

# **CS2AO17 Convex Hull**

L. L. Blumire

January 21, 2018

# Contents

<b>I. Report</b>	<b>4</b>
<b>1. Task Specification</b>	<b>5</b>
<b>2. Design</b>	<b>6</b>
2.1. Design Philosophy . . . . .	6
2.2. Programming Language Choise . . . . .	6
2.3. Desired Output . . . . .	7
2.3.1. Output Format . . . . .	7
2.3.2. Output Data . . . . .	7
2.4. Data Representations . . . . .	8
2.5. Data Transformations . . . . .	10
2.6. Program Flow & Implementation . . . . .	11
2.6.1. Args to File . . . . .	12
2.6.2. File to Input . . . . .	13
2.6.3. Input to Output . . . . .	14
2.6.4. Output to File . . . . .	23
<b>3. Tests</b>	<b>28</b>
3.1. Minimal Input . . . . .	28
3.2. Non Intersecting . . . . .	28
3.3. Simple Intersecting . . . . .	28
3.4. Path Route . . . . .	29
3.5. Example Task 1 . . . . .	30
3.6. Example Task 2 . . . . .	30
3.7. Two Paths one Polygon . . . . .	32
3.8. Arbitrary 15 Point Polygon . . . . .	33
3.9. Concave Escape . . . . .	34
3.10. Intersecting Polygons . . . . .	35
<b>4. Conclusions &amp; Reflections</b>	<b>36</b>



**Part I.**  
**Report**

# 1. Task Specification

**Task (i)** A common problem in Robotics is to identify a trajectory from a start point A to a destination point B as shown in figure 1.1. The Robots sensors indicate that there is an obstacle. Construct the convex hull consisting of the points A, B and the vertices of the polygon P (assume any arbitrary point set for the polygon with perhaps 15 points).

**Task (ii)** Extend the program in (i) to handle multiple obstacles as shown in the Figure 1.2 with the start point A going through B and destination C.

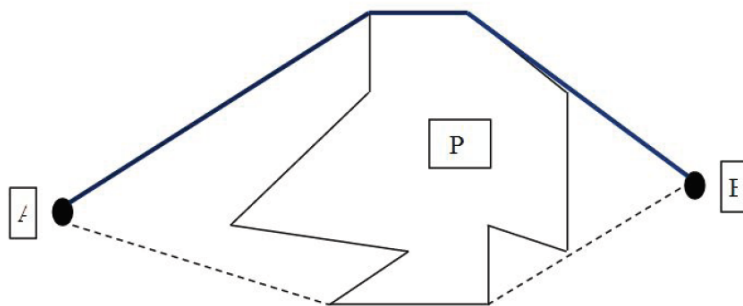


Figure 1.1.: Task 1

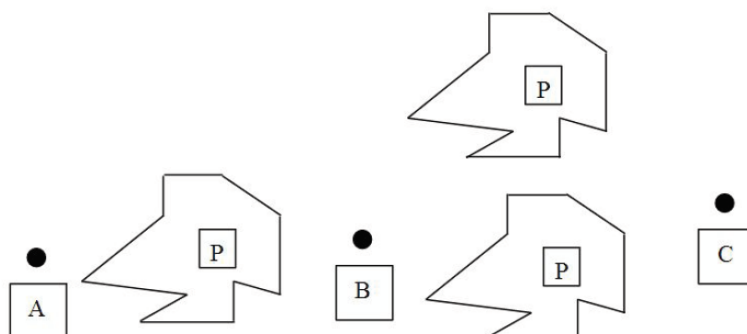


Figure 1.2.: Task 2

## 2. Design

### 2.1. Design Philosophy

I am an advocate of the ‘work-backwards’ design philosophy. By beginning with the programs desired outputs, we can analyse what is required to calculate those outputs, and then what is required to create those requirements, and so on, and so forth, until an acceptable scheme of inputs and calculations have been planned.

I also intend to design the system with the highest level of complexity, Task (ii) in mind. This is because when designing for a simpler system and then attempting to extend it, the modularity and composability of the system may not be sufficient, and substantial rewrites may be required. Designing and building to complete the maximally complex system avoids these problems.

### 2.2. Programming Language Choice

I will be producing this system in Rust, which bills itself as follows:

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

It is a compiled, functional, structured, statically and strongly typed programming language that does implicit memory management (not automatic) and provides the speed of C and C++ with a substantial degree more safety through automatic move semantics and guaranteed memory safety. It also is ideal for supporting parallelisation, although the base program will not be written with multithreading in mind, rust would make extending the program in this respect, if for example speed of execution becomes a concern, trivial. In addition to this, I have a large degree of familiarity with the language and have used it as my general-purpose programming language of choice for the past few years.

As a note, the Rust style guide recommends 100 characters as the heuristic maximum line length. This is longer than the number of characters I can place in monospaced type on the pages of this document, and therefore some lines will incur forced linebreaks hampering readability. Full source code at the time of writing can be seen at <https://github.com/LLBlumire/convex-hull-pf/tree/34ef3c1e4f5f93c2ef1ac59fdf4ffa948f510123>.

## 2.3. Desired Output

### 2.3.1. Output Format

The program is desired to have a format of outputting the data calculated inside the program to the user. It might be nice to present this information as an image, or to be able to generate it in a human and machine parse-able format for further data processing.

For image output, generating a Portable Network Graphics (png) <https://tools.ietf.org/html/rfc2083> file might be the most sensible. It is an ideal way for presenting data from a Cartesian plane to the user, and the convex hulls calculated for this program are to be done with respect to a natural-number-only aligned Cartesian plane.

In addition to this, Javascript Object Notation (json) <https://tools.ietf.org/html/rfc8259> is a standard way of presenting easily machine parse-able data. Almost all programming languages have a mechanism for processing json, making it an ideal candidate. Possible alternative: Extensible Markup Language (xml) <https://www.w3.org/TR/2006/REC-xml11-20060816/> — discarded for its unnecessary complexity.

Finally, Tom's Obvious Minimal Language (toml) <https://github.com/toml-lang/toml/blob/master/versions/en/toml-v0.4.0.md> provides a way of presenting human readable datum. It is designed for configuration files and so is much more ideal for input than output, however it still provides a much more human text format to output data in than other formats. Possible alternative: YAML Ain't Markup Language F.K.A. Yet Another Markup Language (yaml) <http://www.yaml.org/spec/1.2/spec.html> — discarded as it has shifted in focus from human readable to a machine language like json.

### 2.3.2. Output Data

These outputs will need to present the following information to the user. In the following list,

- Start Vertex
- End Vertex
- Intermediate Route Vertices
- Polygon Vertices
- Polygon Edges
- Convex Hull Edges

All fields other than Convex Hull Edges can be pulled directly from our inputs.

## 2.4. Data Representations

From the output data, we can assess a number of different types that we shall need to represent internally.

First of all, a critical decision is to be made about our base data encoding type.

Let us assess the numeric primitive we intend to use for calculations in the system. If the program is to facilitate outputting to png, then negative and fractional numbers are unnecessary, this would lend itself well to a 32 or 64 bit unsigned integer is logical. Holding negative values is however useful when performing calculations, rather than relying on integer wrapping behaviour (which may be inconsistent on some hardware). As such, a 64 bit signed integer for internal calculations, and a 32 bit unsigned integer for output to png (as these types are compatible for casting) will be our underlying data types.

Having decided this, the first data type representation we will need is a vertex. A vertex is a specific instance of a more generic idea, a Cartesian coordinate. As such, we can define as the basis for all coordinates in our system the following:

```
1  /// A coordinate in the cartesian plane.
2  pub struct Coord {
3      /// The x coordinate.
4      pub x: i64,
5      /// The y coordinate.
6      pub y: i64,
7  }
```

In addition to vertices, we also need to produce a means of representing edges. There are a number of options for representing edges. An edge is a specific instance of a more general idea, a line segment. We can represent a line segment as the composition of a start and end coordinate. In addition to this, we mandate that one of the coordinates should be the leftmost (lowest x), and in the case of a tie it should be the topmost (lowest y). This will not be rigidly enforced by the code in the event that for some reason code requires directional edges rather than arbitrary line segments.

I have chosen to use the start–end representation rather than a start–delta to allow quick composing of vertices into a segment.

```
1  /// Represents a line segment AB.
2  pub struct Segment {
3      /// The left start of the line segment.
4      pub a: Coord,
5
6      /// The right end of the line segment.
7      pub b: Coord,
8  }
```



Next, we must determine our representation of polygons. There are two obvious choices here, we could say that a polygon is a set of segments, or we could say that a polygon is an ordered set of vertices, with the final vertex leading into the first vertex. I am going to opt for a vector<sup>1</sup> of vertices, as it eliminates the redundant repetition of vertices that would be present in the set of vertices representation. In addition to this, we mandate that the coordinates listed are done so counterclockwise, this will not be enforced by the code in the event that for some reason we need to define a closed hole rather than a closed polygon.

```
1 /// Represents a polygon.
2 pub struct Polygon {
3     /// The set a points that make up the polygon, ordered counterclockwise.
4     pub points: Vec<Coord>,
5 }
```

Next, we must determine our representation of hulls. They may be represented the same as polygons, however our hulls are more than simple polygon like convex hulls, they can include additional trivial edges between candidate points (start, end, route points) where no polygon blocks the path. Therefore, they should be represented as a set of segments. This will be implemented as a Segment vector.

```
1 /// Represents a Convex Hull
2 pub struct Hull {
3     /// The segments that constitute a hull.
4     pub segment_set: Vec<Segment>,
5 }
```

This initial set of data representations may not be enough for complete processing, however it should be enough for determining the inputs and outputs. The above list of shapes may be reconsidered and extended as design continues.

We are however, now ready to design the Input and Output formats, which will form the basis of how our user interacts with the device.

All facets of the input will be displayed in some format in the output (which includes in addition only the hull), and therefore building the input specification first seems sensible.

First, let us look at the requirements of input. (bulleted with '+' for required, '-' for optional).

- + Starting vertex.
- + Ending vertex.
- Route vertices.

---

<sup>1</sup>A contiguous growable array type, written **Vec**<T> but pronounced 'vector'.

- Polygons.

As such, the following definition would fit these requirements.

```
1  /// The input for deserialization.
2  pub struct Input {
3      /// The start of the path.
4      pub start: Coord,
5
6      /// The end of the path.
7      pub end: Coord,
8
9      /// Points that must be passed in order from start to end.
10     pub route: Vec<Coord>,
11
12     /// The polygons that block the path.
13     pub polygons: Vec<Polygon>,
14 }
```

By using rusts ‘Serde’ framework, serialising and deserializing this from TOML is trivial. We will instruct Serde to in the event that no elements for `route` or `polygons` are provided, to initialise an empty vector in their place. We will also instruct Serde to alias `polygons` to `polygon` for the purpose of human readability when writing input files. This gives us an example input configuration that can be seen in Source Code Listings 14, 15 and 16.

As a follow up, we can define the output struct to contain<sup>2</sup> the input, and in addition the hulls.

```
1  /// The Output of computation.
2  pub struct Output {
3      /// The input that generated this output, if known
4      pub input: Input,
5
6      /// The point to point hulls that make up the outputs along the path.
7      pub hulls: Vec<Hull>,
8  }
```

## 2.5. Data Transformations

Now we have established our data transformations, we must establish how these data types are created, and transformed between each other.

A constructor is created for `Segment`, that validates the suggested normalisation scheme we set out earlier.

---

<sup>2</sup>In traditional inheritance based languages, it would make sense to make `Output` be a child of `Input`, however rust favours composition over inheritance, and so instead `Output` shall compose an instance of `Input`.

```

1  impl Segment {
2      /// Constructs a line segment from it's coordinates AB.
3      pub fn from_coords(a: Coord, b: Coord) -> Segment {
4          let (a, b) = match (a.x.cmp(&b.x), a.y.cmp(&b.y)) {
5              (Ordering::Less, _) | (Ordering::Equal, Ordering::Less) => (a,
6  ↪ b),
7              _ => (b, a),
8          };
9          Segment { a, b }
10 }

```

A constructor for hull is created, simply populating it from it's field.

```

1  impl Hull {
2      /// Constructs a hull from it's segments.
3      pub fn from_segment_set(segment_set: Vec<Segment>) -> Hull {
4          Hull { segment_set }
5      }
6  }

```

Polygon can be turned into it's underlying coordinate vector via an accessor, but it will also be useful to get the alternative representation discussed earlier (a vector of Segment).

All other defined data representations that are required will be created through deserialisation with Serde, and so explicit constructors and transformers are not required.

## 2.6. Program Flow & Implementation

**NB:** I will be peppering implementation code throughout this section. Full code is visible in the Appendix of this report. I will use

```

1  // ...

```

to represent omitted code. With a nearby annotation explaining that code from the function or process has been omitted. Code will be shown to you sequentially, so it is safe to assume code follows on from previously shown code at the first section of omitted code.

\*\*\*

Let us reason about our program flow in terms of transformations.

1. Args → File

2. File → Input
3. Input → Output
4. Output → File

### 2.6.1. Args to File

The first step of this process we can compute with the rust library ‘clap’. Which is a Command Line Argument Parser.

We need to take in a mandatory input file, output file, output format specifier, and scale factor for rendering the image (nearest neighbour upscaling). A sensible default format for outputting would be toml. And a sensible default scale factor would be 1.

```

1 fn main() {
2     let matches = App::new(crate_name!())
3         .version(crate_version!())
4         .author(crate_authors!())
5         .about("Finds a path along a route using a convex hull algorithm.")
6         .arg(
7             Arg::with_name("INPUT")
8                 .help("The input to process")
9                 .required(true)
10                .index(1),
11        )
12        .arg(
13            Arg::with_name("OUTPUT")
14                .help("The file to output to")
15                .required(true)
16                .index(2),
17        )
18        .arg(
19            Arg::with_name("output")
20                .help("Specify the output mode, \"toml\" or \"json\" or
→ \"png\"")
21                .short("o")
22                .takes_value(true),
23        )
24        .arg(
25            Arg::with_name("output-scale")
26                .help("Specify the output scale, only valid in \"png\" mode")
27                .short("s")
28                .takes_value(true),
29        )
30        .get_matches();

```

```

31
32 // Unwrap is safe as CLAP handles requirement of value.
33 let input_file = matches.value_of("INPUT").unwrap();
34 let output_file = matches.value_of("OUTPUT").unwrap();
35
36 let mode = matches.value_of("output").unwrap_or("toml");
37 let scale: u32 = matches
38     .value_of("output-scale")
39     .unwrap_or("1")
40     .parse()
41     .unwrap_or(1);
42
43 // MAIN FUNCTION FOLLOWS
44 // ...
45 }

```

This completes the step of ‘Args → File’, as well as giving us the other required parameters.

## 2.6.2. File to Input

Next, we must get access to the content of the file, and process that text into a serialised input.

```

1 fn main() {
2     // ...
3     // MAIN FUNCTION PRECEDES
4     match File::open(input_file) {
5         Ok(mut file) => {
6             let mut buf = String::new();
7             match file.read_to_string(&mut buf) {
8                 Ok(_) => {
9                     let input = text_to_input(&buf, input_file);
10                    // PROCESSING FOLLOWS
11                    // ...
12                }
13                Err(e) => hard_crash!(1, "Error reading `{}` :: `{}`",
↪ input_file, e),
14            }
15        }
16        Err(e) => {
17            hard_crash!(1, "Error opening `{}` :: `{}`", input_file, e);
18        }
19    }
20 }
21
22 /// Processes the input text file, turning it into an input serial object.

```

```

23 fn text_to_input(input: &str, input_file: &str) -> Input {
24     match toml::from_str(input) {
25         Ok(input) => input,
26         Err(e) => hard_crash!(1, "Error parsing `{}` :: `{}`", input_file,
    ↪ e),
27     }
28 }

```

This gives us access to input of type Input. Completing the ‘File → Input’ step of processing.

### 2.6.3. Input to Output

Next is the most involved step. Calculating the convex hulls and performing ‘Input → Output’. We can start by breaking this off simply as.

```

1 fn main() {
2     // ...
3     // MAIN FUNCTION PROCESSING PRECEDES
4     let output = input_to_output(&input);
5     // FURTHER MAIN PROCESSING FOLLOWS
6     // ...
7 }
8
9 /// Processes the input, converting it to the output.
10 fn input_to_output(input: &Input) -> Output {
11     process(input)
12 }

```

Which brings us to the task of implementing our convex hull finding algorithm. It may be useful here to reassess the program flow of this specific part of the problem.

Once again, I am going to take a back to front approach to assessing the program flow and constructs required.

The final output, should be of the form Output. This requires the computation of a set of hulls, and the forwarding of Input that is already available to us. ‘Hulls, Input → Output’

These hulls can be calculated by the quick-hull algorithm, which requires a set<sup>3</sup> of vertices and populates a set of hulls. ‘Vertices → Hull’

This requires us deciding which vertices will be our inputs. We are to be generating a set of hulls, one for each step of our way from start, through each route vertex, to end. Thus, we must figure out the vertices to include for each step.

---

<sup>3</sup>for both the vertex set and the segment set that constitutes our hull, the rust type HashSet<T, S> is used, which is a set of uniquely hashed elements.

We can gather an initial set of vertices by testing which polygons intersect with the trivial line from origin to destination on our currently checked route segment. However, this is not the end of the story. Once we have generated a hull for a polygon, we must then recalculate the input vertices for that route segment, as the hull may introduce a new polygon collision. If while this process is repeated, it occurs that no new polygons are intersected, we can say that all input points have been calculated. ‘Polygons, Hull, PathSegment → Vertices’

This provides a recurrent definition, whereby Hull is needed to construct the Vertices that are needed to construct a Hull. This has a well defined exit condition that should eventually be reached (as eventually, a path avoiding all polygon collision will be found).

An initial empty Hull can be used to begin this chain of generation.

The Polygons are provided by the ‘Input’.

The PathSegment can be iterated from the path vector, start, and finish provided by the Input.

This gives our sub program flow the following structure:

1. Input → Polygons
2. Input → PathSegments {
  3. Polygons, Hull, PathSegment → Vertices
  4. Vertices → Hull
5. } → Hull
6. Hulls, Input → Output

To begin to implement this, ‘Input → Polygons’ is a trivial accessor.

‘() → Hull’ needs to be computed to start calculation for each Hull of our Hulls. Fortunately this is simply an empty vector constructor.

### **Input -> PathSegments**

The simplest way to do this, is to set up an iterator that starts at the start vertex provided by the input, goes through each vertex of the route provided by the input, and then finally ends with the end vertex provided by the input. Beginning by leaving the origin uninitialised, and initialising the destination as the first point on this path. Each step of the iteration, we must set the new origin to be the old destination, and getting our new destination—ending the process if there is new step in the path (we have hit the end).

```
1 pub fn process(input: &Input) -> Output {
2     // ...
3     // PROCESS FUNCTION PRECEDES
4     let mut path = input.route.clone();
```

```

5 path.push(input.end);
6 path.insert(0, input.start);
7
8 let mut path = path.iter();
9
10 let mut origin;
11 let mut destination = path.next().unwrap(); // Guaranteed to have value
12 'generate_all_hulls: loop {
13     origin = destination;
14     let destination_o = path.next();
15     if destination_o.is_none() {
16         break 'generate_all_hulls;
17     }
18     destination = destination_o.unwrap();
19     // FURTHER PROCESS FOLLOWS
20     // ...
21 }
22 // PROCESS FUNCTION FOLLOWS
23 // ...

```

## Polygons, Hull, PathSegment to Vertices

Now, from our path segment, we can repeatedly get the intersecting polygons of each segment in our hull, and the initial path running from point point to point. Breaking if ever we do not add new polygons.

Calculating the intersections of lines involves computing the orientation of points. This will require defining a new Data Representation that was missed previously, `Orientation` that represents the state of three ordered sequential points being Clockwise, Counterclockwise, or Colinear with respect to each other.

```

1 pub fn process(input: &Input) -> Output {
2     // ...
3     // PROCESS FUNCTION PRECEDES
4     'generate_all_hulls: loop {
5         // ...
6         // FURTHER PROCESS PRECEDES
7         let mut hull: HashSet<Segment> = HashSet::new();
8         let mut final_polypoints;
9         'generate_hull: loop {
10            let mut polypoints = Segment::from_coords(*origin,
↪ *destination)
11                .get_intersecting_polygon_coords(&input.polygons);
12
13            for hull_segment in &hull {

```



```

14         let union = polypoints
15
16     ⇒ .union(&hull_segment.get_intersecting_polygon_coords(&input.polygons))
17         .cloned()
18         .collect();
19
20     polypoints = union;
21 }
22
23 if polypoints == union {
24     final_polypoints = polypoints.clone();
25     final_polypoints.insert(*origin);
26     final_polypoints.insert(*destination);
27     break 'generate_hull;
28 }
29
30 polypoints.insert(*origin);
31 polypoints.insert(*destination);
32 // FURTHER PROCESS FOLLOWS
33 // ...
34 }
35 // FURTHER PROCESS FOLLOWS
36 // ...
37 }
38
39 impl Segment {
40     /// Finds the coordinates of polygons that intersect the segment.
41     pub fn get_intersecting_polygon_coords(&self, polygons: &[Polygon]) ->
42     ⇒ HashSet<Coord> {
43         let mut intersecting_polygons = HashSet::new();
44         'p: for polygon in polygons.iter() {
45             for polygon_segment in polygon.segments() {
46                 if polygon_segment.intersects(self) {
47                     for coord in &polygon.points {
48                         intersecting_polygons.insert(coord.clone());
49                     }
50                     continue 'p;
51                 }
52             }
53         }
54         intersecting_polygons
55     }
56
57     /// Checks if self intersects another line segment.

```

```

57     pub fn intersects(&self, other: &Segment) -> bool {
58         if self.a == other.a || self.a == other.b || self.b == other.a ||
↪ self.b == other.b {
59             return false;
60         }
61
62         let o1 = Orientation::from_coords(self.a, self.b, other.a);
63         let o2 = Orientation::from_coords(self.a, self.b, other.b);
64         let o3 = Orientation::from_coords(other.a, other.b, self.a);
65         let o4 = Orientation::from_coords(other.a, other.b, self.b);
66
67         if o1 != o2 && o3 != o4 {
68             return true;
69         }
70
71         if o1.is_colinear() && Segment::from_coords(self.a,
↪ self.b).contains_colinear_coord(other.a)
72         {
73             return true;
74         }
75
76         if o2.is_colinear() && Segment::from_coords(self.a,
↪ self.b).contains_colinear_coord(other.b)
77         {
78             return true;
79         }
80
81         if o3.is_colinear()
82             && Segment::from_coords(other.a,
↪ other.b).contains_colinear_coord(self.a)
83         {
84             return true;
85         }
86
87         if o4.is_colinear()
88             && Segment::from_coords(other.a,
↪ other.b).contains_colinear_coord(self.b)
89         {
90             return true;
91         }
92
93         false
94     }
95
96     /// Checks if a point lies on self.

```

```

97     pub fn contains_colinear_coord(&self, coord: Coord) -> bool {
98         if Orientation::from_coords(self.a, self.b, coord).is_colinear() {
99             (coord.x <= max(self.a.x, self.b.x) && coord.x >= min(self.a.x,
↪ self.b.x)
100                 && coord.y <= max(self.a.y, self.b.y)
101                 && coord.y >= min(self.a.y, self.b.y))
102         } else {
103             false
104         }
105     }
106 }
107
108 /// Represents the orientation of three points.
109 pub enum Orientation {
110     /// Three points are colinear.
111     Colinear,
112
113     /// Three points are in clockwise orientation.
114     Clockwise,
115
116     /// Three points are in counterclockwise orientation.
117     Counterclockwise,
118 }
119 impl Orientation {
120     /// Computes an orientation from coordinates.
121     pub fn from_coords(p: Coord, q: Coord, r: Coord) -> Orientation {
122         match (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y) {
123             n if n < 0 => Orientation::Clockwise,
124             n if n > 0 => Orientation::Counterclockwise,
125             _ => Orientation::Colinear,
126         }
127     }
128
129     /// Returns true if Orientation is Colinear.
130     pub fn is_colinear(self) -> bool {
131         self == Orientation::Colinear
132     }
133 }

```

## Vertices to Hull (to Hulls)

Finally, we must convert our computed set of vertices to a hull. This will be done using the quick-hull algorithm.

```

1  /// Processes the input into it's output by generating the convex hulls.
2  pub fn process(input: &Input) -> Output {
3      let mut hulls = Vec::new();
4      // ...
5      'generate_all_hulls: loop {
6          // ...
7          'generate_hull: loop {
8              // ...
9              // FURTHER PROCESS PRECEDES
10             hull =
11     ↪ hull.union(&calculate_hull(&polypoints)).cloned().collect();
12             }
13             hull = calculate_hull(&final_polypoints);
14             hulls.push(Hull::from_segment_set(hull.into_iter().collect()));
15         }
16         // PROCESS FUNCTION FOLLOWS
17         // ...
18     }
19
20 /// Calculates the points that lie in the hull of a set of points.
21 pub fn calculate_hull<S: BuildHasher>(polypoints: &HashSet<Coord, S>)
22     ↪ -> HashSet<Segment> {
23     let mut hull = HashSet::new();
24
25     if !quick_hull(polypoints, &mut hull) {
26         panic!();
27     }
28
29     hull
30 }
31
32 /// Calculates the quick hull of a set of points, outputting it into a
33 ↪ buffer.
34 /// Returns true when computation is successful.
35 pub fn quick_hull<S1: BuildHasher, S2: BuildHasher>(
36     input: &HashSet<Coord, S1>,
37     hull: &mut HashSet<Segment, S2>,
38 ) -> bool {
39
40     if input.len() < 2 {
41         return false;
42     }
43
44     let (lefttest, righttest) = input.iter().fold(
45         (None, None),
46         |(mut lefttest, mut righttest), &item| {

```

```

43         if lefttest.is_none() {
44             lefttest = Some(item);
45         }
46         if righttest.is_none() {
47             righttest = Some(item);
48         }
49         if item.x < lefttest.unwrap().x {
50             lefttest = Some(item);
51         }
52         if item.x > righttest.unwrap().x {
53             righttest = Some(item);
54         }
55         (lefttest, righttest)
56     },
57 );
58 let lefttest = lefttest.unwrap();
59 let righttest = righttest.unwrap();
60
61 quick_hull_recurse(input, lefttest, righttest, Orientation::Clockwise,
↪ hull);
62 quick_hull_recurse(
63     input,
64     lefttest,
65     righttest,
66     Orientation::Counterclockwise,
67     hull,
68 );
69
70 true
71 }
72
73 /// The recursive call component of `quick_hull`.
74 fn quick_hull_recurse<S1: BuildHasher, S2: BuildHasher>(
75     input: &HashSet<Coord, S1>,
76     p1: Coord,
77     p2: Coord,
78     orientation: Orientation,
79     hull: &mut HashSet<Segment, S2>,
80 ) {
81     let mut divider: Option<Coord> = None;
82     let mut max_dist = 0;
83
84     for &coord in input.iter() {
85         let dist = Segment::from_coords(p1, p2).coord_distance(coord);

```

```

86         if Orientation::from_coords(p1, p2, coord) == orientation && dist >
↪     max_dist {
87             divider = Some(coord);
88             max_dist = dist;
89         }
90     }
91
92     if let Some(divider) = divider {
93         quick_hull_recurse(
94             input,
95             divider,
96             p1,
97             Orientation::from_coords(divider, p1, p2).invert(),
98             hull,
99         );
100        quick_hull_recurse(
101            input,
102            divider,
103            p2,
104            Orientation::from_coords(divider, p2, p1).invert(),
105            hull,
106        );
107    } else {
108        hull.insert(Segment::from_coords(p1, p2));
109    }
110 }
111
112 impl Segment {
113     /// Returns a value proportional to the distance between the line
↪     (extended
114     /// of the segment) and the points.
115     pub fn coord_distance(&self, other: Coord) -> i64 {
116         ((other.y - self.a.y) * (self.b.x - self.a.x)
117             - (self.b.y - self.a.y) * (other.x - self.a.x))
118             .abs()
119     }
120 }

```

## Hulls, Input to Output

And finally, a trivial step to bundle it all up and ship it back up to our old control flow.

```

1 /// Processes the input into it's output by generating the convex hulls.
2 pub fn process(input: &Input) -> Output {
3     // ...

```

```

4 // PROCESS FUNCTION PRECEDES
5 Output {
6     input: input.clone(),
7     hulls: hulls,
8 }
9 }

```

## 2.6.4. Output to File

Lastly, we have to process our newly generated output to create a file. There are a number of different file types we could have to generate, the generation of these is largely left to Serde, with the exception of our image processing.

The image processing simply goes through step by step, drawing in elements to the image. It simply places pixels for vertexes, and uses Bresenham's line rasterization algorithm for edges.

At this point our main function has suffered a great degree of rightward drift. I have backindented the excerpt of main below by 4 levels for reading convenience. The excerpt follows immediately after the call to `input_to_output`

```

1 fn main() {
2     // ...
3     // MAIN FUNCTION PROCESSING PRECEDES
4     match File::create(output_file) {
5         Ok(mut file) => {
6             if let Err(e) = match mode {
7                 "toml" => file.write(&output_to_toml(&output)),
8                 "json" => file.write(&output_to_json(&output)),
9                 "png" => file.write(&output_to_png(&output, scale)),
10                mode => hard_crash!(1, "Invalid output mode `{}`", mode),
11            } {
12                hard_crash!(1, "Error Writing to `{}` :: `{}`", output_file,
→ e);
13            }
14            if let Err(e) = file.flush() {
15                hard_crash!(1, "Error Flushing `{}` :: `{}`", output_file,
→ e);
16            }
17        }
18        Err(e) => {
19            hard_crash!(1, "Error Opening `{}` :: `{}`", output_file, e);
20        }
21    }
22    // MAIN FUNCTION PROCESSING FOLLOWS
23    // ...
24 }

```

```

25
26 /// Converts the output to a toml binary encoded text format.
27 fn output_to_toml(output: &Output) -> Vec<u8> {
28     format!("{}", toml::Value::try_from(output).unwrap()).into_bytes()
29 }
30
31 /// Converts the output to a json binary encoded text format.
32 fn output_to_json(output: &Output) -> Vec<u8> {
33     serde_json::to_string(output).unwrap().into_bytes()
34 }
35
36 /// Converts the output to a png binary format.
37 fn output_to_png(output: &Output, scale: u32) -> Vec<u8> {
38     let (x_size, y_size) = Some(output.input.start)
39         .into_iter()
40         .chain(Some(output.input.end).into_iter())
41         .chain(output.input.route.clone().into_iter())
42         .chain(
43             output
44                 .input
45                 .polygons
46                 .clone()
47                 .into_iter()
48                 .flat_map(|polygon| polygon.points.into_iter()),
49         )
50     .fold((0, 0), |(mut max_x, mut max_y), coord| {
51         if coord.x > max_x {
52             max_x = coord.x;
53         }
54         if coord.y > max_y {
55             max_y = coord.y;
56         }
57         (max_x, max_y)
58     });
59
60     let mut image: ImageBuffer<Rgb<u8>, Vec<u8>> =
61     ↪ ImageBuffer::from_pixel(
62         (x_size + 20) as u32,
63         (y_size + 20) as u32,
64         Rgb {
65             data: [255, 255, 255],
66         },
67     );
68     // Draw Polygons

```



```

69     for segment in output
70         .input
71         .polygons
72         .iter()
73         .flat_map(|polygon| polygon.segments())
74     {
75         bresenham_line(
76             segment.a.x,
77             segment.a.y,
78             segment.b.x,
79             segment.b.y,
80             &mut image,
81             10,
82             Rgb { data: [0, 0, 0] },
83         );
84     }
85
86     // Draw Hulls
87     for segment in output.hulls.iter().flat_map(|hull|
↪ hull.segment_set.iter()) {
88         bresenham_line(
89             segment.a.x,
90             segment.a.y,
91             segment.b.x,
92             segment.b.y,
93             &mut image,
94             10,
95             Rgb {
96                 data: [255, 0, 255],
97             },
98         );
99     }
100
101     // Draw Polypoints
102     for point in output
103         .input
104         .polygons
105         .iter()
106         .flat_map(|polygon| polygon.points.iter())
107     {
108         image.put_pixel(
109             point.x as u32 + 10,
110             point.y as u32 + 10,
111             Rgb { data: [0, 0, 255] },
112         );

```

```

113     }
114
115     // Draw Route
116     for point in &output.input.route {
117         image.put_pixel(
118             point.x as u32 + 10,
119             point.y as u32 + 10,
120             Rgb { data: [128, 0, 0] },
121         )
122     }
123
124     // Draw Start
125     image.put_pixel(
126         output.input.start.x as u32 + 10,
127         output.input.start.y as u32 + 10,
128         Rgb { data: [0, 255, 0] },
129     );
130
131     // Draw End
132     image.put_pixel(
133         output.input.end.x as u32 + 10,
134         output.input.end.y as u32 + 10,
135         Rgb { data: [255, 0, 0] },
136     );
137
138     // Scale image up (nearest neighbour)
139     let scaled_image: ImageBuffer<Rgb<u8>, Vec<u8>> =
140     ↪ ImageBuffer::from_fn(
141         (x_size + 20) as u32 * scale,
142         (y_size + 20) as u32 * scale,
143         |x, y| *image.get_pixel(x / scale, y / scale),
144     );
145
146     // Return png data
147     let mut buf = Vec::new();
148     PNGEncoder::new(Cursor::new(&mut buf))
149         .encode(
150             &scaled_image.into_vec(),
151             (x_size + 20) as u32 * scale,
152             (y_size + 20) as u32 * scale,
153             RGB(8),
154         )
155     .unwrap();
156     buf
157 }

```

```

157
158 /// Standard Bresenham Line Algorithm
159 fn bresenham_line<G, P>(mut x0: i64, mut y0: i64, x1: i64, y1: i64, g:
    ↪ &mut G, pad: u32, color: P)
160 where
161     G: GenericImage<Pixel = P>,
162     P: Pixel,
163 {
164     let dx = x1 - x0;
165     let sx = dx.signum();
166     let dx = dx.abs();
167
168     let dy = y1 - y0;
169     let sy = dy.signum();
170     let dy = dy.abs();
171
172     let mut err = if dx > dy { dx } else { -dy } / 2;
173
174     let mut e2;
175
176     loop {
177         g.put_pixel(x0 as u32 + pad, y0 as u32 + pad, color);
178         if x0 == x1 && y0 == y1 {
179             break;
180         }
181         e2 = err;
182         if e2 > -dx {
183             err -= dy;
184             x0 += sx;
185         }
186         if e2 < dy {
187             err += dx;
188             y0 += sy;
189         }
190     }
191 }

```

## 3. Tests

### 3.1. Minimal Input

A start point and an end point, no polygons.

```
[start]
```

```
x = 0
```

```
y = 0
```

```
[end]
```

```
x = 100
```

```
y = 0
```



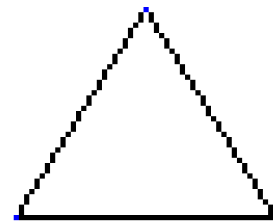
```
x = 75
```

```
y = 40
```

```
[[polygon.point]]
```

```
x = 50
```

```
y = 0
```



### 3.2. Non Intersecting

A start point, and an end point, with a polygon that does not prevent pathing.

```
[start]
```

```
x = 0
```

```
y = 50
```

```
[end]
```

```
x = 100
```

```
y = 50
```

```
[[polygon]]
```

```
[[polygon.point]]
```

```
x = 25
```

```
y = 40
```

```
[[polygon.point]]
```

### 3.3. Simple Intersecting

A start point, and an end point, with a polygon that prevents pathing.

```
[start]
```

```
x = 0
```

```
y = 0
```

```
[end]
```

```
x = 100
```

```
y = 50
```

```
[[polygon]]
```

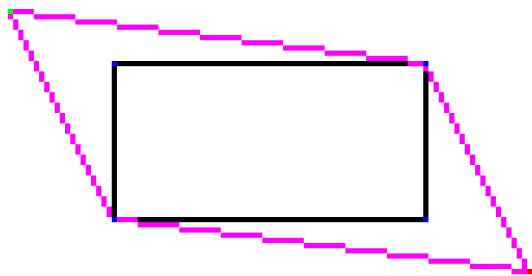
```
[[polygon.point]]
```

```
x = 20
```

```

y = 10
[[polygon.point]]
x = 80
y = 10
[[polygon.point]]
x = 80
y = 40
[[polygon.point]]
x = 20
y = 40

```



```

[[polygon.point]]
x = 20
y = 0
[[polygon.point]]
x = 70
y = 0
[[polygon.point]]
x = 70
y = 20
[[polygon.point]]
x = 20
y = 20

```

```

[[polygon]]
[[polygon.point]]
x = 80
y = 20
[[polygon.point]]
x = 80
y = 40
[[polygon.point]]
x = 100
y = 40
[[polygon.point]]
x = 100
y = 20

```

### 3.4. Path Route

A start point, a single route point, and an end point, with a polygon between each step.

```

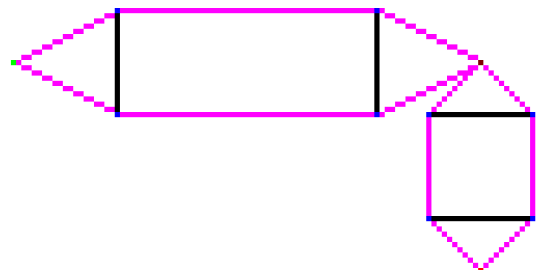
[start]
x = 0
y = 10

[end]
x = 90
y = 50

[[route]]
x = 90
y = 10

[[polygon]]

```



### 3.5. Example Task 1

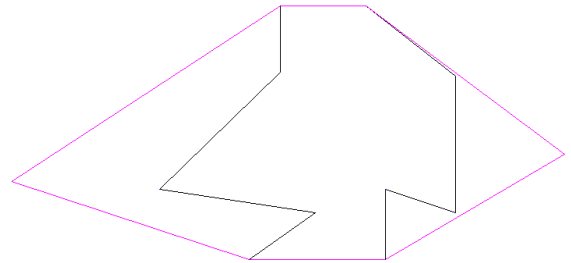
The example from task 1

```
[start]
x = 70
y = 240
```

```
[end]
x = 780
y = 205
```

```
[[polygon]]
  [[polygon.point]]
  x = 375
  y = 340
  [[polygon.point]]
  x = 550
  y = 340
  [[polygon.point]]
  x = 550
  y = 250
  [[polygon.point]]
  x = 640
  y = 280
  [[polygon.point]]
  x = 640
  y = 105
  [[polygon.point]]
  x = 525
  y = 15
  [[polygon.point]]
  x = 415
  y = 15
  [[polygon.point]]
  x = 415
  y = 100
  [[polygon.point]]
```

```
x = 260
y = 250
[[polygon.point]]
x = 460
y = 280
```



### 3.6. Example Task 2

The Example from task 2

```
[start]
x = 40
y = 280
```

```
[end]
x = 780
y = 218
```

```
[[route]]
x = 390
y = 260
```

```
[[polygon]]
  [[polygon.point]]
  x = 150
  y = 350
  [[polygon.point]]
  x = 250
  y = 350
  [[polygon.point]]
```

```

x = 250
y = 300
[[polygon.point]]
x = 300
y = 320
[[polygon.point]]
x = 300
y = 230
[[polygon.point]]
x = 240
y = 190
[[polygon.point]]
x = 175
y = 190
[[polygon.point]]
x = 175
y = 230
[[polygon.point]]
x = 90
y = 300
[[polygon.point]]
x = 200
y = 320
[[polygon]]
[[polygon.point]]
x = 525
y = 365
[[polygon.point]]
x = 625
y = 365
[[polygon.point]]
x = 625
y = 315
[[polygon.point]]
x = 675
y = 335
[[polygon.point]]
x = 675
y = 245
[[polygon.point]]
x = 615
y = 205
[[polygon.point]]
x = 550
y = 205
[[polygon.point]]
x = 550
y = 245
[[polygon.point]]
x = 465
y = 315
[[polygon.point]]
x = 575
y = 335
[[polygon]]
[[polygon.point]]
x = 507
y = 167
[[polygon.point]]
x = 607
y = 167
[[polygon.point]]
x = 607
y = 117
[[polygon.point]]
x = 657
y = 137
[[polygon.point]]
x = 657
y = 47
[[polygon.point]]
x = 597

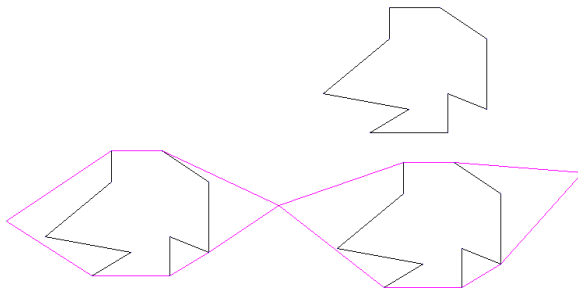
```

```
y = 7
[[polygon.point]]
x = 532
y = 7
[[polygon.point]]
x = 532
y = 47
[[polygon.point]]
x = 447
y = 117
[[polygon.point]]
x = 557
y = 137
```

```
x = 90
y = 50

[[route]]
x = 90
y = 10

[[polygon]]
[[polygon.point]]
x = 20
y = 0
[[polygon.point]]
x = 70
y = 0
[[polygon.point]]
x = 70
y = 20
[[polygon.point]]
x = 100
y = 20
[[polygon.point]]
x = 100
y = 40
[[polygon.point]]
x = 20
y = 40
```

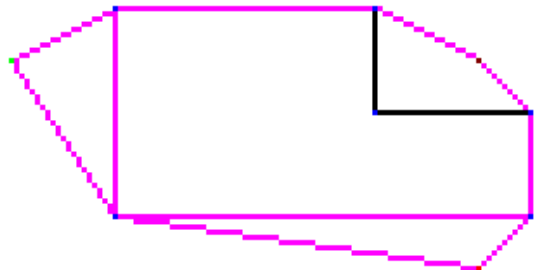


### 3.7. Two Paths one Polygon

A start point, a single route point, and an end point, with a polygon that blocks the path between both start and route, and end and route. Due to the nature of the specification, this generates a strange but ultimately correct path. The hulls from the route point to the end doubles back on the existing hull, routing through the polygon for its anticlockwise hull.

```
[start]
x = 0
y = 10

[end]
```





### 3.8. Arbitrary 15 Point Polygon

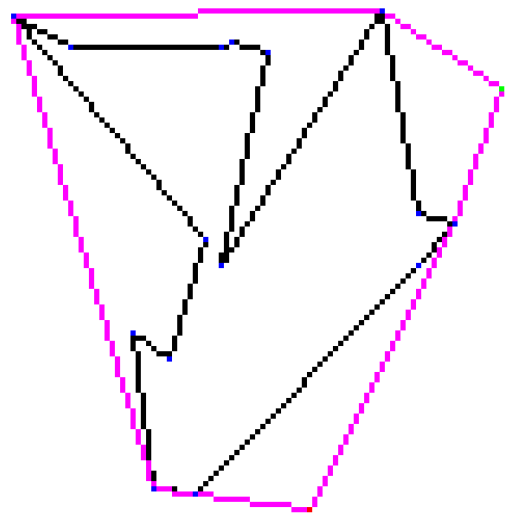
A 15 point polygon is generated from random coordinates. The path start and end are generated randomly too to be outside the polygon.

```
[start]  
x = 100  
y = 17
```

```
[end]  
x = 63  
y = 98
```

```
[[polygon]]  
  [[polygon.point]]  
  x = 6  
  y = 3  
  [[polygon.point]]  
  x = 17  
  y = 9  
  [[polygon.point]]  
  x = 46  
  y = 9  
  [[polygon.point]]  
  x = 48  
  y = 8  
  [[polygon.point]]  
  x = 55  
  y = 10  
  [[polygon.point]]  
  x = 46  
  y = 51  
  [[polygon.point]]  
  x = 77  
  y = 2
```

```
[[polygon.point]]  
x = 84  
y = 41  
[[polygon.point]]  
x = 91  
y = 43  
[[polygon.point]]  
x = 84  
y = 51  
[[polygon.point]]  
x = 41  
y = 95  
[[polygon.point]]  
x = 33  
y = 94  
[[polygon.point]]  
x = 29  
y = 64  
[[polygon.point]]  
x = 36  
y = 69  
[[polygon.point]]  
x = 43  
y = 46
```



### 3.9. Concave Escape

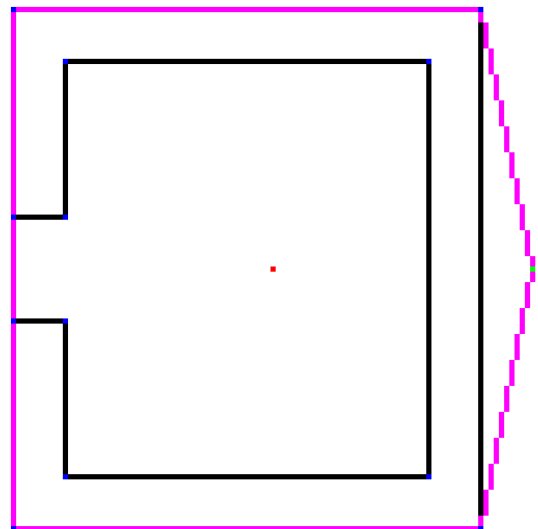
A concave polygon attempts to keep the path from escaping. Test fails as expected, path excludes point, no code was written to handle concave cases such as this.

```
[start]
x = 100
y = 50
```

```
[end]
x = 50
y = 50
```

```
[[polygon]]
  [[polygon.point]]
  x = 0
  y = 0
  [[polygon.point]]
  x = 90
  y = 0
  [[polygon.point]]
  x = 90
  y = 100
  [[polygon.point]]
  x = 0
  y = 100
  [[polygon.point]]
  x = 0
  y = 60
  [[polygon.point]]
  x = 10
  y = 60
  [[polygon.point]]
  x = 10
```

```
y = 90
[[polygon.point]]
x = 80
y = 90
[[polygon.point]]
x = 80
y = 10
[[polygon.point]]
x = 10
y = 10
[[polygon.point]]
x = 10
y = 40
[[polygon.point]]
x = 0
y = 40
```



### 3.10. Intersecting Polygons

Two polygons are constructed such that one is in the path from start to end, and one is not. However, the secondary one is intersected by the hull of the start, end, and intersecting polygon.

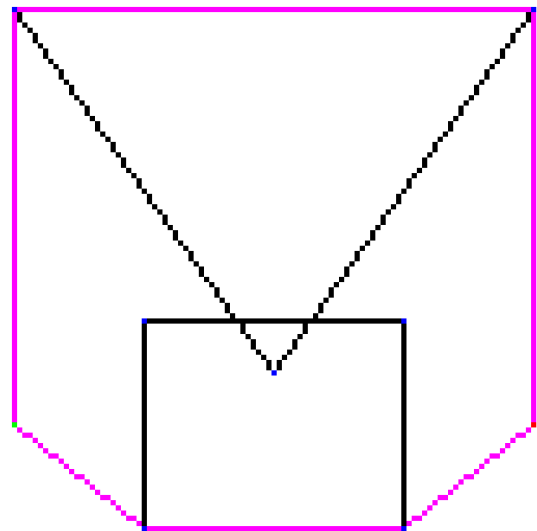
```
[start]
x = 0
y = 80
```

```
[end]
x = 100
y = 80
```

```
[[polygon]]
  [[polygon.point]]
    x = 25
    y = 60
  [[polygon.point]]
    x = 75
    y = 60
  [[polygon.point]]
    x = 75
    y = 100
  [[polygon.point]]
    x = 25
```

```
y = 100
```

```
[[polygon]]
  [[polygon.point]]
    x = 0
    y = 0
  [[polygon.point]]
    x = 100
    y = 0
  [[polygon.point]]
    x = 50
    y = 70
```



## 4. Conclusions & Reflections

The purpose of this report was to demonstrate the functionality of quick-hull and its applications into path finding.

I successfully implemented Quick-Hull, as well as another standard algorithm in Bresenham's Line Rasterization algorithm. In addition to this, the complexity of selecting polygons to include in the calculation when finding the convex hull was properly considered to deal with overlapping polygons.

I did not however consider the case of concave polygons, or polygons intersecting across multiple path segments. For the first case, the program simply cannot handle them and excludes the first point from the hull. The quick-hull algorithm could possibly be modified to always have to include certain points, and this would be an interesting extension for handling concave shapes. Polygons intersecting across multiple path segments behaves in a well defined, and somewhat correct way; it could be interesting to extend the program to not show paths that require doubling back over an already visited segment, this would eliminate the awkwardness of the second segment anti-oriented hull overlapping with the first segment oriented hull. This could be done by checking for hull inclusion in latter path calculations, and excluding that directional hull from the path origin in future instances. This may not however be a desirable change.

Overall, I believe that though there are better pathfinding algorithms in a number of situations, the application of convex hull algorithms to pathfinding facilitating arbitrary polygons is an interesting one. The algorithm was satisfying to implement, and the meta-program flow to solve some of the problems above was equally satisfying to figure out.

**Part II.**

**Appendix**

# List of Program Code

1.	/Cargo.toml . . . . .	38
2.	/src/lib.rs . . . . .	38
3.	/src/main.rs . . . . .	39
4.	/src/io/mod.rs . . . . .	46
5.	/src/io/input.rs . . . . .	46
6.	/src/io/output.rs . . . . .	46
7.	/src/process/mod.rs . . . . .	47
8.	/src/shape/mod.rs . . . . .	50
9.	/src/shape/coord.rs . . . . .	51
10.	/src/shape/hull.rs . . . . .	51
11.	/src/shape/orientation.rs . . . . .	52
12.	/src/shape/polygon.rs . . . . .	52
13.	/src/shape/segment.rs . . . . .	53
14.	/examples/simple.toml . . . . .	56
15.	/examples/task1.toml . . . . .	57
16.	/examples/task2.toml . . . . .	58

## Program Code 1: /Cargo.toml

```
1 [package]
2 name = "convex-hull-pf"
3 version = "0.1.0"
4 authors = ["Lucille Blumire <llblumire@gmail.com>"]
5
6 [dependencies]
7 clap = "*"
8 image = "*"
9 serde = "*"
10 serde_derive = "*"
11 toml = "*"
12 serde_json = "*"
```

## Program Code 2: /src/lib.rs

```

1  ///! Provides tools for calculating a path around a set of polygons through
   ↪ a set of points using
2  ///! convex hulls.
3
4  #![warn(missing_docs)]
5
6  #[macro_use]
7  extern crate serde_derive;
8
9  extern crate image;
10 extern crate serde;
11
12 pub mod io;
13 pub mod process;
14 pub mod shape;

```

Program Code 3: /src/main.rs

```

1  #[macro_use]
2  extern crate clap;
3
4  extern crate convex_hull_pf;
5  extern crate image;
6  extern crate serde_json;
7  extern crate toml;
8
9  use clap::{App, Arg};
10 use std::io::Read;
11 use std::fs::File;
12 use convex_hull_pf::io::input::Input;
13 use convex_hull_pf::io::output::Output;
14 use convex_hull_pf::process::process;
15 use std::io::Write;
16 use std::io::Cursor;
17 use image::ImageBuffer;
18 use image::GenericImage;
19 use image::png::PNGEncoder;
20 use image::RGB;
21 use image::Rgb;
22 use image::Pixel;
23
24 macro_rules! hard_crash {
25     ($code:expr, $($arg:tt)*) => {{
26         println!($($arg)*);
27         $crate::std::process::exit($code);

```

```

28     }}
29 }
30
31 fn main() {
32     let matches = App::new(crate_name!())
33         .version(crate_version!())
34         .author(crate_authors!())
35         .about("Finds a path along a route using a convex hull algorithm.")
36         .arg(
37             Arg::with_name("INPUT")
38                 .help("The input to process")
39                 .required(true)
40                 .index(1),
41         )
42         .arg(
43             Arg::with_name("OUTPUT")
44                 .help("The file to output to")
45                 .required(true)
46                 .index(2),
47         )
48         .arg(
49             Arg::with_name("output")
50                 .help("Specify the output mode, \"toml\" or \"json\" or
→ \"png\"")
51                 .short("o")
52                 .takes_value(true),
53         )
54         .arg(
55             Arg::with_name("output-scale")
56                 .help("Specify the output scale, only valid in \"png\" mode")
57                 .short("s")
58                 .takes_value(true),
59         )
60         .get_matches();
61
62     // Unwrap is safe as CLAP handles requirement of value.
63     let input_file = matches.value_of("INPUT").unwrap();
64     let output_file = matches.value_of("OUTPUT").unwrap();
65
66     let mode = matches.value_of("output").unwrap_or("toml");
67     let scale: u32 = matches
68         .value_of("output-scale")
69         .unwrap_or("1")
70         .parse()
71         .unwrap_or(1);

```



```

72
73 match File::open(input_file) {
74     Ok(mut file) => {
75         let mut buf = String::new();
76         match file.read_to_string(&mut buf) {
77             Ok(_) => {
78                 let input = text_to_input(&buf, input_file);
79                 let output = input_to_output(&input);
80                 match File::create(output_file) {
81                     Ok(mut file) => {
82                         if let Err(e) = match mode {
83                             "toml" =>
↪ file.write(&output_to_toml(&output)),
84                             "json" =>
↪ file.write(&output_to_json(&output)),
85                             "png" => file.write(&output_to_png(&output,
↪ scale)),
86                             mode => hard_crash!(1, "Invalid output mode
↪ `{}`", mode),
87                                 } {
88                                     hard_crash!(1, "Error Writing to `{}` ::
↪ `{}`", output_file, e);
89                                 }
90                                 if let Err(e) = file.flush() {
91                                     hard_crash!(1, "Error Flushing `{}` ::
↪ `{}`", output_file, e);
92                                 }
93                                 }
94                                 Err(e) => {
95                                     hard_crash!(1, "Error Opening `{}` :: `{}`",
↪ output_file, e);
96                                 }
97                             }
98                         }
99                         Err(e) => hard_crash!(1, "Error reading `{}` :: `{}`",
↪ input_file, e),
100                     }
101                 }
102                 Err(e) => {
103                     hard_crash!(1, "Error opening `{}` :: `{}`", input_file, e);
104                 }
105             }
106 }
107
108 /// Processes the input text file, turning it into an input serial object.

```

```

109 fn text_to_input(input: &str, input_file: &str) -> Input {
110     match toml::from_str(input) {
111         Ok(input) => input,
112         Err(e) => hard_crash!(1, "Error parsing `{}` :: `{}`", input_file,
113     ↪ e),
114     }
115 }
116 /// Processes the input, converting it to the output.
117 fn input_to_output(input: &Input) -> Output {
118     process(input)
119 }
120
121 /// Converts the output to a toml binary encoded text format.
122 fn output_to_toml(output: &Output) -> Vec<u8> {
123     format!("{}", toml::Value::try_from(output).unwrap()).into_bytes()
124 }
125
126 /// Converts the output to a json binary encoded text format.
127 fn output_to_json(output: &Output) -> Vec<u8> {
128     serde_json::to_string(output).unwrap().into_bytes()
129 }
130
131 /// Converts the output to a png binary format.
132 fn output_to_png(output: &Output, scale: u32) -> Vec<u8> {
133     let (x_size, y_size) = Some(output.input.start)
134         .into_iter()
135         .chain(Some(output.input.end).into_iter())
136         .chain(output.input.route.clone().into_iter())
137         .chain(
138             output
139                 .input
140                 .polygons
141                 .clone()
142                 .into_iter()
143                 .flat_map(|polygon| polygon.points.into_iter()),
144     )
145     .fold((0, 0), |(mut max_x, mut max_y), coord| {
146         if coord.x > max_x {
147             max_x = coord.x;
148         }
149         if coord.y > max_y {
150             max_y = coord.y;
151         }
152         (max_x, max_y)

```

```

153         });
154
155     let mut image: ImageBuffer<Rgb<u8>, Vec<u8>> =
↪ ImageBuffer::from_pixel(
156         (x_size + 20) as u32,
157         (y_size + 20) as u32,
158         Rgb {
159             data: [255, 255, 255],
160         },
161     );
162
163     // Draw Polygons
164     for segment in output
165         .input
166         .polygons
167         .iter()
168         .flat_map(|polygon| polygon.segments())
169     {
170         bresenham_line(
171             segment.a.x,
172             segment.a.y,
173             segment.b.x,
174             segment.b.y,
175             &mut image,
176             10,
177             Rgb { data: [0, 0, 0] },
178         );
179     }
180
181     // Draw Hulls
182     for segment in output.hulls.iter().flat_map(|hull|
↪ hull.segment_set.iter()) {
183         bresenham_line(
184             segment.a.x,
185             segment.a.y,
186             segment.b.x,
187             segment.b.y,
188             &mut image,
189             10,
190             Rgb {
191                 data: [255, 0, 255],
192             },
193         );
194     }
195

```

```

196 // Draw Polypoints
197 for point in output
198     .input
199     .polygons
200     .iter()
201     .flat_map(|polygon| polygon.points.iter())
202 {
203     image.put_pixel(
204         point.x as u32 + 10,
205         point.y as u32 + 10,
206         Rgb { data: [0, 0, 255] },
207     );
208 }
209
210 // Draw Route
211 for point in &output.input.route {
212     image.put_pixel(
213         point.x as u32 + 10,
214         point.y as u32 + 10,
215         Rgb { data: [128, 0, 0] },
216     )
217 }
218
219 // Draw Start
220 image.put_pixel(
221     output.input.start.x as u32 + 10,
222     output.input.start.y as u32 + 10,
223     Rgb { data: [0, 255, 0] },
224 );
225
226 // Draw End
227 image.put_pixel(
228     output.input.end.x as u32 + 10,
229     output.input.end.y as u32 + 10,
230     Rgb { data: [255, 0, 0] },
231 );
232
233 // Scale image up (nearest neighbour)
234 let scaled_image: ImageBuffer<Rgb<u8>, Vec<u8>> =
↪ ImageBuffer::from_fn(
235     (x_size + 20) as u32 * scale,
236     (y_size + 20) as u32 * scale,
237     |x, y| *image.get_pixel(x / scale, y / scale),
238 );
239

```

```

240 // Return png data
241 let mut buf = Vec::new();
242 PNGEncoder::new(Cursor::new(&mut buf))
243     .encode(
244         &scaled_image.into_vec(),
245         (x_size + 20) as u32 * scale,
246         (y_size + 20) as u32 * scale,
247         RGB(8),
248     )
249     .unwrap();
250 buf
251 }
252
253 /// Standard Bresenham Line Algorithm
254 fn bresenham_line<G, P>(mut x0: i64, mut y0: i64, x1: i64, y1: i64, g:
    → &mut G, pad: u32, color: P)
255 where
256     G: GenericImage<Pixel = P>,
257     P: Pixel,
258 {
259     let dx = x1 - x0;
260     let sx = dx.signum();
261     let dx = dx.abs();
262
263     let dy = y1 - y0;
264     let sy = dy.signum();
265     let dy = dy.abs();
266
267     let mut err = if dx > dy { dx } else { -dy } / 2;
268
269     let mut e2;
270
271     loop {
272         g.put_pixel(x0 as u32 + pad, y0 as u32 + pad, color);
273         if x0 == x1 && y0 == y1 {
274             break;
275         }
276         e2 = err;
277         if e2 > -dx {
278             err -= dy;
279             x0 += sx;
280         }
281         if e2 < dy {
282             err += dx;
283             y0 += sy;

```

```

284     }
285 }
286 }

```

#### Program Code 4: /src/io/mod.rs

```

1  /// Provides input and output serialization functionality.
2
3  pub mod input;
4  pub mod output;

```

#### Program Code 5: /src/io/input.rs

```

1  /// Provides the input struct.
2
3  use shape::coord::Coord;
4  use shape::polygon::Polygon;
5
6  /// The input for deserialization.
7  #[derive(Serialize, Deserialize, Debug, Clone)]
8  pub struct Input {
9      /// The start of the path.
10     pub start: Coord,
11
12     /// The end of the path.
13     pub end: Coord,
14
15     /// Points that must be passed in order from start to end.
16     #[serde(default = "Vec::new")]
17     pub route: Vec<Coord>,
18
19     /// The polygons that block the path.
20     #[serde(rename = "polygon", default = "Vec::new")]
21     pub polygons: Vec<Polygon>,
22 }

```

#### Program Code 6: /src/io/output.rs

```

1  /// Provides the Output struct.
2
3  use shape::hull::Hull;
4  use io::input::Input;
5
6  /// The Output of computation.

```

```

7  #[derive(Serialize, Deserialize, Debug)]
8  pub struct Output {
9      /// The input that generated this output, if known
10     pub input: Input,
11
12     /// The point to point hulls that make up the outputs along the path.
13     pub hulls: Vec<Hull>,
14 }

```

Program Code 7: /src/process/mod.rs

```

1  ///! Provides the process function, as well as housing the internals for
   ↪ computing convex hulls.
2
3  use io::input::Input;
4  use io::output::Output;
5  use shape::orientation::Orientation;
6  use shape::coord::Coord;
7  use shape::segment::Segment;
8  use std::collections::HashSet;
9  use shape::hull::Hull;
10 use std::hash::BuildHasher;
11
12 /// Processes the input into it's output by generating the convex hulls.
13 pub fn process(input: &Input) -> Output {
14     let mut hulls = Vec::new();
15
16     let mut path = input.route.clone();
17     path.push(input.end);
18     path.insert(0, input.start);
19
20     let mut path = path.iter();
21
22     let mut origin;
23     let mut destination = path.next().unwrap(); // Guaranteed to have value
24     'generate_all_hulls: loop {
25         origin = destination;
26         let destination_o = path.next();
27         if destination_o.is_none() {
28             break 'generate_all_hulls;
29         }
30         destination = destination_o.unwrap();
31
32         let mut hull: HashSet<Segment> = HashSet::new();
33         let mut final_polypoints;

```

```

34     'generate_hull: loop {
35         let mut polypoints = Segment::from_coords(*origin,
↪ *destination)
36             .get_intersecting_polygon_coords(&input.polygons);
37
38         for hull_segment in &hull {
39             let union = polypoints
40
↪ .union(&hull_segment.get_intersecting_polygon_coords(&input.polygons))
41                 .cloned()
42                 .collect();
43         }
44         if polypoints == union {
45             final_polypoints = polypoints.clone();
46             final_polypoints.insert(*origin);
47             final_polypoints.insert(*destination);
48             break 'generate_hull;
49         }
50
51         polypoints = union;
52
53
54         polypoints.insert(*origin);
55         polypoints.insert(*destination);
56
57         hull =
↪ hull.union(&calculate_hull(&polypoints)).cloned().collect();
58     }
59     hull = calculate_hull(&final_polypoints);
60     hulls.push(Hull::from_segment_set(hull.into_iter().collect()));
61 }
62
63 Output {
64     input: input.clone(),
65     hulls: hulls,
66 }
67 }
68
69 /// Calculates the points that lie in the hull of a set of points.
70 pub fn calculate_hull<S: BuildHasher>(polypoints: &HashSet<Coord, S>)
↪ -> HashSet<Segment> {
71     let mut hull = HashSet::new();
72
73     if !quick_hull(polypoints, &mut hull) {
74         panic!();

```



```

75     }
76
77     hull
78 }
79
80 /// Calculates the quick hull of a set of points, outputting it into a
81 ⇒ buffer.
82 /// Returns true when computation is successful.
83 pub fn quick_hull<S1: BuildHasher, S2: BuildHasher>(
84     input: &HashSet<Coord, S1>,
85     hull: &mut HashSet<Segment, S2>,
86 ) -> bool {
87     if input.len() < 2 {
88         return false;
89     }
90
91     let (lefttest, righttest) = input.iter().fold(
92         (None, None),
93         |(mut lefttest, mut righttest), &item| {
94             if lefttest.is_none() {
95                 lefttest = Some(item);
96             }
97             if righttest.is_none() {
98                 righttest = Some(item);
99             }
100            if item.x < lefttest.unwrap().x {
101                lefttest = Some(item);
102            }
103            if item.x > righttest.unwrap().x {
104                righttest = Some(item);
105            }
106            (lefttest, righttest)
107        },
108    );
109
110     let lefttest = lefttest.unwrap();
111     let righttest = righttest.unwrap();
112
113     quick_hull_recurse(input, lefttest, righttest, Orientation::Clockwise,
114     ⇒ hull);
115     quick_hull_recurse(
116         input,
117         lefttest,
118         righttest,
119         Orientation::Counterclockwise,
120         hull,

```

```

118     );
119
120     true
121 }
122
123 /// The recursive call component of `quick_hull`.
124 fn quick_hull_recurse<S1: BuildHasher, S2: BuildHasher>(
125     input: &HashSet<Coord, S1>,
126     p1: Coord,
127     p2: Coord,
128     orientation: Orientation,
129     hull: &mut HashSet<Segment, S2>,
130 ) {
131     let mut divider: Option<Coord> = None;
132     let mut max_dist = 0;
133
134     for &coord in input.iter() {
135         let dist = Segment::from_coords(p1, p2).coord_distance(coord);
136         if Orientation::from_coords(p1, p2, coord) == orientation && dist >
→ max_dist {
137             divider = Some(coord);
138             max_dist = dist;
139         }
140     }
141
142     if let Some(divider) = divider {
143         quick_hull_recurse(
144             input,
145             divider,
146             p1,
147             Orientation::from_coords(divider, p1, p2).invert(),
148             hull,
149         );
150         quick_hull_recurse(
151             input,
152             divider,
153             p2,
154             Orientation::from_coords(divider, p2, p1).invert(),
155             hull,
156         );
157     } else {
158         hull.insert(Segment::from_coords(p1, p2));
159     }
160 }

```

### Program Code 8: /src/shape/mod.rs

```
1  /// Provides tools for manipulating shapes and relationships in the  
    → cartesian plane.  
2  
3  pub mod coord;  
4  pub mod polygon;  
5  pub mod segment;  
6  pub mod orientation;  
7  pub mod hull;
```

### Program Code 9: /src/shape/coord.rs

```
1  /// Provides the Coord struct.  
2  
3  /// A coordinate in the cartesian plane.  
4  #[derive(Serialize, Deserialize, Debug, Copy, Clone, Eq, PartialEq, Hash)]  
5  pub struct Coord {  
6      /// The x coordinate.  
7      pub x: i64,  
8      /// The y coordinate.  
9      pub y: i64,  
10 }  
}
```

### Program Code 10: /src/shape/hull.rs

```
1  /// Provides the Hull struct.  
2  
3  use shape::segment::Segment;  
4  
5  /// Represents a Convex Hull  
6  #[derive(Serialize, Deserialize, Debug)]  
7  pub struct Hull {  
8      /// The segments that constitute a hull.  
9      pub segment_set: Vec<Segment>,  
10 }  
11  
12 impl Hull {  
13     /// Constructs a hull from it's segments.  
14     pub fn from_segment_set(segment_set: Vec<Segment>) -> Hull {  
15         Hull { segment_set }  
16     }  
17 }
```

Program Code 11: /src/shape/orientation.rs

```

1  /// Provides the Orientation enum.
2  use shape::coord::Coord;
3
4  /// Represents the orientation of three points.
5  #[derive(Copy, Clone, Debug, Eq, PartialEq)]
6  pub enum Orientation {
7      /// Three points are colinear.
8      Colinear,
9
10     /// Three points are in clockwise orientation.
11     Clockwise,
12
13     /// Three points are in counterclockwise orientation.
14     Counterclockwise,
15 }
16 impl Orientation {
17     /// Computes an orientation from coordinates.
18     pub fn from_coords(p: Coord, q: Coord, r: Coord) -> Orientation {
19         match (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y) {
20             n if n < 0 => Orientation::Clockwise,
21             n if n > 0 => Orientation::Counterclockwise,
22             _ => Orientation::Colinear,
23         }
24     }
25
26     /// Returns true if Orientation is Colinear.
27     pub fn is_colinear(self) -> bool {
28         self == Orientation::Colinear
29     }
30
31     /// Returns the opposite orientation, Clockwise becomes
32     → Counterclockwise, Counterclockwise
33     /// becomes Clockwise, and Colinear stays the same.
34     pub fn invert(self) -> Self {
35         match self {
36             Orientation::Colinear => Orientation::Colinear,
37             Orientation::Clockwise => Orientation::Counterclockwise,
38             Orientation::Counterclockwise => Orientation::Clockwise,
39         }
40     }

```

Program Code 12: /src/shape/polygon.rs

```

1  /// Provides the Polygon struct.
2
3  use shape::coord::Coord;
4  use shape::segment::Segment;
5
6  /// Represents a polygon.
7  #[derive(Serialize, Deserialize, Debug, Eq, PartialEq, Clone, Hash)]
8  pub struct Polygon {
9      /// The set a points that make up the polygon, ordered counterclockwise.
10     #[serde(rename = "point")]
11     pub points: Vec<Coord>,
12 }
13
14 impl Polygon {
15     /// Returns a vector of the segments that make up the polygon.
16     pub fn segments(&self) -> Vec<Segment> {
17         let cycleiter =
18     ↪ self.points.iter().chain(self.points.iter().take(1));
19         let cycleiter2 =
20     ↪ self.points.iter().chain(self.points.iter().take(2)).skip(1);
21         let edges = cycleiter
22             .zip(cycleiter2)
23             .map(|(&a, &b)| Segment::from_coords(a, b));
24         edges.collect:::<Vec<_>>()
25     }
26 }

```

Program Code 13: /src/shape/segment.rs

```

1  /// Provides the Segment struct.
2
3  use shape::coord::Coord;
4  use std::cmp::max;
5  use std::cmp::min;
6  use shape::orientation::Orientation;
7  use std::collections::HashSet;
8  use shape::polygon::Polygon;
9  use std::cmp::Ordering;
10
11 /// Represents a line segment AB.
12 #[derive(Serialize, Deserialize, Debug, Copy, Clone, Eq, PartialEq, Hash)]
13 pub struct Segment {
14     /// The left start of the line segment.

```

```

15     pub a: Coord,
16
17     /// The right end of the line segment.
18     pub b: Coord,
19 }
20 impl Segment {
21     /// Constructs a line segment from it's coordinates AB.
22     pub fn from_coords(a: Coord, b: Coord) -> Segment {
23         let (a, b) = match (a.x.cmp(&b.x), a.y.cmp(&b.y)) {
24             (Ordering::Less, _) | (Ordering::Equal, Ordering::Less) => (a,
↪ b),
25             _ => (b, a),
26         };
27         Segment { a, b }
28     }
29
30     /// Checks if a point lies on self.
31     pub fn contains_colinear_coord(&self, coord: Coord) -> bool {
32         if Orientation::from_coords(self.a, self.b, coord).is_colinear() {
33             (coord.x <= max(self.a.x, self.b.x) && coord.x >= min(self.a.x,
↪ self.b.x)
34                 && coord.y <= max(self.a.y, self.b.y)
35                 && coord.y >= min(self.a.y, self.b.y))
36         } else {
37             false
38         }
39     }
40
41     /// Checks if self intersects another line segment.
42     pub fn intersects(&self, other: &Segment) -> bool {
43         if self.a == other.a || self.a == other.b || self.b == other.a ||
↪ self.b == other.b {
44             return false;
45         }
46
47         let o1 = Orientation::from_coords(self.a, self.b, other.a);
48         let o2 = Orientation::from_coords(self.a, self.b, other.b);
49         let o3 = Orientation::from_coords(other.a, other.b, self.a);
50         let o4 = Orientation::from_coords(other.a, other.b, self.b);
51
52         if o1 != o2 && o3 != o4 {
53             return true;
54         }
55

```

```

56         if o1.is_colinear() && Segment::from_coords(self.a,
↪ self.b).contains_colinear_coord(other.a)
57         {
58             return true;
59         }
60
61         if o2.is_colinear() && Segment::from_coords(self.a,
↪ self.b).contains_colinear_coord(other.b)
62         {
63             return true;
64         }
65
66         if o3.is_colinear()
67             && Segment::from_coords(other.a,
↪ other.b).contains_colinear_coord(self.a)
68         {
69             return true;
70         }
71
72         if o4.is_colinear()
73             && Segment::from_coords(other.a,
↪ other.b).contains_colinear_coord(self.b)
74         {
75             return true;
76         }
77
78         false
79     }
80
81     /// Returns a value proportional to the distance between the line
↪ (extended
82 /// of the segment) and the points.
83     pub fn coord_distance(&self, other: Coord) -> i64 {
84         ((other.y - self.a.y) * (self.b.x - self.a.x)
85          - (self.b.y - self.a.y) * (other.x - self.a.x))
86         .abs()
87     }
88
89     /// Finds the polygons that intersect with a segment.
90     pub fn get_intersecting_polygons(&self, polygons: &[Polygon]) ->
↪ HashSet<Polygon> {
91         let mut intersecting_polygons = HashSet::new();
92         'p: for polygon in polygons.iter() {
93             for polygon_segment in polygon.segments() {
94                 if polygon_segment.intersects(self) {

```

```

95             intersecting_polygons.insert(polygon.clone());
96             continue 'p;
97         }
98     }
99 }
100 intersecting_polygons
101 }
102
103 /// Finds the coordinates of polygons that intersect the segment.
104 pub fn get_intersecting_polygon_coords(&self, polygons: &[Polygon]) ->
↪ HashSet<Coord> {
105     let mut intersecting_polygons = HashSet::new();
106     'p: for polygon in polygons.iter() {
107         for polygon_segment in polygon.segments() {
108             if polygon_segment.intersects(self) {
109                 for coord in &polygon.points {
110                     intersecting_polygons.insert(coord.clone());
111                 }
112                 continue 'p;
113             }
114         }
115     }
116     intersecting_polygons
117 }
118 }

```

Program Code 14: /examples/simple.toml

[start]

x = 70

y = 70

[end]

x = 190

y = 70

[[polygon]]

[[polygon.point]]

x = 130

y = 40

[[polygon.point]]



```
x = 150
```

```
y = 90
```

```
[[polygon.point]]
```

```
x = 110
```

```
y = 90
```

```
[[polygon.point]]
```

```
x = 130
```

```
y = 70
```

Program Code 15: /examples/task1.toml

```
[start]
```

```
x = 70
```

```
y = 240
```

```
[end]
```

```
x = 780
```

```
y = 205
```

```
[[polygon]]
```

```
[[polygon.point]]
```

```
x = 375
```

```
y = 340
```

```
[[polygon.point]]
```

```
x = 550
```

```
y = 340
```

```
[[polygon.point]]
```

```
x = 550
```

```
y = 250
```

```
[[polygon.point]]
```

```
x = 640
```

```
y = 280
```

```
[[polygon.point]]
```

```
x = 640
```

```
y = 105
```

```
[[polygon.point]]
```

```
x = 525
y = 15
[[polygon.point]]
x = 415
y = 15
[[polygon.point]]
x = 415
y = 100
[[polygon.point]]
x = 260
y = 250
[[polygon.point]]
x = 460
y = 280
```

Program Code 16: /examples/task2.toml

```
[start]
x = 40
y = 280

[end]
x = 780
y = 218

[[route]]
x = 390
y = 260

[[polygon]]
  [[polygon.point]]
  x = 150
  y = 350
  [[polygon.point]]
  x = 250
  y = 350
  [[polygon.point]]
  x = 250
```

```
y = 300
[[polygon.point]]
x = 300
y = 320
[[polygon.point]]
x = 300
y = 230
[[polygon.point]]
x = 240
y = 190
[[polygon.point]]
x = 175
y = 190
[[polygon.point]]
x = 175
y = 230
[[polygon.point]]
x = 90
y = 300
[[polygon.point]]
x = 200
y = 320
```

```
[[polygon]]
[[polygon.point]]
x = 525
y = 365
[[polygon.point]]
x = 625
y = 365
[[polygon.point]]
x = 625
y = 315
[[polygon.point]]
x = 675
y = 335
[[polygon.point]]
```

```
x = 675
y = 245
[[polygon.point]]
x = 615
y = 205
[[polygon.point]]
x = 550
y = 205
[[polygon.point]]
x = 550
y = 245
[[polygon.point]]
x = 465
y = 315
[[polygon.point]]
x = 575
y = 335
```

```
[[polygon]]
[[polygon.point]]
x = 507
y = 167
[[polygon.point]]
x = 607
y = 167
[[polygon.point]]
x = 607
y = 117
[[polygon.point]]
x = 657
y = 137
[[polygon.point]]
x = 657
y = 47
[[polygon.point]]
x = 597
y = 7
```

```
[[polygon.point]]  
x = 532  
y = 7  
[[polygon.point]]  
x = 532  
y = 47  
[[polygon.point]]  
x = 447  
y = 117  
[[polygon.point]]  
x = 557  
y = 137
```