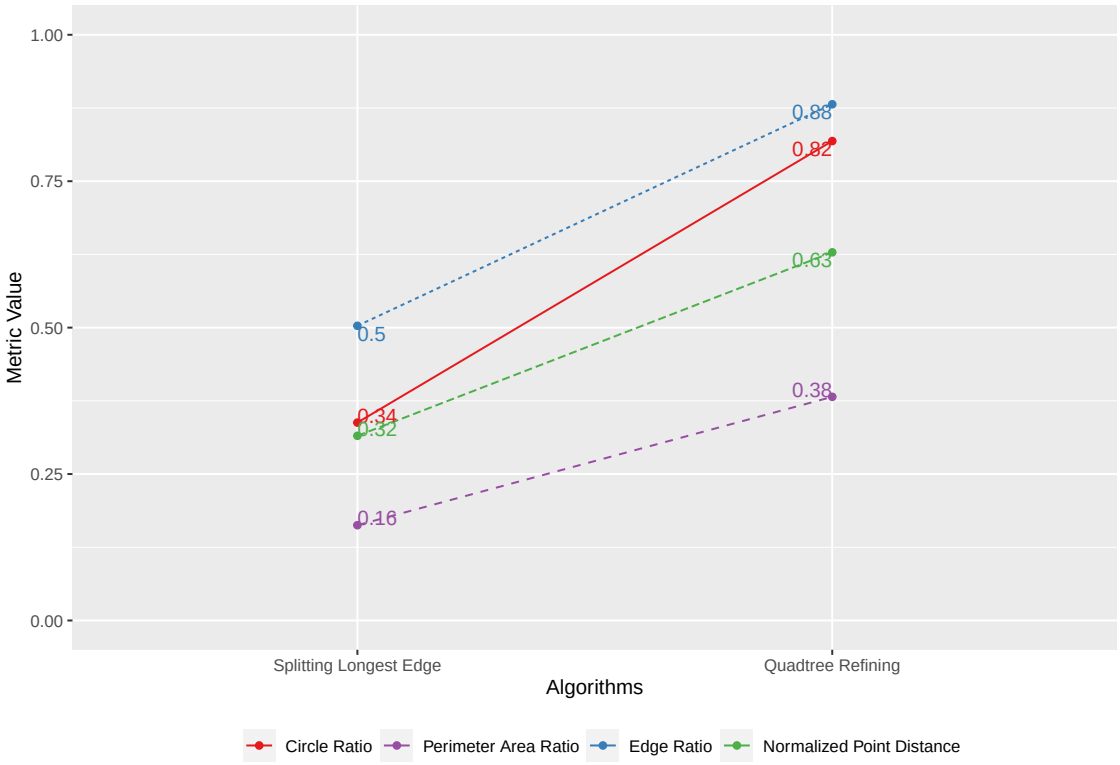


Figure 6.29: Metrics for one half of Average Length.

6.5.2.3. Analysis of metric results

1. **Circle Ratio (CR)**: For this metric, the best result was obtained by the Quadtree Refining algorithm, going from 0.78 to 0.82, maintaining the upward trend.

2. **Edge Ratio (ER)**: Like the previous metric, the best result was obtained by the Quadtree Refining algorithm, going from 0.79 to 0.88.

3. **Normalized Point Distance (NPD)**: Again the best result was obtained with the Quadtree Refining algorithm, going from 0.57 to 0.63.

4. **Perimeter Area Ratio (PAR)**: In this metric, the Quadtree Refining algorithm remained relatively constant, with the Splitting Longest Edge algorithm obtaining a better result, dropping from 0.22 to 0.16.

6.6. Comparing meshes with Triangle

We now present the comparisons between the results obtained by our application and those obtained by Triangle. We compare using the initial mesh obtained by a quadtree with division using mid point, and the initial mesh generated by kd-tree, as shown in the Figure 6.6 with two strategies: Quadtree Refining and Splitting Longest Edge, until a certain criterion

is fulfilled. First we focus on the main elements of the mesh, such as the number of Polygons and average length of Edges and then we compare the quality metrics in each scenario.

6.6.1. Comparing meshes

Below we show the results obtained from comparing the initial meshes, and their subsequent refining until meeting the condition imposed by the user in reference to the maximum possible value of area and length of the polygons. Something important to note is that the time is not comparable between Triangle and our application, because the results shown include the rendering time and frame changes, so a more accurate time analysis considers only the formation time of the mesh, leaving this as future work.

6.6.1.1. Maximum area equal to 1979 area units

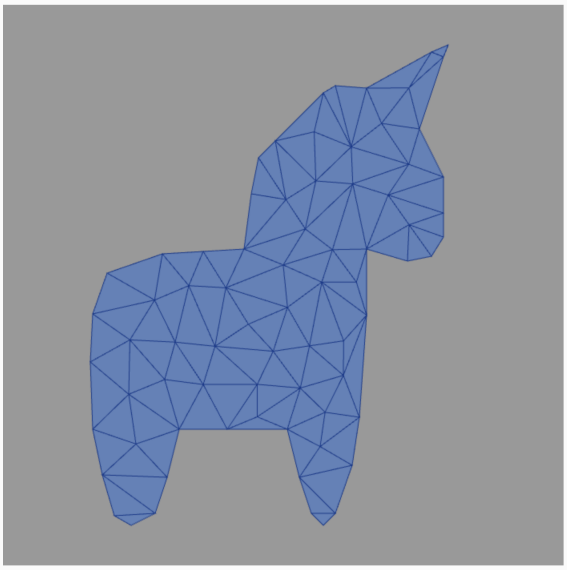


Figure 6.30: Unicorn geometry refined by Triangle (Max: 1979 area units).

The results obtained show that in number of polygons and points, Triangle obtains better performance than any of our algorithms, the closest being using kd-trees with the splitting longest edge strategy. Regarding the minimum length, Triangle gets a higher value, followed by KD-Tree with splitting longest edge, which indicates that Triangle forms bigger triangles with respect to our meshes. Finally, with respect to the average minimum angle, our algorithms generate higher values than Triangle, in particular the Quadtree strategies maintain a higher minimum angle value due to the natural incorporation of quadrilaterals in refinements.

Table 6.20: Comparing main study characteristics with Triangle (Max: 1979 area units).

	Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
KD-Tree	Quadtree Refining	222	333	23.75	73.41	606.3	347.25
	Splitting Longest Edge	119	151	39.73	43.6	1131.08	326.23
Quadtree with mid point strategy	Quadtree Refining	123	203	28.61	78.5	1094.18	119.73
	Splitting Longest Edge	122	196	32.26	61.42	1103.15	3.75
Triangle	-	108	74	56.88	41.58	1246.15	2

However, there are scenarios where it is not necessary to refine the entire mesh, but only a certain region is of interest. For example, if we want to perform a simulation with mathematical methods on a polygon mesh, it is usual that the region of interest is a small part of the entire geometry, so we are interested in obtaining the quality criterion by refining only in that area.

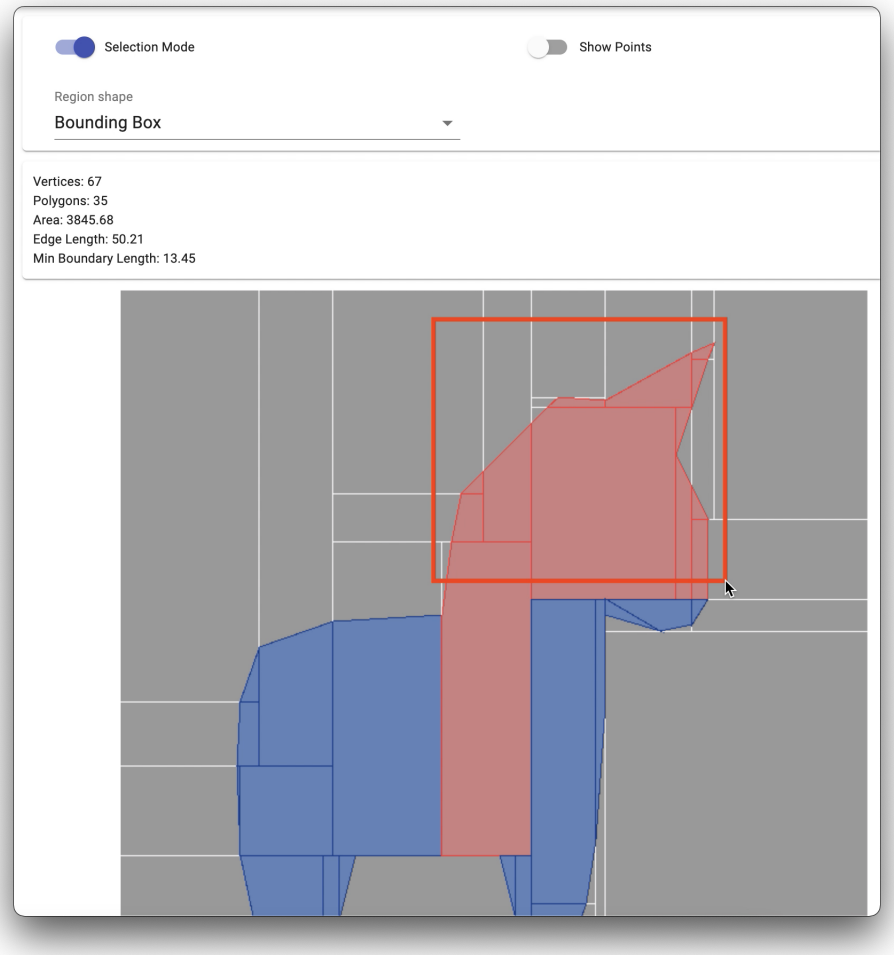


Figure 6.31: Unicorn region to be refined (Max: 1979 area units).

We take as an example the region of interest shown in the Figure 6.31. What we did was to use as a criterion a maximum of 1972 area units for all polygons within that area, using as initial mesh one generated by kdtree, and as refinement algorithm the splitting longest edge.

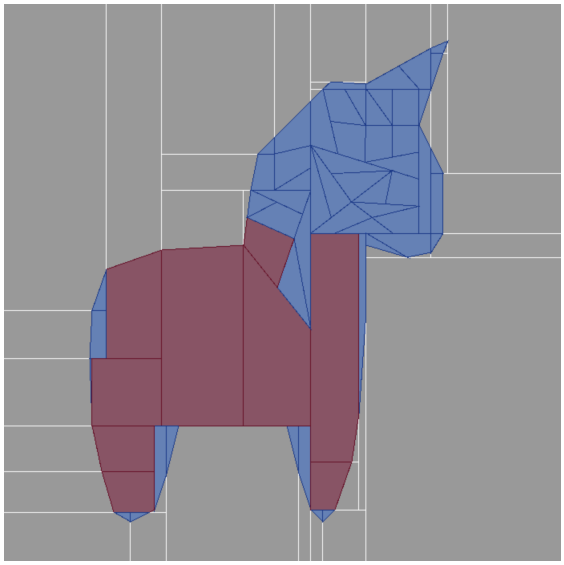


Figure 6.32: Unicorn region refined (Max: 1979 area units).

As shown in Figure 6.32, there are still polygons in the mesh that do not meet the quality criteria, however they are not considered when refining the polygons that belong within the region of interest. In particular we can see that the region demarcated by the user has a deeper level of refinement than the rest of the polygon mesh.

The results obtained are shown in Table 6.21. Because Triangle is not able to refine a certain region, it performs a refinement on the whole mesh to reach the imposed area criterion. On the other hand, because our application is able to refine only the polygons that do not meet the criterion, our refined mesh has 63 polygons, compared to 108 polygons in the mesh generated by Triangle. In relation to the number of points, Triangle has a better optimization at the time of constructing the mesh, obtaining a smaller number of points than our application. In relation to the construction time, similar results were obtained, being our application 0.4 ms faster than Triangle.

Table 6.21: Comparison of refinement by region between our application using a (kdtree and splitting longest edge) and triangle. (Criterion: maximum area 1979 units).

Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
KDTree - Splitting Longest Edge	63	95	45.07	43.83	2136.49	1.6
Triangle	108	74	56.88	41.58	1246.15	2

6.6.1.2. Maximum area equal to 198 area units

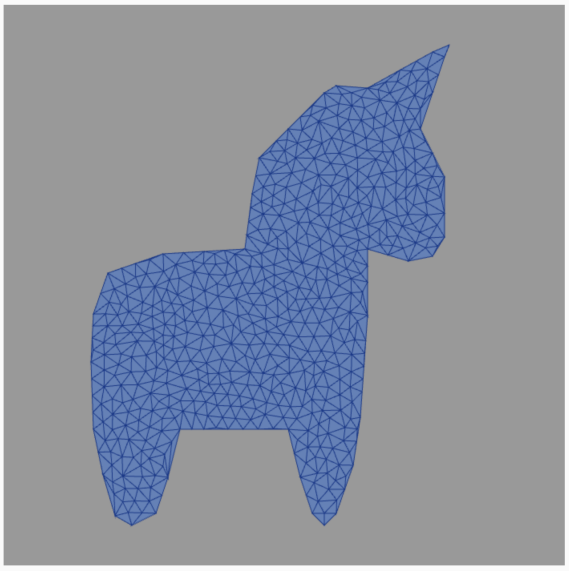


Figure 6.33: Unicorn geometry refined by Triangle (Max: 198 area units).

The results when the refinement is stricter, show that the number of polygons in the trees that used splitting longest edge is less than that generated by Triangle. However, the number of points needed to generate the mesh, practically double. We attribute this to the existence of collinear points that are formed in the refinement of splitting longest edge in the neighboring polygon. In relation to the minimum length, similar results are obtained between Triangle and the longest edge splitting strategy in each tree, and finally in relation to the average minimum angle of the mesh, Triangle obtains a better result. We also note that the trend to an average close to 90^{o} for refinement per quadtree is maintained, due to the generation of quadrilaterals.

Table 6.22: Comparing main study characteristics with Triangle (Max: 198 area units).

	Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
KD-Tree	Quadtree Refining	3304	3817	6.75	85.6	40.74	329.58
	Splitting Longest Edge	1032	1064	17.16	32.31	130.43	325.44
Quadtree with mid point strategy	Quadtree Refining	1963	2230	7.58	87.39	68.57	25736.54
	Splitting Longest Edge	1056	1122	15.05	35.68	127.45	109.2
Triangle	-	1069	573	17.94	44.67	125.9	4

6.6.1.3. Maximum length equal 30 length units

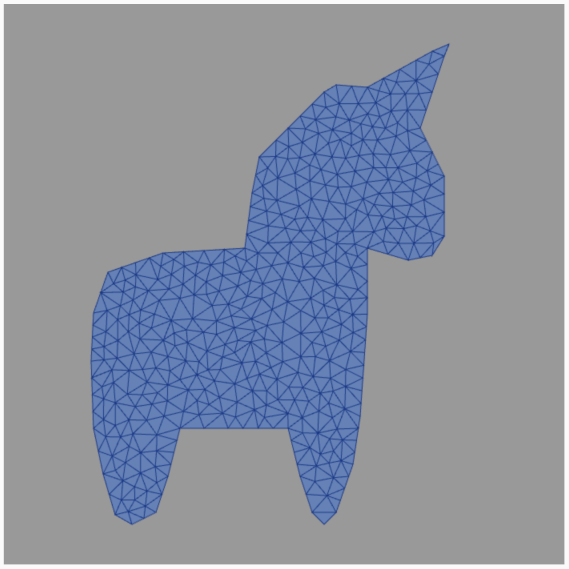


Figure 6.34: Unicorn geometry refined by Triangle (Max: 30 length units).

Now the criterion is applied to the maximum length of an Edge that a polygon can have. When we impose a threshold of 30 units of length, we see that the refinement by quadtree obtains a smaller amount of polygons in both strategies with respect to Triangle, however, it uses a greater amount of points for this. A particular case is the poor performance of KD-tree, obtaining a large number of elements to meet the quality criteria. In relation to the areas, we see that the polygons generated by the quadtree strategy have a higher value than those obtained by Triangle.

Table 6.23: Comparing main study characteristics with Triangle (Max: 30 length units).

	Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
KD-Tree	Quadtree Refining	3295	3804	6.76	85.66	40.85	342.99
	Splitting Longest Edge	1119	1151	16.76	23.49	120.28	360.18
Quadtree with mid point strategy	Quadtree Refining	566	720	13.7	85.1	237.81	2790.32
	Splitting Longest Edge	739	805	17.33	35.14	182.12	39.86
Triangle	-	781	442	20.46	47.09	172.32	2

6.6.1.4. Maximum length equal 15 length units

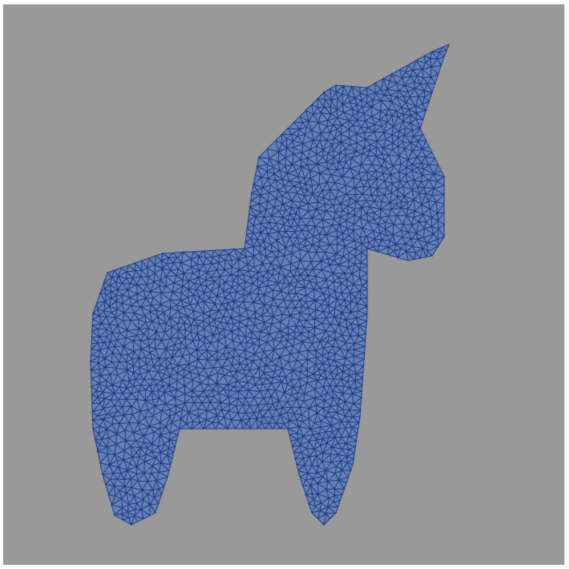


Figure 6.35: Unicorn geometry refined by Triangle (Max: 15 length units).

If we make the criterion more strict imposing a maximum length of 15 units, then we see that the refinement by Quadtree obtains fewer Polygons in its iterative strategy of continuing to partition according to the midpoint of each quadrant in relation to Triangle. We also see that the performance of splitting longest edge using Quadtrees drops, being surpassed by Triangle. Another important aspect is that Triangle occupies a smaller number of points in all cases, practically half compared to the best refinement that follows, which is Quadtree.

Table 6.24: Comparing main study characteristics with Triangle (Max: 15 length units).

	Algorithm	$N_{polygons}$	N_{points}	Min Length$_{mean}$	Angle$_{min}$	Area$_{mean}$	Time (ms)
KD-Tree	Quadtree Refining	9601	10513	4.19	87.65	14.02	364.56
	Splitting Longest Edge	5922	5954	8.21	17.54	22.73	342.12
Quadtree with mid point strategy	Quadtree Refining	2234	2529	7.07	87.53	60.25	28500.06
	Splitting Longest Edge	4445	4511	8.3	23.49	30.28	531.41
Triangle	-	3106	1655	10.25	47.37	43.33	9

6.6.2. Comparison in quality metrics

6.6.2.1. Initial Mesh: Quadtree with Mid Point strategy - Maximum area equal to 198 area units

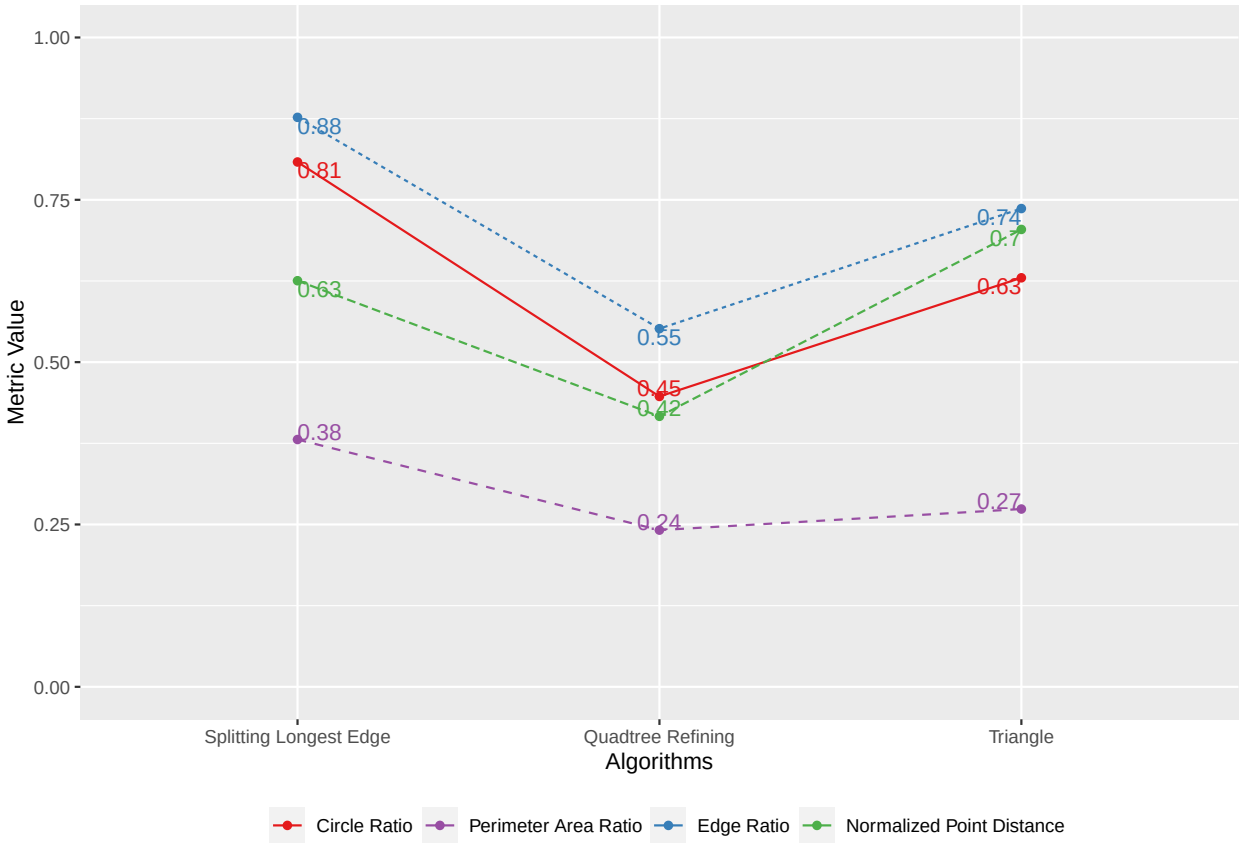


Figure 6.36: Initial Mesh: Quadtree - Quality metric comparison with Triangle (Max: 198 area units).

1. **Circle Ratio (CR)**: In this metric, triangle obtained 0.63, while the Splitting longest edge algorithm obtained a better metric with a value of 0.81. The quadtree refining algorithm obtained 0.45, a lower value than the other two methods.

2. **Edge Ratio (ER)**: Continuing with the previous behavior, the refinement of Splitting Longest Edge obtained 0.88, while Triangle obtained 0.74.

3. **Normalized Point Distance (NPD)**: In this metric Triangle obtained 0.7 being the algorithm that had the best value, followed by Splitting Longest Edge with a value of 0.63.

4. **Perimeter Area Ratio (PAR)**: Finally, the algorithm that obtained the best metric was Quadtree Refining with 0.24, remembering that lower is better.

6.6.2.2. Initial Mesh: KD-tree - Maximum area equal to 198 area units

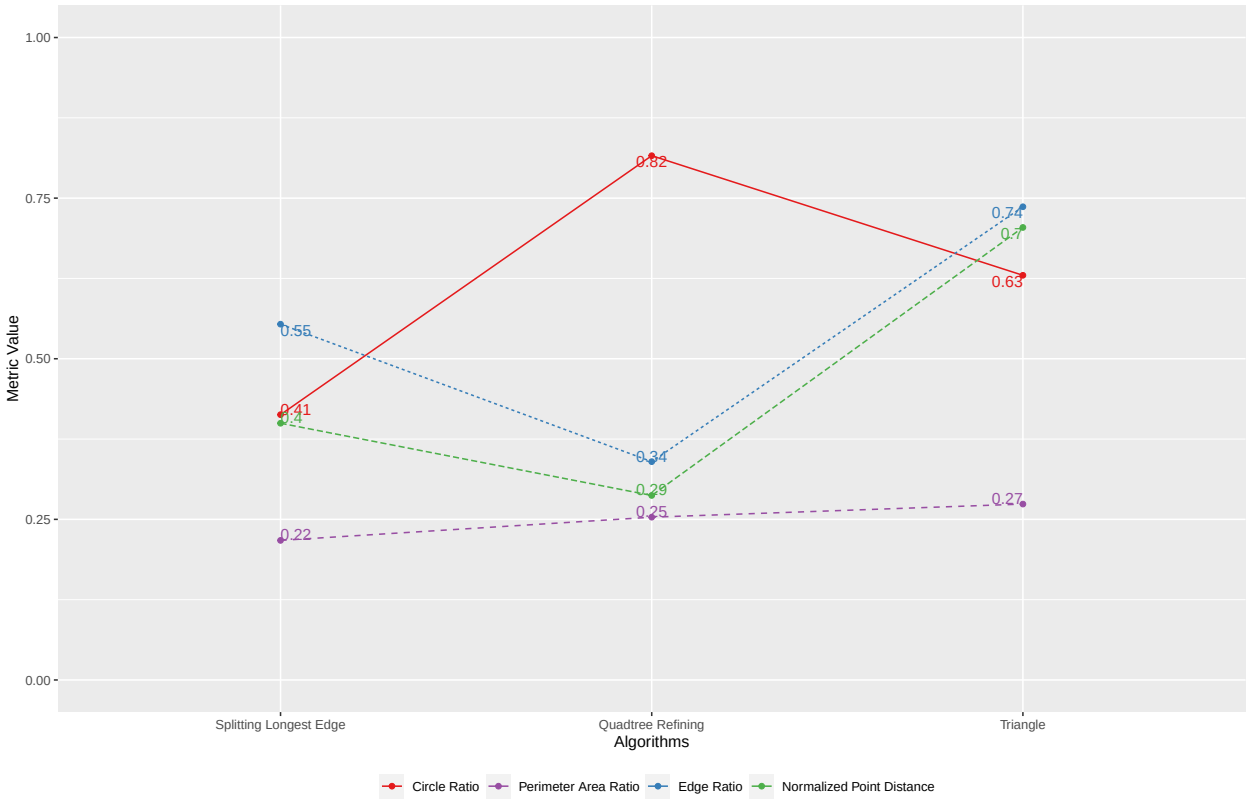


Figure 6.37: Initial Mesh: KD-tree - Quality metric comparison with Triangle (Max: 198 area units).

1. **Circle Ratio (CR)**: In this metric the best result was obtained by Quadtree Refining performed on the KD-tree, obtaining a value of 0.82 compared to Triangle, which obtained 0.63. The lowest result was obtained with Splitting longest edge, which had 0.41.

2. **Edge Ratio (ER)**: In this metric Triangle obtained the best performance with 0.74, followed by Splitting longest ege with 0.55.

3. **Normalized Point Distance (NPD)**: Here Triangle also obtained the best result with 0.7, followed by Splitting longest edge which was 0.4.

4. **Perimeter Area Ratio (PAR)**: In this metric Triangle obtained the worst performance, being the best Splitting longest edge with 0.22 and then Quadtree Refining with 0.25. (The smaller it is, the better the metric).

6.6.2.3. Initial Mesh: Quadtree with Mid Point strategy - Maximum length equal to 15 length units

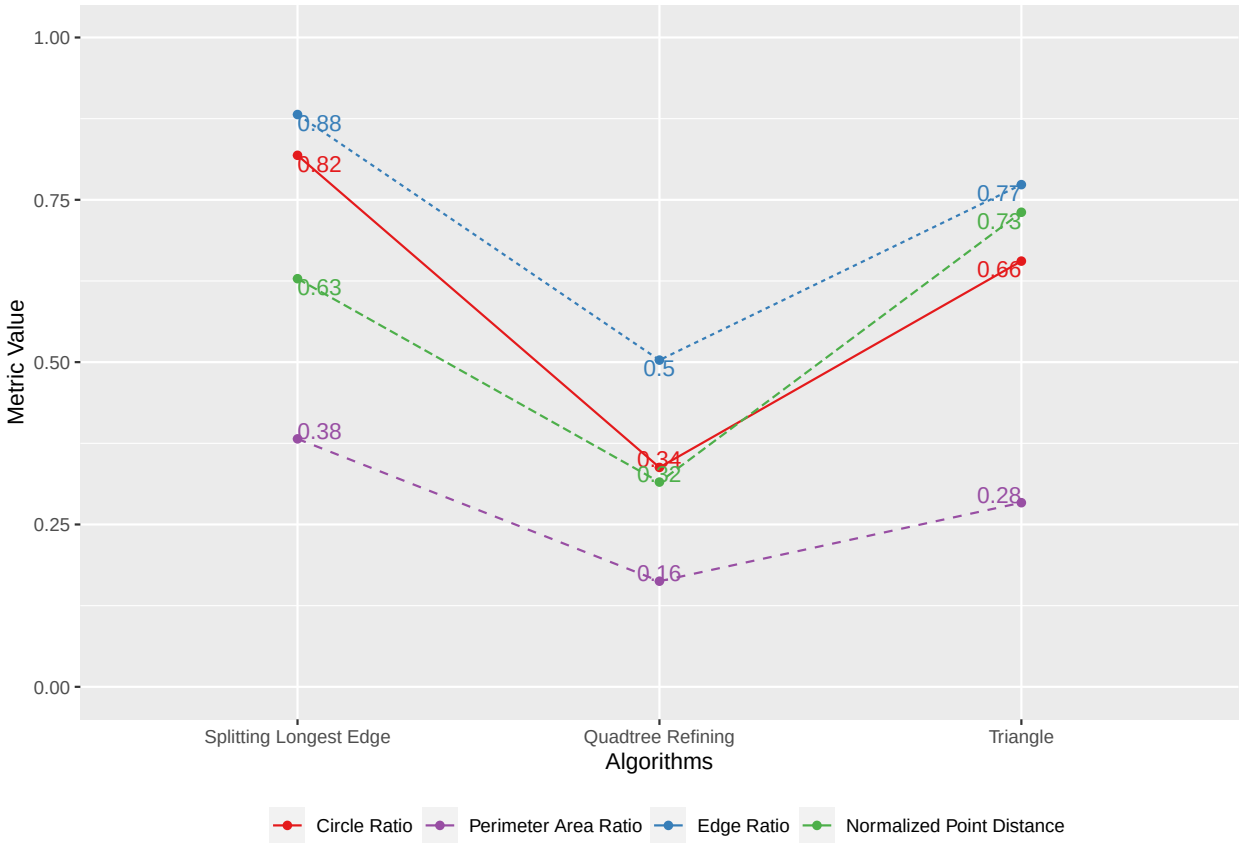


Figure 6.38: Initial Mesh: Quadtree - Quality metric comparison with Triangle (Max: 15 length units).

1. **Circle Ratio (CR)**: In this metric, triangle obtained 0.66, while the Splitting longest edge algorithm obtained a better metric with a value of 0.82. The quadtree refining algorithm obtained 0.34, a lower value than the other two methods.

2. **Edge Ratio (ER)**: Continuing with the previous behavior, the refinement of Splitting Longest Edge obtained 0.88, while Triangle obtained 0.66.

3. **Normalized Point Distance (NPD)**: In this metric Triangle obtained 0.76 being the algorithm that had the best value, followed by Splitting Longest Edge with a value of 0.63.

4. **Perimeter Area Ratio (PAR)**: Finally, the algorithm that obtained the best metric was Quadtree Refining with 0.16, remembering that lower is better.

6.6.2.4. Initial Mesh: KD-tree - Maximum length equal to 15 length units

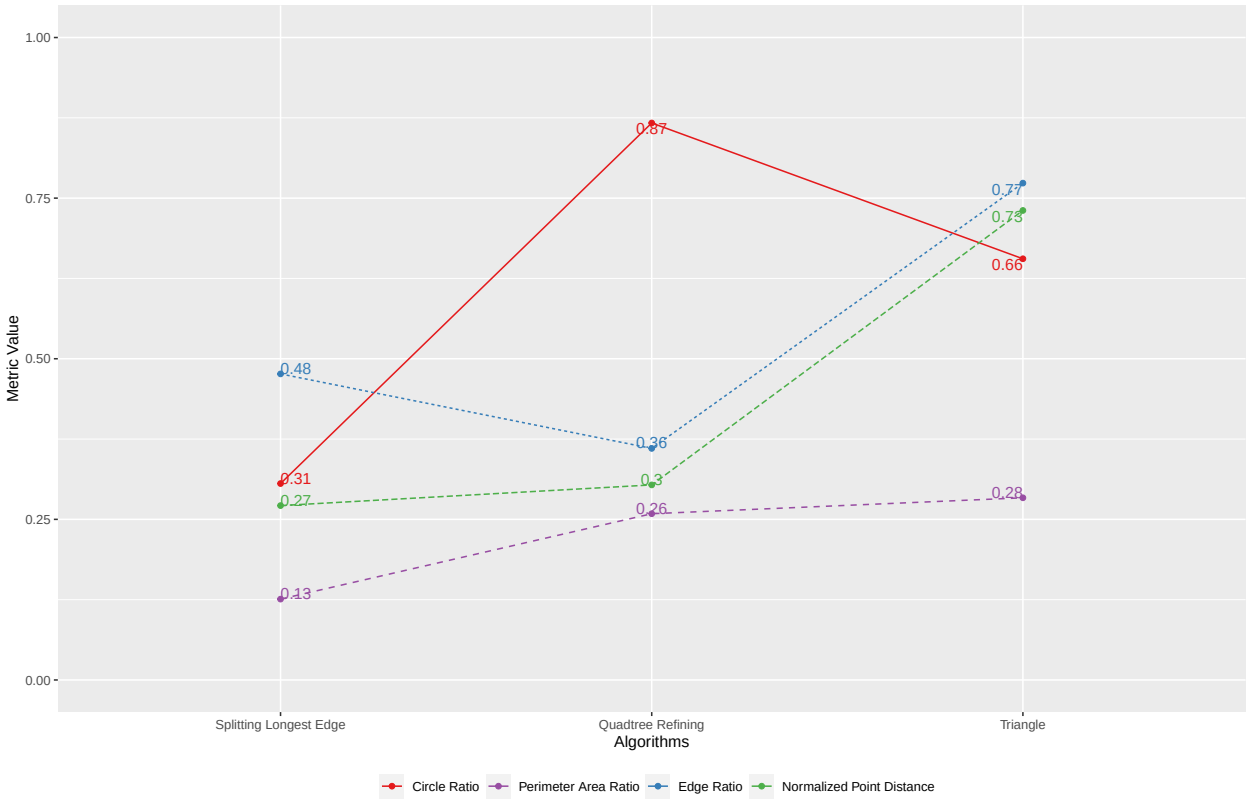


Figure 6.39: Initial Mesh: KD-tree - Quality metric comparison with Triangle (Max: 15 length units).

1. **Circle Ratio (CR)**: In this metric, the Quadtree refining algorithm obtained the best performance with a value of 0.87, followed by Triangle with a value of 0.66.

2. **Edge Ratio (ER)**: Here Triangle had the best performance with a value of 0.77, with the Splitting Longest Edge algorithm being the second best with 0.48.

3. **Normalized Point Distance (NPD)**: Like the previous metric, Triangle had the best result at 0.75, while our two strategies had similar results close to 0.3.

4. **Perimeter Area Ratio (PAR)**: Finally in PAR, the best metric was obtained in Splitting longest edge with 0.13, followed by Quadtree Refining and Triangle with results of 0.26 and 0.28 respectively.

Chapter 7

Conclusions

In this thesis work, we validate the veracity of our hypotheses regarding the meshes of arbitrary polygons generated by quadtrees and kd-trees, in particular, about the least number of elements and the quality of these in relation to the metrics. To test the hypotheses, we create a mesh generator that uses different types of point insertion algorithms to explore how the mesh creation and refinement behaves comparing them with Triangle.

7.1. Application

The application was developed using the web application technologies that are used today, being written in Typescript to give it a more object-oriented approach. We built this application in order to be an aid in the visualization of polygon meshes, and the study of different refinement algorithms, so performance in time and resources is not an objective of this thesis. The performance at the level of visualization and drawing of meshes was not optimized at a deep level, using a library that abstracts the functionalities of WebGL. As future work, we proposed an implementation that takes these aspects into account, integrating GPU and parallelism.

The application was built thinking about the ease of integration of new refinement algorithms, new data structures that partition two-dimensional space, and new metrics for polygons. That is why interfaces and abstract classes were built in such a way that adding a new strategy means only implementing or extending some of them. Another important aspect of the application is its ease of use and the versatility it has of being up on a server and being accessed from any web browser. In addition to this, the user is able to obtain information on the current mesh graphically, and propose refinements to certain areas or to certain polygons that do not meet a certain criteria.

The application within its functionalities has the ability to create a geometry using as contour points the clicks that the user makes on the screen. When the user finishes the process, the points are joined in the same order in which they were inserted and an initial geometry is formed, which is refined using any of the implemented algorithms, generating an initial polygon mesh.

In addition to the above, the application is capable of performing refinements in a certain

region chosen by the user using the mouse. This feature is not present in Triangle, so to obtain a density of polygons in a certain region in that program, the entire mesh must be refined. Refine by regions is important to carry out simulations using the VEM method on polygon meshes where a certain region is relevant to the user since collinear points are inserted in the edges, reducing the final number of polygons.

7.2. Results of Experiments

Below we present the conclusions of the results of our experiments, dividing them into three categories: Quadtree results, KD-tree results and finally the comparison with Triangle.

7.2.1. Quadtree results

The initial meshes generated by Quadtrees obtain a greater number of elements, due to the cuts that must be made with each polygon. Quadtrees using the mid point algorithm tend to create quadrilaterals, while the algorithm of random or non-arbitrary points forms elongated figures if the points are very close to each other.

The analysis of the metrics in the quality refinements was done on an initial mesh generated by a quadtree with the division algorithm using the midpoint. Taking this into consideration, the results obtained are that by imposing restrictions on the mesh area, the stricter it is and the better the result of successive quadtree refining, versus the application of quality algorithms. This behavior is seen in all metrics except the Perimeter Area Ratio, whose best performance was obtained with refinement using the centroid. The same trend happens when the restriction is over the length of the mesh.

7.2.2. KD-tree results

The initial meshes generated by KD-trees are faster than those generated by quadtrees, and they generate fewer elements and of better quality. We attribute this to the fact that the cuts are made at the same insertion points, and on a single axis, so there are fewer intersections.

Its in-depth analysis was made when comparing the algorithms with Triangle. The number of polygons generated when a restriction is imposed on the maximum area of a mesh turned out to be less than those generated by Triangle when using a KD-tree with the Splitting Longest Edge algorithm, however, the number of points for the construction of the mesh turned out to be double.

On the other hand, the performance of KD-trees when a restriction is imposed on the maximum length of a polygon mesh drops drastically, generating three times as many polygons with refinement per quadtree and twice as many polygons using splitting longest edge, in relation to what was obtained with Triangle.

7.2.3. Comparison with Triangle

In the experiment of placing a limit on the maximum area going from 1979 to 198, we see that the number of polygons generated by a KD-tree with splitting longest edge (1032) and those generated by a Quadtree with splitting longest edge (1056) are less than those polygons generated by Triangle (1069). In spite of that, Triangle does a better treatment of points obtaining in number approximately half of those required in our best algorithms.

However, where our application has better results than Triangle is when it is necessary to perform a refinement in a certain area of the polygon mesh. According to the results obtained in our experiments, when refining inside a region chosen by the user we obtained 63 polygons and 95 points, while Triangle obtained 108 polygons and 74 points. The results show to be good for the application of simulations using mathematical methods such as VEM, since the number of polygons is lower, but still meets the quality criteria imposed by the user in the region of interest. Moreover, our refining algorithm generates collinear points at the edges of the polygons, which contributes in a higher density of points inside the region and therefore, obtaining a better solution when performing mathematical simulations.

The process of generating the refined mesh in a certain region was faster in our application taking 1.6 ms (without considering the painting of the mesh on canvas), while Triangle took 2 ms. However, we believe that a larger number of experiments are needed to obtain a more accurate time analysis.

In relation to metrics, when the maximum area is restricted to 178, the polygons generated by a quadtree with splitting longest edge obtain better metrics in ER and CR than Triangle. On the other hand, when using a KD-tree the metrics are all lower in our algorithms except CR using a refinement with quadtree.

When we limit the maximum length of a polygon mesh from 30 to 15 units long, we see that a quadtree that subdivides itself with the midpoint strategy, gets 2234 polygons, while Triangle gets 3106. Triangle in this case also has a better treatment of points, less in related to all our algorithms. In relation to metrics, the same trend is maintained when the maximum area was limited.

Future Work

In relation to future work, the application has a lot of room for improvement. In the first instance, as previously explained, the application was not made with time efficiency in mind, so a programming language such as C ++ that allows more effective handling of program memory and better handling of cutting operations, it would bring about a substantial improvement to the application.

We propose an improvement in the number of elements that the application is capable of processing. Currently the application is capable of processing refinements up to level 4 or 5, depending on the algorithm, in a time close to 1 minute, for an approximate number of 7000 polygons. However, having a greater number of polygons, the time increases considerably, so the refinement algorithms can be optimized for a greater number of polygons, or optimize the painting of the mesh on the application canvas, since this process is costly. For this, a better mesh buffer management is proposed redrawing only those polygons that have changed, and keep in the canvas those that remain unchanged.

In relation to the type of polygons that the application can support, currently it has as a restriction that these cannot be auto-intersected nor can they have holes. This is why a proposed improvement is to modify the application so that it can support these types of polygons. The application's polygons cannot contain interior points inside them either, since when receiving a geometry drawn by the user, the application assumes that all points are on the edge of the polygon and not outside or inside it. The same happens when receiving a mesh in .off format, assuming that the existing points are the ones that make up the polygons, ignoring those that are not on their edges.

There is also a lot of room for improvement in terms of new algorithms. For example, it is proposed to create new quadtrees or KD-trees division algorithms that do not produce elements that tend to be quadrilaterals with right angles, or an algorithm that locally sees which is the best union of points to not produce elements unnecessarily small.

Finally, a validation analysis of the meshes generated by our application using mathematical methods such as VEM is proposed, since according to our results, the meshes produced have characteristics (greater refinement and high density of points in the region of interest) that would allow us to generate better solutions when the user is interested in performing a simulation in a certain region of the polygon mesh.

Bibliography

[1] Michael Aftosmis, Marsha Berger, and John Melton. "Adaptive Cartesian Mesh Generation". In: (Nov. 2000).

[2] M Attene et al. "Benchmark of Polygon Quality Metrics for Polytopal Element Methods". In: *arXiv preprint arXiv:1906.01627* (2019).

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The Quickhull algorithm for convex hulls". In: *ACM Transactions on Mathematical Software* 22.4 (1996), pp. 469–483.

[4] María Cecilia Bastarrica and Nancy Hitschfeld-Kahler. "Designing a product family of meshing tools". In: *Advances in Engineering Software* 37.1 (2006), pp. 1–10.

[5] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008.

[6] Marshall W Bern and Paul E Plassmann. "Mesh Generation." In: *Handbook of computational geometry* 38 (2000).

[7] Paul Bourke. "Calculating the area and centroid of a polygon". In: *Swinburne Univ. of Technology* 7 (1988).

[8] Aldo Canepa, Nancy Hitschfeld-Kahler, and Claudio Lobos. "Camarón: a visualization tool for the quality inspection of polyhedrical meshes". In: (2015).

[9] Aldo Canepa et al. "Camarón: An Open-source Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes". In: *International Conference on Computer Graphics Theory and Applications*. Vol. 2. SCITEPRESS. 2016, pp. 130–137.

[10] The Geometry Centre. *QHull*. 1995. URL: http://www.qhull.org/ (visited on 10/22/2017).

[11] Heng Chi, Lourenço Beirão da Veiga, and Glaucio H Paulino. "A simple and effective gradient recovery scheme and a posteriori error estimator for the Virtual Element Method (VEM)". In: *Computer Methods in Applied Mechanics and Engineering* 347 (2019), pp. 21–58.

[12] Heng Chi et al. "Virtual element method (VEM)-based topology optimization: an integrated framework". In: *Structural and Multidisciplinary Optimization* 62.3 (2020), pp. 1089–1114.

[13] Paolo Cignoni et al. "Meshlab: an open-source mesh processing tool." In: *Eurographics Italian chapter conference*. Vol. 2008. Salerno, Italy. 2008, pp. 129–136.

[14] Q. Du, V. Faber, and M. Gunzburger. "Centroidal Voronoi Tessellations: Applications and Algorithms". In: *SIAM Rev.* 41 (1999), pp. 637–676.

[15] Qiang Du and Max Gunzburger. "Grid generation and optimization based on centroidal Voronoi tessellations". In: *Applied mathematics and computation* 133.2-3 (2002), pp. 591–607.

[16] Qiang Du and Desheng Wang. "Recent progress in robust and quality Delaunay mesh generation". In: *Journal of Computational and Applied Mathematics* 195.1-2 (2006), pp. 8–23.

[17] Qiang Du and Desheng Wang. "Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations". In: *International journal for numerical methods in engineering* 56.9 (2003), pp. 1355–1373.

[18] Qiang Du et al. "Centroidal Voronoi tessellation algorithms for image compression, segmentation, and multichannel restoration". In: *Journal of Mathematical Imaging and Vision* 24.2 (2006), pp. 177–194.

[19] Ramsay Dyer, Hao Zhang, and Torsten Möller. "Voronoi-Delaunay duality and Delaunay meshes". In: *Proceedings of the 2007 ACM symposium on Solid and physical modeling*. 2007, pp. 415–420.

[20] Steven Fortune. "Voronoi diagrams and Delaunay triangulations". In: *Computing in Euclidean geometry*. World Scientific, 1995, pp. 225–265.

[21] Erich L Foster, Kai Hormann, and Romeo Traian Popa. "Clipping simple polygons with degenerate intersections". In: *Computers & Graphics: X* 2 (2019), p. 100007.

[22] G. Garretón et al. "A New Approach for 2-D Mesh Generation for Complex Device Structures". In: *NUPAD V - Technical Digest* (1994), pp. 159–162.

[23] Günther Greiner and Kai Hormann. "Efficient clipping of arbitrary polygons". In: *ACM Transactions on Graphics (TOG)* 17.2 (1998), pp. 71–83.

[24] Leonidas Guibas and Jorge Stolfi. "Primitives for the manipulation of general subdivisions and the computation of Voronoi". In: *ACM Transactions on Graphics* 4.2 (1985), pp. 74–123. arXiv: arXiv:1011.1669v3. URL: http://portal.acm.org/citation.cfm?doid=282918.282923.

[25] Nancy Hitschfeld-Kahler. "Generation of 3D mixed element meshes using a flexible refinement approach". In: *Engineering with Computers* 21.2 (2005), pp. 101–114.

[26] K. Ho-Le. "Finite element mesh generation methods: a review and classification". In: *Computer-Aided Design* 20.1 (1988), pp. 27–38.

[27] Grégory Legrain, Raphaël Allais, and Patrice Cartraud. "On the use of the extended finite element method with quadtree/octree meshes". In: *International Journal for Numerical Methods in Engineering* 86.6 (2011), pp. 717–743.

[28] Claudio Lobos and Eugenio González. "Mixed-element Octree: A meshing technique toward fast and real-time simulations in biomedical applications". In: *International Journal for Numerical Methods in Biomedical Engineering* 31.12 (Dec. 2015), n/a–n/a. arXiv: NIHMS150003. URL: http://onlinelibrary.wiley.com/doi/10.1002/cnm.1494/full%20http://doi.wiley.com/10.1002/cnm.2725.

[29] F Pascal and JL Marechal. "Fast adaptive quadtree mesh generation". In: *1998 International Meshing Roundtable* (1998).

[30] Axelle Pochet et al. “A new quadtree-based approach for automatic quadrilateral mesh generation”. In: *Engineering with Computers* 33.2 (2017), pp. 275–292.

[31] J. Ruppert. “A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation”. In: *Journal of Algorithms* 18 (1994), pp. 548–585. URL: http://hdl.handle.net/2060/19970014411.

[32] Jonathan Richard Shewchuk. “Delaunay refinement algorithms for triangular mesh generation.” In: *Computational Geometry* 22 (2002), pp. 21–74.

[33] Jonathan Richard Shewchuk. “Reprint of: Delaunay refinement algorithms for triangular mesh generation”. In: *Computational Geometry: Theory and Applications* 47.7 (2014), pp. 741–778. URL: http://dx.doi.org/10.1016/j.comgeo.2014.02.005.

[34] Jonathan Richard Shewchuk. *Triangle A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.* 2005. URL: https://www.cs.cmu.edu/~quake/triangle.html (visited on 10/22/2017).

[35] Jonathan Richard Shewchuk. “Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator”. In: (1996), pp. 203–222. URL: http://link.springer.com/10.1007/BFb0014497.

[36] Hang Si. “TetGen, a Delaunay-based quality tetrahedral mesh generator”. In: *ACM Transactions on Mathematical Software (TOMS)* 41.2 (2015), pp. 1–36.

[37] Hang Si. *TetView: A tetrahedral mesh and piecewise linear complex viewer.* 2011.

[38] Lokesh Singh. “Mesh Generation : A Critical Review”. In: 4.6 (2016), pp. 15–21.

[39] Ivan E Sutherland and Gary W Hodgman. “Reentrant polygon clipping”. In: *Communications of the ACM* 17.1 (1974), pp. 32–42.

[40] A Tabarraei and N Sukumar. “Adaptive computations on conforming quadtree meshes”. In: *Finite elements in Analysis and Design* 41.7-8 (2005), pp. 686–702.

[41] A Tabarraei and N Sukumar. “Extended finite element method on polygonal and quadtree meshes”. In: *Computer Methods in Applied Mechanics and Engineering* 197.5 (2008), pp. 425–438.

[42] Shang-Hua Teng and Chi Wai Wong. “Unstructured mesh generation: Theory, practice and perspectives”. In: 10.3 (2000), pp. 227–266.

[43] Weiming Wu, Alejandro Sánchez, and Mingliang Zhang. “An implicit 2-D shallow water flow model on unstructured quadtree rectangular mesh”. In: *Journal of Coastal Research* 59 (2011), pp. 15–26.

[44] Mark A Yerry and Mark S Shephard. “A modified quadtree approach to finite element mesh generation”. In: *IEEE Computer Graphics and Applications* 3.1 (1983), pp. 39–46.

Appendix A

.OFF file format

The .off extension file is a way to represent the decomposition of polygons of a geometric figure. There are many variants of this type of file, which include additional information such as the color in RGB format of points and faces and the edges of the polygon mesh.

The simplest way to define an .OFF file is as follows:

1. An optional OFF header, denoting the file type
2. Number of vertices and number of faces
3. One line for each vertex, with its X, Y and Z coordinates
4. One line for each face, with the number of vertices that make it up and then the indices of each vertex separated by a space.

The following is an example [1]of a .OFF file for a cube that has RGBA color information for each of its faces.

```
OFF
#
#  cube.off
#  A cube.
#  There is extra RGBA color information specified for the faces.
#
8 6 12
   1.632993   0.000000   1.154701
   0.000000   1.632993   1.154701
  -1.632993   0.000000   1.154701
   0.000000  -1.632993   1.154701
   1.632993   0.000000  -1.154701
   0.000000   1.632993  -1.154701
  -1.632993   0.000000  -1.154701
   0.000000  -1.632993  -1.154701
   4   0 1 2 3   1.000 0.000 0.000 0.75
   4   7 4 0 3   0.300 0.400 0.000 0.75
   4   4 5 1 0   0.200 0.500 0.100 0.75
   4   5 6 2 1   0.100 0.600 0.200 0.75
   4   3 2 6 7   0.000 0.700 0.300 0.75
   4   6 5 4 7   0.000 1.000 0.000 0.75
```

[1] Example extracted from the page https://people.sc.fsu.edu/~jburkardt/data/off/off.html

In our work we used a variant of this OFF format, in which only point 2 changes. We put only the number of vertices n_v in that line, then n_v lines of vertex coordinates, later we put a line with the number n_f of faces, and later n_f lines identical to those described for the faces in the OFF format.

For example, the .OFF file for the unicorn outline used in this thesis work is shown below, where there are only the number of vertices and then the coordinates of each point, omitting the OFF line. Since the definition lines of a polygon does not exist, it is assumed that all the points are connected in the order entered, and that they make up a contour of a geometry.

```
37
160 650
190 635
205 590
220 530
280 530
355 530
370 590
385 635
400 650
415 635
436 575
445 515
454 387.5
454 305
505 320
535 314
550 290
550 260
550 215
520 155
550 65
556 50
535 59
454 104
415 101
400 110
340 170
319 191
310 236
301 305
199 311
130 335
112 386
109 446
112 530
124 587
139 638
```