Implementation and Specification of a Simple Polygon Triangulation Algorithm in Isabelle/HOL

Mika Skjelnes
Abstract

Implementation and Specification of a Simple Polygon Triangulation Algorithm in Isabelle/HOL

Mika Skjelnes

Unexpected behaviour in software can be both expensive and time-consuming to resolve. Unit-testing is a common method used to gain confidence in the correctness of implementations where the basic idea is to simulate a finite set of input-values and check if the program produces the expected outputs. This method is far from perfect as bugs are still present and a great concern in software. Formal verification have in some cases been successfully used to prove correctness in programs when exhaustive unit-testing was infeasible. This approach relies on using formal methods to prove whether the I/O-behaviour of the implementation is equivalent to its formal specification.

Computational geometry is a field that has become increasingly relevant in the industry. Numerous of algorithms in this field have a multitude of applications and are used to help solving real-life problems. As the systems using geometry algorithms scale the need of for its' components to function properly grows more critical. Formal verification can be a great tool for this task as it can prove the correctness of the components, essentially eliminating implementation bugs. Little research is however being done on formal verification in computational geometry even though it could supplement the flaws of unit testing, as seen in some other fields.

In this project, the first few steps of formal verification is performed. A program that triangulates a simple polygon is implemented in Isabelle/HOL, followed up by a formal specification of the program. In other words, the pre- and postconditions for simple polygon triangulation are specified, which could be used for verification of the implementation. The results show that formal verification is applicable for this program as the expected I/O-behaviour for the program was naturally translated into a formal specification.
## Contents

1 Introduction ........................................... 4

2 Methods ............................................... 6

3 Background ........................................... 6
   3.1 Simple Polygon .................................... 6
   3.2 Orientation ....................................... 8
   3.3 Two-Ears Theorem .................................. 8
   3.4 The Ear-Clipping Method ......................... 9
   3.5 Point In Polygon by Ray-casting .................. 10
      3.5.1 Edge-Crossing Rules ......................... 10
   3.6 A Small Showcase of Isabelle/HOL ............... 12
      3.6.1 Identifiers used in Isabelle ............... 12

4 Results .................................................. 13
   4.1 Implementation of The Ear-Clipping Method ....... 13
      4.1.1 Haskell Implementation .................... 14
      4.1.2 Isabelle-specific Implementation .......... 14
      4.1.3 Call Graph of The Triangulation Procedure . 18
   4.2 Implementation of Correctness conditions .......... 18
      4.2.1 Pre-conditions ............................. 18
      4.2.2 Post-conditions ............................ 19
      4.2.3 Call Graph of The Correctess Conditions .... 21
   4.3 Triangulation ..................................... 22

5 Related Work .......................................... 22

6 Discussion & Conclusions .............................. 23

7 Future Work ......................................... 25

8 References ............................................ 26
1 Introduction

Computational geometry is the subset of computer science whose main focus is to study algorithms associated with geometric problems. This has applications in multiple fields such as computer graphics, robotics, manufacturing among others [1]. In the last decade, the interest in such algorithms has increased greatly due to the surge of its applications in plentiful of real-life problems.

For instance, in robotics it is desirable to be able to model a geometric representation of physical objects, which generates a set of geometric problems since the robots act in physical space while making decisions based on the computed model [2]. How something is virtually represented can be approached differently. In video game engines objects are commonly modeled as triangle meshes [3]. If computations are to be done on 3D-objects, say lighting and shading, said types of algorithms are once more needed [4]. Architecture [5] and city planning [6] are additional examples of areas where geometry algorithms are used. Assuming that the relevancy of Computational Geometry will continue growing, then there would be an increased demand for research in this field.

A broadened application scope would likely also imply an increased complexity of the applications. For this to work smoothly, we need to make sure that everything works as intended. Material failure aside, it is important that the computations always deliver the expected data in comparison to the inputted data. A crash in a system of large scale due to an error in some of the numerous algorithms could be costly both in time and money to resolve such as in avionics where the software used make up a major part on the on-board systems [7].

To ensure that the algorithms themselves do not behave in an unexpected fashion, it would be desirable to be able to prove the correctness of the I/O-behaviour of the system. The act of using formal methods of mathematics to prove the correctness of algorithms is known as formal verification [8].

Formal verification traditionally belongs to computer science and has been used to show that a process exclusively works in accordance to its specification which is much similar to the goal of unit testing. The difference lies in how the goal is approached: where unit testing relies on simulations to discover bugs, formal verification relies on exploiting known properties of a process in order to structure a formal proof of its input- and output behavior [9].

Other than of those strictly part of computer science, formal verification has been applied in multiple fields such as algebra, molecular biology and quantum physics. Using different programs with support for formal verification such as Coq [10] and Isabelle [11] researchers, among others, are working on formally proving a selection of 100 mathematical theorems [12]. Formal verification in forms of probabilistic model checking has successfully been applied to the analysis of biological pathways. [13] In certain branches of quantum physics there have been some applications of formal verification. [14]

Outside academia some companies have adopted interest for formal verification as a supplementation for unit testing. Intel has, for instance, applied this approach for the last decade as a way to ensure the reliability of their components. [15] [16] In aviation, Airbus has been utilizing different formal verification techniques with the aim to ensure that their on-board systems follow the standard for avionics software [7].

Returning to Computational Geometry, this is an area where the amount of research in for-
mal verification is rather lackluster [17] [18]. Research does exist in this field and has been established around convex hull algorithms for instance [19]. As the systems using geometrical algorithms grow more complex, the desire for those algorithms being proven correct may increase as we have seen in aviation and computer hardware, to mention a few.

Polygon triangulation is a classical problem in Computational Geometry which itself can be used to solve other problems. A well-known problem is Chvatal’s Art Gallery Problem where one wants to place watchmen in an art gallery consisting of n walls. Every watchman has infinite viewing distance and can look in any direction. The only constraint is that it cannot look through walls. The aim is to place as few watchmen as possible while maintaining complete view of the whole art gallery [20]. By representing the floor as a simple polygon and generating a triangulation of it, one can determine the upper-bound of the minimal number of watchmen needed and their respective placement [21].

Figure 1: Left: Overhead view of an art gallery modeled as a simple polygon. Right: A 3-coloring can be generated on the triangulation of the polygon. Placing a watchman at each node of one specific color will result in coverage of the entire gallery. This is a direct translation of Marianne Freiberger’s example [22].

This project has its aim of implementing a triangulation algorithm on simple polygons known as “Ear-clipping method” in Isabelle and to use its support for theorem proving to formulate the correctness of the algorithm in terms of its input/output-behaviour.

The ear-clipping method is a simple-polygon triangulation algorithm that rely on finding ears in the polygon which are removed resulting in a new polygon with one less vertex. This process is repeated until only a triangle remains as the polygon. The removed ears are all triangles, whose union constitute the original polygon. Ears are consecutive triplets of vertices in a polygon with certain properties with respect to the polygon. According to the two-ears theorem, every simple polygon has at least two ears, making the ear-clipping method feasible.

As there exist problems that can be modeled as simple polygons which can be solved with the help of a triangulation of the polygon, it is desirable to be able to construct a reliable algorithm that does so for all simple polygons. To ensure that one have constructed a correct algorithm, it would be helpful to define correctness conditions for the algorithm to satisfy. This means that a set of conditions are to be specified such that if the postcondition of the algorithm satisfy the all of the correctness conditions, then the output is a triangulation of the input-polygon.

For this project, the ear-clipping method has been selected due to its simplicity. This simple polygon triangulation algorithm will be implemented in Isabelle, which has support for some useful logic such as set theory.
2 Methods

José Nelson, among others, organize methodologies of Computing Science in five categories: “Formal”, “Experimental”, “Build”, “Process” and “Model” [23]. The paper is used as a basis for the following summary of methodologies relevant to this project.

Formal methodologies are commonly concerned with examining algorithms with the aim of providing proofs regarding properties of the algorithm. A study may have its roots in a problem where the goal is to specify an algorithm that solves the problem and then prove certain properties of the algorithm such as its time-complexity or its input/output-behavior. In some cases, the problem already have known solutions. Then the purpose of the study may be to modify the solution in order to solve more general cases of the problem, or to improve its efficiency.

Build methodologies are more focused on building a product in accordance to a specification, such as implementation of an algorithm. It is not uncommon for build project to include unit testing and documentation, in addition to the implementation.

This project is a cross between a build and a formal project. The implementation part of the projects is strictly build as is consist of writing multiple small components which in the end should form a program with certain functionality, as well as writing test cases for each of the components in order to gain confidence that the desired functionality is provided. The specification of the ear-clipping method is to be considered the formal part of the project. Defining its pre- and post conditions in detail allows for its input/output-behavior to be studied.

3 Background

The next couple of paragraphs contains terminology definitions and supplementary background theory. Illustrations are provided as a visual aid to make the definitions less abstract.

3.1 Simple Polygon

Let \( V = v_0, v_1, ..., v_n \), \( n \geq 3 \) be an ordered list of points in the plane such that \( v_0 = v_n \). A \( n \)-sided Polygon (\( n \)-gon) \( P \) is defined by \( V \), with vertices \( v_0, v_1, ..., v_n \) and edges \( v_0v_1, v_1v_2, ..., v_{n-1}v_n \).

The polygon definition is based on the specification described by Tom Davis [24]. In the paper he refers to a subset of polygons called “Jordan Polygons”, which are polygons that divide the plane into an exterior and interior region. An alternative name for Jordan Polygons is “Simple Polygon”.

A \( n \)-gon \( P \) is a simple polygon \( S \) iff:

- \( v_0, v_1, ..., v_{n-1} \) are distinct
- Two intersecting edges only have one point in common, which is the vertex they intersect in.

For the sake of clarity, distinction is made between the interior of a polygon and the region of a polygon. A point \( q \) is strictly inside a simple polygon \( P \) if it is in the space defined by the boundary of \( P \) but not on its boundary. The interior of \( P \) is the set of all points which is strictly inside the polygon. The region of \( P \) is the union of points in the interior of \( P \) and the boundary of \( P \).
Figure 2: A simple polygon that consist of 8 vertices. The interior of the polygon is marked in orange and its boundary in blue. The region of the polygon is the union of its interior and boundary.

Figure 3: Not a simple polygon as edges \((v_2v_3)\) and \((v_4v_5)\) intersect at a non-vertex point.

**Triangulation of a Simple Polygon**

A triangulation of a simple polygon is a set of triangles with some properties. A high-level description of the properties is:

Given a simple polygon \(S\), a triangulation of \(S\) is a set \(T\) of triangles such that

(i) \(\forall t, t' \in T, t \neq t' \iff t \text{ and } t' \text{ have no interior point in common.}\)

(ii) \(\bigcup_{t \in T} \text{(Region of } t) = \text{(Region of } P)\)

(iii) \(\bigcup_{(v_1,v_2,v_3) \in T} \{v_1,v_2,v_3\} = \bigcup_{v \in S} \{v\}\)

The triangles in \(T\)

- are **disjoint** if (i) is satisfied,
- **cover exactly** \(S\) if (ii) is satisfied.
- have the same vertices as \(S\) if (iii) is satisfied.
3.2 Orientation

For consistency and to be able to compare the orientation of lines and simple polygons, it is desirable to state an orientation convention for the sets of points that construct the polygons. For all subsequent ordered sets of points, the resulting geometric structure will be built in a counter-clockwise fashion. In the case of a simple polygon $P$, if $P$ is ordered counter-clockwise then when traversing the boundary of the polygon by following its edges in order, its interior will always appear left of the edge [24]. The start point of an edge $e_k = (v_kv_{k+1})$ is $v_k$. Its end point is $v_{k+1}$. The start- and end points of the edge is determined by the orientation of the polygon such that its direction is in accordance to the direction of traversal when following the polygon boundary.

![Figure 5: A counter-clockwise oriented simple polygon. The arrows indicate the direction of the polygon edges where $e_0 = (v_0v_1)$ meaning that $v_0$ is the start point of $e_0$ and $v_1$ is its end point. $v_1$ is also the start point for $e_1 = (v_1v_2)$](image)

3.3 Two-Ears Theorem

Given a simple polygon $S$ with at least 4 vertices, a triple of consecutive vertices $(v_m, v, v_p) \in S$ that satisfies:

- The edge $e = (v_pv_m)$ lies entirely in the region of $S$
constitutes an ear at $v$ [25]. It is stated that every simple polygon has at least two non-intersecting ears. Removing $v$ from $S$ (clipping the ear) yields a new simple polygon $S'$ with one less vertex.

![Diagram of a polygon showcasing two triplets with their interiors shaded in blue and orange. The “blue” triplet forms an ear while the “orange” triplet doesn’t since its new edge $(v_pv_m)$ has points outside the region of the polygon.](image)

**Figure 6:** A polygon showcasing two triplets with their interiors shaded in blue and orange. The “blue” triplet forms an ear while the “orange” triplet doesn’t since its new edge $(v_pv_m)$ has points outside the region of the polygon.

### 3.4 The Ear-Clipping Method

Given a simple polygon $S$, the ear-clipping method utilizes the two-ears theorem to triangulate. We know that $S$ has at least two non-overlapping ears. The ear constitutes a triangle, which can be utilized in following matter.

Let $S = v_0v_1, v_1v_2, ... v_{n-1}v_n$ and $t = v_{k-1}, v_k, v_{k+1}$ is an ear in $S$.

By clipping $t$ from $S$ and removing $v_k$ from $S$, we get a new simple polygon $S' = v_0v_1, v_1v_2, ... v_{k-1}v_{k+1} ... v_{n-1}v_n$

$S'$ has two non-overlapping ears and since $S'$ has one less vertex than $S$, we can repeat this process until we only have a triangle left which together with all of the clipped ears constitutes the triangulation of $S$.

The time-complexity of ear-clipping triangulation is $O(n^3)$ [26] which is slow compared to other known triangulation algorithms. During the last few decades of the 20th century multiple articles were published regarding triangulation algorithms that run in $O(n \log n)$ time, such as one based on trapezoidal decompositions proposed by Raimund Seidel [27]. A triangulation algorithm that runs in linear time exist which Bernard Chazelle is responsible for discovering [28].
3.5 Point In Polygon by Ray-casting

A simple method of determining if a point $p$ is strictly inside a polygon is by shooting a ray from $p$ and counting the number of intersections with the polygon. If the final number is odd, the point is inside the polygon [29]. If the point is on the border of the polygon, the algorithm may yield different results depending on where on the polygon the point is located. Such cases should be checked prior to ray casting.

The algorithm checks for intersection with all the edges of the polygon. This creates degenerate cases of when the ray intersects a vertex of the polygon. Since the vertex is the intersection of two edges, the naïve algorithm will incorrectly count this intersection twice, one for each edge. Recall that the polygon is oriented and that every edge has one start- and end point each. Applying special rules for vertex-intersection with regards to the orientation of the edge in question will help combatting this problem.

3.5.1 Edge-Crossing Rules

Given a ray that starts at $p$ and a polygon $Ps$, the ray is shot horizontally right from its start point. All intersections between the ray and the polygon edges are counted, except for the following [29].

- The edge is upwards and intersection is made on its end point.
- The edge is downwards and intersection is made on its start point.
- The edge is parallel with the ray.
- Intersection is not made strictly right of $p$’s start point.
Examples

Figure 8: A ray that starts at R. The edge colors denote if they are oriented upwards (blue), downwards (orange) or neither (black). According to the rules for intersections with vertices, the point R is outside the polygon since the ray of R intersects with the polygon an even amount of times. S is in the other hand determined to be inside the polygon due to the odd number of intersections.

Figure 9: Ignoring cases of when points are on the boundary of the polygon may yield different results. R is considered inside the square while S is not. By checking if the point is on the polygon’s boundary prior to ray casting, one may catch such cases and draw consistent conclusions from them.
3.6 A Small Showcase of Isabelle/HOL

Isabelle is a proof assistant that allows expression of mathematical formulas in a formal language, which then can be proven using the tools provided by the program [11].

3.6.1 Identifiers used in Isabelle

*fun* or *function* are used to define functions in a similar fashion as in Haskell. Such functions may be declared recursively and by pattern matching. Isabelle will automatically try to find a termination proof for the function defined with “fun”.

Quick comparison of a recursive Haskell function translated to a “fun” in Isabelle.

```haskell
fooBar :: [Int] -> Bool
fooBar [] = True
fooBar (n:ns)
    | (n mod 2 == 1) = False
    | otherwise = fooBar ns
fun fooBar :: int list ⇒ bool
where
    fooBar [] = True
    | fooBar (n#ns) =
        (if (n mod 2 = 1) then False
          else fooBar ns)
```

The structure of a simple recursion function is pretty similar in Haskell and Isabelle. However when translating from Haskell to Isabelle, keep in mind that Isabelle does not have the same support for guards as Haskell does.

*definition* is used for defining non-recursive functions and does not support pattern matching. In the other hand, Isabelle does not have to find a termination proof for definitions unlike for funs and functions. A non-recursive version of fooBar could be represented in following manner.

```isabelle
definition fooBar2 :: int list ⇒ bool
where
    fooBar2 ns = (∀ n∈set ns. n mod 2 = 0)
```

*value* is used to evaluate function outputs with any inputs and then compare it to expected output values. Confidence in the correctness of the implementation can be gained by testing the function with a couple of inputs. In other words, this can be used to perform some unit-testing.

```isabelle
value fooBar [6,0,10,8,32] = fooBar2 [6,0,10,8,32] = True
```

The value will be compared with the evaluation and in this case “True” will be the conclusion since the statement is correct.
4 Results

The next couple of paragraphs show the structure of the final program and what functions were defined to build it. Results from the Haskell implementation are brief since its contents were translated into Isabelle. All functions are summarized, with the exception of auxiliary functions and those which are determined to serve little significant functionality. All statements in every example are evaluated to true. A design choice for representation of polygons is made that slightly diverges from the definition. Polygons in the implementation do not have their first vertex appear twice, meaning that \( v_0 \neq v_n \). With this change, a \( n \)-gon is the list of vertices \( v_1 \ldots v_n \) with edges \( v_1v_2 \ldots v_nv_1 \).

4.1 Implementation of The Ear-Clipping Method

The main function of the implementation is a simple recursive function that clips an ear if it finds one until the base case is reached. The major component of the algorithm is \texttt{isEar} which the triangulation solely depends on. To determine whether a set of three consecutive points \((v_m, v, v_p)\) is entirely inside the polygon \( P_s \), two conditions were checked.

- Other than \( v_m, v, \) and \( v_p \), no vertices in \( P_s \) are inside \( T \) or on its boundary.
- The center point of \((v_pv_m)\) is strictly inside \( P_s \)

“isEar” determines whether a triangle is an ear in a polygon by checking that no vertex other than those of the triangle is in the triangle, boundary included, and that the center point of the new edge is inside the polygon. The vertices in the input-triangle are assumed to be three consecutive vertices in the polygon.

\[
\text{isEar} :: (\text{triangle point Float}) \rightarrow \text{[point Float]} \rightarrow \text{Bool}
\]

![Figure 10: A case of when the interior of the potential ear \( t = \Delta(v_m, v, v_p) \) is inside the polygon, but a polygon vertex \( v_k \) is on the boundary of \( t \), thus concluding \( t \) not an ear.]

A couple of other functions are used in \texttt{isEar}.

“raycastIntersections” inputs a start point for the ray and a polygon to check how many times the horizontal ray will intersect with the polygon. This is done by representing the ray and every edge of the polygon as infinite lines, which adds one additional condition on top of the edge-crossing rules.

- The point of intersection with a specific edge is inside the boundaries of the finite line defined by the edge.
"noPointsInTriangle" accepts a potential ear and the set of vertices that are in the polygon but not part of the ear. It will determine if there exist a vertex among the set that is either strictly inside the triangle or on its boundary. To check if a point is inside the triangle is done by raycasting, while checking if a point is on the border is done by checking if the two vertices responsible for the new edge and the point of interest are on the same infinite line.

\[
\text{noPointsInTriangle} :: \text{triangle} \rightarrow \text{point} \rightarrow \text{Int}
\]

"isSameLine" determines whether two finite lines are on the same infinite line. This is done by comparing their slopes and vertical offsets.

\[
\text{isSameLine} :: \text{point} \rightarrow \text{point} \rightarrow \text{point} \rightarrow \text{Bool}
\]

"onFinLine" determines whether a point is on or between the finite line defined by two points, given that the three points are on the same infinite line. This function is mainly used in the raycasting function to determine if the intersection between the ray and a polygon edge is within the start and end point of the edge.

\[
\text{onFinLine} :: \text{point} \rightarrow \text{point} \rightarrow \text{point} \rightarrow \text{Bool}
\]

4.1.1 Haskell Implementation

Prior to getting started in Isabelle, Haskell was the programming language of choice to implement the algorithm. The purpose was to get familiar with the algorithm in a familiar environment where issues with language-specific syntax and structure wouldn’t be present. All functions written in haskell were then translated to isabelle, therefore the function descriptions are available in the section about the isabelle implementation.

A package containing some elementary data structures for points, lines and others related to geometry, named “geom2d” was used to access readily available data structures for points, lines and polygons, and functions such as “pointInTriangle”.

The program requests the user to specify the file name of document containing data about the polygon to be triangulated. The document should contain a set of floats with a single space in between every entry such that every pairwise entry represents a point in the plane. The points should in a counter-clockwise fashion construct a simple polygon by the order they appear in. Finally, a triangulation represented as a list of triplets will in the stored in a new text-file under the name “output_filename”.

The function “triangulate” fetches the three first points in the list and checks if the triangle forms an ear in the polygon. A found ear is clipped by removing v from the polygon, creating new edge (v, vp) and saving (vm, v, vp) to the triangle list. If the triple is shown not to be an ear, vm is sent to the back of the list. The two-ears theorem ensures termination, given that the input is a simple polygon.

\[
\text{triangulate} :: \text{point} \rightarrow \text{triangle} \rightarrow \text{point} \rightarrow \text{Int}
\]

4.1.2 Isabelle-specific Implementation

Unlike in the Haskell implementation, no geometry package was used for geometry functions in Isabelle.
We let a point be represented as a tuple of rationals \((\text{rat} \times \text{rat})\), abbreviated as \text{rat point}. A triangle would be represented as a triple of points.

\[(\text{rat point} \times (\text{rat point} \times \text{rat point})) \iff ((\text{rat} \times \text{rat}) \times ((\text{rat} \times \text{rat}) \times (\text{rat} \times \text{rat})))\]

This could naturally be used together with the list-type of Isabelle to represent a polygon.

\[(\text{rat point}) \ list \iff (\text{rat} \times \text{rat}) \ list\]

\text{type-synonym} 'a point = 'a \times 'a

\text{type-synonym} 'a triangle = 'a point \times 'a point

Doing this instead of defining new data types makes it possible to rely on already-existing proofs for pairs in the main package. Accessing x and y-values of a point is done by functions “fst” and “snd”.

Following is an overview of the majority of functions implemented in Isabelle. Functions excluded are those of trivial functionality, simple computations for instance.

Determines whether an edge is oriented upwards or not as described in Orientation. An edge can only be upwards if the y-value of its end point is explicitly greater than the y-value of its start point.

\[
\text{fun edgeIsUpwards :: } (\text{rat point} \times \text{rat point}) \Rightarrow \text{bool} \\
\text{where} \\
\text{edgeIsUpwards} (vs, ve) = (\text{if snd ve} > \text{snd vs} \text{ then True else False})
\]

\text{Example}

\text{value edgeIsUpwards } ((0,0),(1,1)) = \text{True} \\
\text{value edgeIsUpwards } ((0,0),(1,0)) = \text{False}

“isSameLine” used to determined whether two edges are on the same infinite line. Edges are essentially finite lines and can under certain circumstances be on one infinite line. “slope” computes the slope of a line and “offset” computes the vertical offset of a line. Given a point \((x, y)\) on a line, some may refer the slope as the “k-value” \((\Delta y / \Delta x)\) and the vertical offset as the “m-value” \((y - (kx))\).

\text{definition isSameLine :: } (\text{rat point} \times \text{rat point}) \Rightarrow (\text{rat point} \times \text{rat point}) \Rightarrow \text{bool} \\
\text{where} \\
isSameLine L1 L2 = (\text{let} \\
slope1 = \text{slope } L1; \\
slope2 = \text{slope } L2 \\
in \\
slope1 = \text{slope2} \land \\
((\text{case } \text{slope1} \text{ of } \text{None } \Rightarrow (\text{fst } \text{slope1}) = (\text{fst } \text{slope2}))) \\
\lor \\
(\text{slope1 } \neq \text{None}) \\
\land \\
(\text{calcLinear } (\text{the } \text{slope1}) \ (\text{offset } \text{snd } L1) \ (\text{the } \text{slope1}) \ 0) \\
= (\text{calcLinear } (\text{the } \text{slope1}) \ (\text{offset } \text{snd } L2) \ (\text{the } \text{slope1}) \ 0)))
\]

\text{Example}

\text{value isSameLine } ((1,0),(2,0)) ((8,0),(10,0)) = \text{True}
“onFinLine” determines if a point is on a specified finite line, such as an edge. The auxiliary function assumes that the point is on the infinite line described by the edge and that the point is neither the start or end point of the edge. These conditions are checked and caught in advance.

```haskell
fun onFinLine :: rat point ⇒ rat point ⇒ rat point ⇒ bool
where
  onFinLine p vs ve = (if (p = vs ∨ p = ve) then True else
    if ¬(isSameLine (p, vs) (p, ve)) then False
    else (onFinLineAux p vs ve (slope (p, vs))))
```

Example

```haskell
value onFinLine (2,2) (−10,−10) (1,1) = False
value onFinLine (1,1) (2,2) (−10,−10) = True
```

“lineIntersection” computes the intersection point of two lines if there is any. In the case of the lines being parallel, “None” is returned. Intersection is checked between two infinite lines, described by the two finite lines in the input. If the infinite lines are not parallel, then there exist a single point of intersection which is computed by finding the x-value of intersection and then computing the corresponding y-value for that x-value. “getCommonX” finds the x-value of intersection and “calcLinear” computes the y-value for a line given a x-value, a slope value, and a vertical offset value.

```haskell
fun lineIntersection :: (rat point × rat point) ⇒ (rat point × rat point) ⇒ (rat point) option
where
  lineIntersection (p1, p2) (q1, q2) =
    liAux (p1, p2) (slope (p1, p2)) (q1, q2) (slope (q1, q2))
```

Example

```haskell
value lineIntersection ((17, 1), (18,1)) ((22,−2), (20,4)) = Some (21, 1)
value lineIntersection ((0,0),(1,0)) ((5,0),(6,0)) = None
```

As described in “Point In Polygon by Raycasting”, this function will calculate the number of times a horizontal ray intersects with the polygon with respect to the intersection-rules specified. This is achieved by recursively traversing all of the polygon edges while keeping track of the intersection count.

```haskell
fun raycastIntersections :: rat point ⇒ (rat point) list ⇒ nat
where
  raycastIntersections p [] = 0
  | raycastIntersections p Ps = rciAux p Ps
```

Example

```haskell
value raycastIntersections (17, 1) [(20,−2),(22,4),(1,1)] = 1
```

“pointOnPolygonBoundary” determines whether a point p is on the boundary of a polygon Ps. This is achieved by checking if p is on at least one of the edges (finite lines) of Ps. If p is a vertex of Ps, then p is on two edges of Ps.
**definition** pointOnPolygonBoundary :: rat point ⇒ (rat point) list ⇒ bool
where
pointOnPolygonBoundary p Ps = (∃(vs, ve) ∈ (list.set (polyedges Ps)). (onFinLine p vs ve))

**Example**

value pointOnPolygonBoundary (0, 0) [(1, 0), (2, 0), (2, 1), (1, 1)] = False
value pointOnPolygonBoundary (5, 0) [(0, 0), (2, 0), (4, 0), (6, 0), (5, 7)] = True

“PointInPolygon” uses the results from “raycastIntersections” to determine whether the point is strictly inside the polygon or not. “raycastIntersections” may consider a point on the polygon boundary to be inside or not, therefore an additional check that the point is not on the boundary is needed.

**fun** pointInPolygon :: rat point ⇒ (rat point) list ⇒ bool
where
pointInPolygon - [] = False
| pointInPolygon p vs = (if (raycastIntersections p vs) mod 2 = 1 ∧ ¬(pointOnPolygonBoundary p vs) then True else False ) |

**Example**

value pointInPolygon (0, 1) [(1, 1), (26, -5), (24, -4), (22, -2), (21, 1), (22, 4), (24, 6), (26, 7)] = False
value pointInPolygon (0, 1) [(0,0),(2,0),(2,2),(0,2)] = False

“isEar” is the primary component of the algorithm and uses a couple of functions to determine whether a set of three points constitute an ear in the polygon. The input is assumed to be three subsequent points in the polygon. There exist multiple different types of cases when the input-triangle isn’t an ear. The elementary cases are when either (a) a vertex of the polygon is strictly inside the triangle or (b) when the center point of the edge $v_pv_m$ is outside the polygon. A less obvious case is shown in Fig.10 where a vertex of the polygon is on the triangle boundary. This case is not caught by (a) since it’s not strictly inside the triangle, which calls for another required check. It is solved by asserting that no point is on the newly created edge, in other words “onFinLine” is called.

**definition** isEar :: rat triangle ⇒ (rat point) list ⇒ bool
where
isEar T Ps = (if length(Ps) < 3 then False 
else (isEarAux T (drop 3 Ps) Ps))

**Example**

value isEar ((22, -2), (21, 1), (22, 4)) [(1, 1), (26, -5), (24, -4), (22, -2), (21, 1), (22, 4), (24, 6), (26, 7)] = False
value isEar ((1, 1), (26, -5), (24, -4)) [(1, 1), (26, -5), (24, -4), (22, -2), (21, 1), (22, 4), (24, 6), (26, 7)] = True

The top-level triangulation function will recursively loop through the polygon, progressing once it finds an ear, until only a triangle is left from the input. The termination solely depends on that every simple polygon has at least two non-intersecting ears.

**fun** triangulate :: (rat point) list ⇒ (rat triangle) list
where
\[
\text{triangulate} \; [] = [] \\
\text{triangulate} \; Ps = (\text{if} \; (\text{length} \; Ps) < 3 \; \text{then} \; [] \; \text{else} \; (\text{triangulateAux} \; Ps \; []))
\]

Example
value triangulate [\( (0,0) , (0,-5) , (5,-10) , (10,-5) \] = [(\((0, 0), (5, -10), 10, -5) , ((0, 0), (0, -5), 5, -10) \)]

4.1.3 Call Graph of The Triangulation Procedure

4.2 Implementation of Correctness conditions

In addition of translating from Haskell, correctness conditions for the ear-clipping method were specified in Isabelle. The precondition is that the input constitutes a simple polygon. Its specification, which has been already been described in the background section, translates to the following predicates.

4.2.1 Pre-conditions

Distinct checks whether every point in the input is distinct by comparing the number of elements
in the input-list with the number of elements in its corresponding set. Unlike a list, a set does not allow multiple occurrences of the same value.

**definition** distinct :: (rat point) list ⇒ bool

where

distinct Ps = (card (list.set Ps) = size(Ps))

“noSelfIntersect” is a predicate on the input that disallows any intersections between edges other than at their start or end points. The predicate itself allows for polygons that self-intersect at vertex-points, but those polygons are not valid with the inclusion of the “distinct”-predicate as such polygons whose corresponding list would have some vertex occur multiple times.

**definition** noSelfIntersect :: (rat point) list ⇒ bool

where

noSelfIntersect Ps = (let Es = (list.set (polyedges Ps)) in (∀ e∈Es. (∀ e′∈(Es−{e}). (let sect = (lineIntersection e e′) in (sect ∈ {Some (fst e), Some (snd e), None})))))

**Example**

value noSelfIntersect [(0,0), (1,0), (1,1), (0,1)] = True
value noSelfIntersect [(0,0), (1,1), (1,0), (0,1)] = False

“isSimplePolygon” is the top-level function that both predicates hold for the input-list.

**definition** isSimplePolygon :: (rat point) list ⇒ bool

where

isSimplePolygon Ps = (noSelfIntersect Ps ∧ distinct Ps)

4.2.2 Post-conditions

The postcondition of the output-list is that the triangles form a triangulation of the input-polygon. In the background section, a high-level specification is provided, which after decomposition results in following predicates.

“isDisjoint” checks whether two triangles are disjoint by representing each as a set of points, where a point is in the set iff the point is in the triangle interior, and then asserting that the intersection of the two sets is the empty set.

**definition** isDisjoint :: rat triangle ⇒ rat triangle ⇒ bool

where

isDisjoint T1 T2 = ((\{p:.(rat point)}. pointInPolygon p (trisToList T1)) ∩ \{q:.(rat point)}. pointInPolygon q (trisToList T2)) = {})

“allTrianglesDisjoint” is a predicate on the output-set requiring that every triangle is disjoint from each other.

**definition** allTrianglesDisjoint :: (rat triangle) list ⇒ bool

where

allTrianglesDisjoint Ts = (∀ t∈(list.set Ts). (∀ t′∈((list.set Ts) − {t}). (isDisjoint t t′)))

“polygonRegion” computes the set of points that are strictly inside a polygon or on its boundary.

**definition** polygonRegion :: (rat point) list ⇒ (rat point) set
where
\[\text{polygonRegion } Ps = \{ p. \text{pointInPolygon } p \text{ Ps} \} \cup \{ p. \text{pointOnPolygonBoundary } p \text{ Ps} \}\]

“coverExactly” is a predicate on the output-set that requires the combined area of the triangles to be equal to the area of the polygon. It is achieved by comparing the region of the polygon with the union of each triangle region.

fun coverExactly :: (rat point) list ⇒ (rat triangle) list ⇒ bool
  where
  coverExactly [] [] = True
  | coverExactly P T = ((\text{polygonRegion } P) =
    (\bigcup (v1,v2,v3) \in (\text{list set } T). (\text{polygonRegion } [v1, v2, v3])))

“isSameVertices” is a predicate requiring that all vertices from the triangles only are from the polygon.

definition isSameVertices :: (rat point) list ⇒ (rat triangle) list ⇒ bool
  where
  isSameVertices Ps Ts = ((\text{list set } Ps) = (\bigcup (v1,v2,v3) \in (\text{list set } Ts). \{v1,v2,v3\}))

“isTriangulation” is the top-level predicate which requires that all three predicates on the output-list are satisfied.

definition isTriangulation :: (rat point) list ⇒ (rat triangle) list ⇒ bool
  where
  isTriangulation Ps Ts = ((\text{allTrianglesDisjoint } Ts) ∧ (coverExactly Ps Ts) ∧ (isSameVertices Ps Ts))

“earclipTriangulates” is a lemma stating that given a simple polygon P, the output of the triangulation function is a triangulation of P. “oops” simply tells Isabelle that the proof isn’t finished.

lemma earclipTriangulates:
  assumes isSimplePolygon Ps
  shows (isTriangulation Ps (triangulate Ps))

oops
4.2.3 Call Graph of The Correctess Conditions

Figure 12: A graph that shows the general structure of the correctness conditions. Functions called by “pointInPolygon” have already been described and are therefore not displayed here.
4.3 Triangulation

Figure 13: Left: A selection of simple polygons with slightly different properties. Top: A simple polygon with 17 vertices. Middle: A simple polygon with only two ears. Bottom: A simple polygon consisting of 7 vertices where $v_0...v_6$ are on the same infinite line. Right: A triangulation of the polygons generated by the ear-clipping algorithm implemented in Isabelle.

By checking the correctness conditions, it can be confirmed that the triangle-sets generated by the algorithm for the polygons specified indeed are triangulations. It is assumed that the correctness conditions specified are sufficient for the specification of a triangulation of a simple polygon.

5 Related Work

Triangulation by Ear-Clipping generates a set of non-overlapping triangles that cover the whole polygon without introducing new vertices, given that the input-vertices constitutes a Simple Polygon. This definition of triangulation can be more abstractly defined in order to cover more general sets of vertices as valid inputs. Yves Bertot inspects a triangulation method that works
over any set of unique points in $\mathbb{R}^2$, with triangulation defined as a set of non-overlapping triangles where every point in the input-set $Q$ is a vertex in at least one triangle. The combined area of the triangles must cover the convex hull of $Q$. This version of triangulation is then formally proven by using “Knuth’s Axioms” which is a number of elementary properties of points’ counter-clockwise orientation predicate in the plane. [17]

Formal verification has also been done on a Delaunay triangulation algorithm, where Delaunay triangulation is defined as given a set of points $Q$, the triangulation is a set of triangles whose vertices are the input points and s.t none of the points in $Q$ are inside the circumcircle of any triangle. Jean-François Dufourd and Yves Bertot specifies an algorithm that inputs a triangulation $T$ of a set of points and finds a Delaunay triangulation $T'$ by flipping the edges of $T$ [30].

Bertot and David Pichardie have also formally proven the correctness of non-triangulation convex-hull algorithms in the Euclidian Plane on paper and by implementation in Coq. The incremental-algorithm calculates the new convex-hull given a set of points $Q$, the convex hull of $Q$ and a new point $q$. The package-wrapping algorithm finds the convex hull of a set of points using a technique that relies on comparing distinct pairs of points, given an initial edge known to be part of the convex hull. Knuth’s Axioms are used to justify any decisive step of both algorithms [19].

Bertot and Pichardie also in-depth explores degenerated cases such as when aligned points are allowed in the input-set. If allowing any number of aligned points, the assumption that any 3-set of points forms a triangle no longer holds. Pay attention to the case of when the three points is on the same line. Knuth’s axioms don’t apply for such cases of three points. Therefore, Bertot and Pichardie introduces a new orientation predicate, which is used to modify Knuth’s 3rd axiom and to add a handful of additional axioms in order to accomplish a form of internal consistency of the predicate. The definition of the convex hull is relaxed in order to make the solution work. Another approach to allow degenerated cases is proposed, which also results in the definition of convex hull having to be relaxed in order make the solution work [19].

Research on triangulation algorithms also exist outside the subject of formal verification. Gang Mei, among others, works on improving the Ear-Clipping method in terms of what types of triangles the triangulation algorithms divides the polygon into and what types of polygons that the algorithm can triangulate. The concern of what triangles that are generated in the triangulation algorithm origins in that “flat triangles” bare more computation errors than non-flat triangles. The formal definition of a flat triangle is not set, as any non-equilateral triangle may considered flat if any other angles than $\frac{\pi}{3}$ in the triangle are required for non-flat property [26]. The aim of generating a triangulation with certain constraints of the resulting triangles is also related to Delaunay triangulation [31].

6 Discussion & Conclusions

Before discussing the result itself, there will be a brief paragraph describing my opinion on how the project elapsed and in which ways I would have acted and/or worked differently if the project would start today.

In general, I think that more could have been achieved in the ten-week period had I worked slightly differently. My main critique is that I had a tendency to not seek assistance from my supervisor regarding trivial problems. Instead I would waste precious time on solving these problems which could be resolved in a ten-minute meeting. For instance, I wasn’t initially aware of Isabelle’s support for set theory. Therefore I would express the predicates for triangulation
with unintuitive calculus which in the end didn’t make any sense altogether. This was quickly reversed and corrected at the next meeting, approximately half a week later. However it is still time which could have been used to engage in (unscheduled) proof work.

The triangulation function appears to output the correct set of triangles according to figure 13. Additional confidence that the implementation is correct is gained from testing of its sub-components. The weakest link in the program would probably be the parts responsible for handling the degenerate cases of intersection between the ray and the polygon. Recall that a couple of edge-crossing rules were defined to exclude certain cases of intersection. There doesn’t seem to be anything wrong with the rules as they are easy to understand and don’t interfere with regular cases of intersection. There is however a lack of published theory behind the rules. The source does not provide any proof regarding the correctness of the rules. The source also has a very low cite count and in general the rules are not really mentioned anywhere else.

Recall that this project is described as partially build and partially formal oriented. It should be trivial to see that the statement holds by reviewing what has been done in the project. The ear-clipping method has been implemented, consisting of a top-level function "triangulate" which uses a handful of various lower-level functions to compute expected output for some valid input. The formal aspect of the project falls into the second parts of the project, which entirely concerns the written correctness conditions of the triangulation algorithm. In other words, the specification of the ear-clipping method is the formal work done in the project.

The design choice regarding polygon representation was made to avoid potential confusion. In definitions it could make more sense to “force” the polygon to be closed in the plane by letting the first and last vertex in the list be the same. In the implementation this simply did not seem very intuitive, as special checks then would have to be made to not accidently pass on the same vertex twice as input to ex “isEar”.

In the results it is mentioned that a condition was added to the already existing ones defined by the edge-crossing rules. This is a side-effect of representing the polygon edges and the ray as infinite lines during intersection computation. The additional condition is necessary to ensure that intersection with the edges only counts if the actual intersection is made within the boundaries of the edge and not somewhere else on the infinite line the edge is part of.

For the most part, the implementation is roughly a direct translation of existing definitions and methods. However for some cases such as for identifying an ear, the implementation diverges from the definition, while still being logically equivalent. Reviewing these diversions will be the basis for the argument that they still are logically equivalent to the original description.

The two conditions for the set of three consecutive vertices seems to be sufficient to be equivalent to the high-level description specified in the two-ears theorem. The first condition is only satisfiable if there are no points in the polygon that are in or on the triangle, which means that if the condition is satisfied then the triangle is either strictly inside or strictly outside the polygon. Therefore it should be safe to assume that if one point of the edge is inside the polygon, then all other points in the edge are as well, given that the first condition is satisfied. The same argument could probably be made for any arbitrary point in the triangle itself since the premise that the triangle either is completely inside or outside the polygon still holds.

The functions associated with the correctness conditions are a little bit different from those associated with triangulation in the way they respond to “value” in isabelle. All non-trivial test cases of functions such as “coverExactly” and “allTrianglesDisjoint” would result in a wellsort-
edness error. The problem might have something to do with set-quantifiers such as “∀” which, when used with certain boolean functions such as “pointInPolygon”, can create a countably infinite set of rational points which can be hard to compare to other infinite sets of rational points. Therefore testing these functions have been difficult, and could be shown to contain faults when proving the lemma “earclipTriangulates”. However, arguments for its correct implementation can be achieved by reviewing correctness condition.

“isSimplePolygon” uses ”distinct” and ”noSelfIntersect” to determine whether an input-set of vertices construct a simple polygon. Both are a direct translation from the formal definition of a simple polygon. ”noSelfIntersect” relies on the correct implementation of ”lineIntersection” which is a function that computes to point of intersection between two lines, and has been tested for a multitude of different of inputs with edge-cases in consideration.

“allTrianglesDisjoint” (1) checks whether any two triangles in a triangle set are disjoint, where disjoint was defined as the intersection of two polygon interiors being the empty set. This predicate uses ”isDisjoint” which in its case solely relies on the correct implementation of ”pointInPolygon” which has been rigorously tested.

“coversExactly” (2) checks whether the union of the region of a triangle set is equal to the region of a polygon. This predicate uses ”polygonRegion” which also solely relies on the correct implementation of ”pointInPolygon”.

“isSameVertices” (3) simply checks if the union of the vertices of the triangle set is equal to the set of vertices of the polygon.

“isTriangulation” will check the three predicates above are satisfied. (1) is important to ensure that no triangles in the output-set overlap since such sets would not be considered a triangulation, even if (2) and (3) are satisfied. (2) ensures that the output-set covers the entirety of the polygon, no more and no less. A triangulation must cover the polygon exactly. (3) requires the output-set to not introduce any vertices other than what the input-polygon consist of.

All three predicates are required and cover certain cases of output-sets which the other predicates doesn’t. A set only satisfying (1) and (2) could contain triangles with vertices not in the polygon, which wouldn’t be a triangulation of the polygon. Only satisfying (1) and (3) could yield a triangle-set with ”holes” or with triangles on the wrong side of the polygon. Omitting (1) would allow overlapping triangles which a triangulation cannot contain.

To summarize, although everything stated in the project plan was completed, it is only bittersweet of what was achieved. For what was completed, the results suggest that the implementation does generate a triangulation if provided a simple polygon. But would not be fair to conclude that it does so, since proof showing that the stated correctness conditions are equivalent to the specification are missing, in addition to that it is not known if the correctness conditions are satisfied for all simple polygons.

7 Future Work

This project only concerns specification and implementation of an algorithm without proving its’ correctness. Finalizing formal verification is therefore another task of itself which is left as future work. As the pre- and post-conditions already are specified, the next step would be to prove total correctness of the implementation. This is done by showing partial correctness of the program and that it always terminates. Most likely, one would have to prove total correctness
of all the functions involved in the triangulation in order for proving correctness of the top-level function to even be feasible.

As mentioned in the introduction, formally proving algorithms in Computational Geometry is not a common occurrence and has plenty of room for expansion. It could be beneficial for the field if formal verification is done to such algorithms that are used frequently in the industry.

8 References


