CS2AO17 Convex Hull

L. L. Blumire

January 21, 2018

Contents

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.

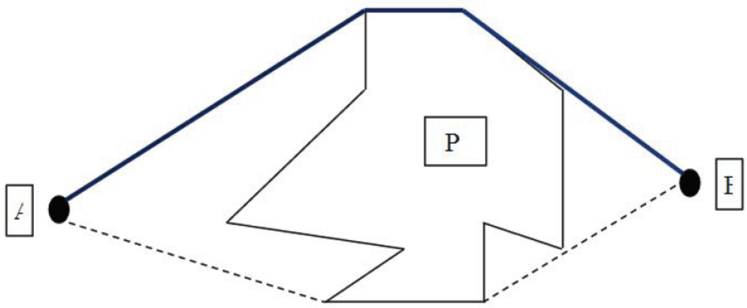


Figure 1.1.: Task 1

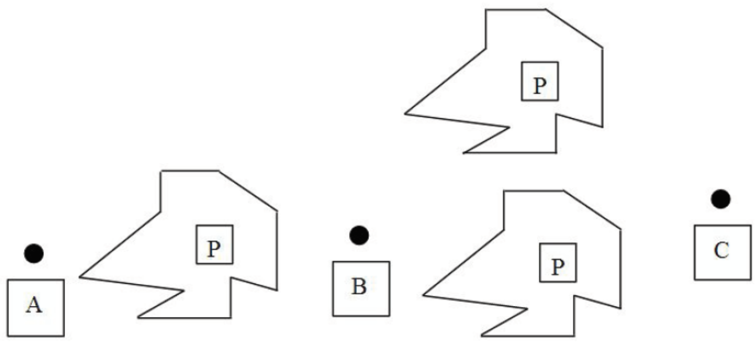


Figure 1.2.: Task 2

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 Choise

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 guarantied 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 langauges 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 it's unnecesary 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:

```rust
/// A coordinate in the cartesian plane.
pub struct Coord {
    /// The x coordinate.
    pub x: i64,
    /// The y coordinate.
    pub y: i64,
}
```

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.

```rust
/// Represents a line segment AB.
pub struct Segment {
    /// The left start of the line segment.
    pub a: Coord,

    /// The right end of the line segment.
    pub b: Coord,
}
```

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[1] 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.

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

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.

```
/// Represents a Convex Hull
pub struct Hull {
    /// The segments that constitute a hull.
    pub segment_set: Vec&lt;Segment&gt;,
}
```

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 Ouput 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.

[1]A contiguous growable array type, written `Vec<T>` but pronounced 'vector'.

- Polygons.

As such, the following definition would fit these requirements.

```rust
/// The input for deserialization.
pub struct Input {
    /// The start of the path.
    pub start: Coord,

    /// The end of the path.
    pub end: Coord,

    /// Points that must be passed in order from start to end.
    pub route: Vec<Coord>,

    /// The polygons that block the path.
    pub polygons: Vec<Polygon>,
}
```

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[2] the input, and in addition the hulls.

```rust
/// The Output of computation.
pub struct Output {
    /// The input that generated this output, if known
    pub input: Input,

    /// The point to point hulls that make up the outputs along the path.
    pub hulls: Vec<Hull>,
}
```

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.

[2]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`.

```rust
impl Segment {
    /// Constructs a line segment from it's coordinates AB.
    pub fn from_coords(a: Coord, b: Coord) -> Segment {
        let (a, b) = match (a.x.cmp(&b.x), a.y.cmp(&b.y)) {
            (Ordering::Less, _) | (Ordering::Equal, Ordering::Less) => (a,
    ↪  b),
            _ => (b, a),
        };
        Segment { a, b }
    }
}
```

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

```rust
impl Hull {
    /// Constructs a hull from it's segements.
    pub fn from_segment_set(segment_set: Vec&lt;Segment&gt;) -> Hull {
        Hull { segment_set }
    }
}
```

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

```rust
// ...
```

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.

```rust
fn main() {
    let matches = App::new(crate_name!())
        .version(crate_version!())
        .author(crate_authors!())
        .about("Finds a path along a route using a convex hull algorithm.")
        .arg(
            Arg::with_name("INPUT")
                .help("The input to process")
                .required(true)
                .index(1),
        )
        .arg(
            Arg::with_name("OUTPUT")
                .help("The file to output to")
                .required(true)
                .index(2),
        )
        .arg(
            Arg::with_name("output")
                .help("Specify the output mode, \"toml\" or \"json\" or \"png\"")
                .short("o")
                .takes_value(true),
        )
        .arg(
            Arg::with_name("output-scale")
              .help("Specify the output scale, only valid in \"png\" mode")
                .short("s")
                .takes_value(true),
        )
        .get_matches();
```

```rust

    // Unwrap is safe as CLAP handles requirement of value.
    let input_file = matches.value_of("INPUT").unwrap();
    let output_file = matches.value_of("OUTPUT").unwrap();

    let mode = matches.value_of("output").unwrap_or("toml");
    let scale: u32 = matches
        .value_of("output-scale")
        .unwrap_or("1")
        .parse()
        .unwrap_or(1);

    // MAIN FUNCTION FOLLOWS
    // ...
}
```

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.

```rust
fn main() {
    // ...
    // MAIN FUNCTION PRECEDES
    match File::open(input_file) {
        Ok(mut file) => {
            let mut buf = String::new();
            match file.read_to_string(&mut buf) {
                Ok(_) => {
                    let input = text_to_input(&buf, input_file);
                    // PROCESSING FOLLOWS
                    // ...
                }
                Err(e) => hard_crash!(1, "Error reading `{}` :: `{}`", input_file, e),
            }
        }
        Err(e) => {
            hard_crash!(1, "Error opening `{}` :: `{}`", input_file, e);
        }
    }
}

/// Processes the input text file, turning it into an input serial object.
```

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

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.

```
fn main() {
                    // ...
                    // MAIN FUNCTION PROCESSING PRECEDES
                    let output = input_to_output(&input);
                    // FURTHER MAIN PROCESSING FOLLOWS
                    // ...
}

/// Processes the input, converting it to the output.
fn input_to_output(input: &Input) -> Output {
    process(input)
}
```

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[3] 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.

[3] 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 the 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).

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

```
    path.push(input.end);
    path.insert(0, input.start);

    let mut path = path.iter();

    let mut origin;
   let mut destination = path.next().unwrap(); // Guarunteed to have value
    'generate_all_hulls: loop {
        origin = destination;
        let destination_o = path.next();
        if destination_o.is_none() {
            break 'generate_all_hulls;
        }
        destination = destination_o.unwrap();
        // FURTHER PROCESS FOLLOWS
        // ...
    }
    // PROCESS FUNCTION FOLLOWS
    // ...
```

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.

```
pub fn process(input: &Input) -> Output {
    // ...
    // PROCESS FUNCTION PRECEDES
    'generate_all_hulls: loop {
        // ...
        // FURTHER PROCESS PRECEDES
        let mut hull: HashSet&lt;Segment&gt; = HashSet::new();
        let mut final_polypoints;
        'generate_hull: loop {
            let mut polypoints = Segment::from_coords(*origin,
 ↪  *destination)
                .get_intersecting_polygon_coords(&input.polygons);

            for hull_segment in &hull {
```

```rust
                let union = polypoints
                    .union(&hull_segment.get_intersecting_polygon_coords(&input.polygons))
                    .cloned()
                    .collect();

                polypoints = union;
            }

            if polypoints == union {
                final_polypoints = polypoints.clone();
                final_polypoints.insert(*origin);
                final_polypoints.insert(*destination);
                break 'generate_hull;
            }

            polypoints.insert(*origin);
            polypoints.insert(*destination);
            // FURTHER PROCESS FOLLOWS
            // ...
        }
        // FURTHER PROCESS FOLLOWS
        // ...
    }
}

impl Segment {
    /// Finds the coordinates of polygons that intersect the segment.
    pub fn get_intersecting_polygon_coords(&self, polygons: &[Polygon]) -> HashSet<Coord> {
        let mut intersecting_polygons = HashSet::new();
        'p: for polygon in polygons.iter() {
            for polygon_segment in polygon.segments() {
                if polygon_segment.intersects(self) {
                    for coord in &polygon.points {
                        intersecting_polygons.insert(coord.clone());
                    }
                    continue 'p;
                }
            }
        }
        intersecting_polygons
    }

    /// Checks if self intersects another line segment.
```

```
    pub fn intersects(&self, other: &Segment) -> bool {
        if self.a == other.a || self.a == other.b || self.b == other.a || self.b == other.b {
            return false;
        }

        let o1 = Orientation::from_coords(self.a, self.b, other.a);
        let o2 = Orientation::from_coords(self.a, self.b, other.b);
        let o3 = Orientation::from_coords(other.a, other.b, self.a);
        let o4 = Orientation::from_coords(other.a, other.b, self.b);

        if o1 != o2 && o3 != o4 {
            return true;
        }

        if o1.is_colinear() && Segment::from_coords(self.a, self.b).contains_colinear_coord(other.a)
        {
            return true;
        }

        if o2.is_colinear() && Segment::from_coords(self.a, self.b).contains_colinear_coord(other.b)
        {
            return true;
        }

        if o3.is_colinear()
            && Segment::from_coords(other.a, other.b).contains_colinear_coord(self.a)
        {
            return true;
        }

        if o4.is_colinear()
            && Segment::from_coords(other.a, other.b).contains_colinear_coord(self.b)
        {
            return true;
        }

        false
    }

    /// Checks if a point lies on self.
```

```
    pub fn contains_colinear_coord(&self, coord: Coord) -> bool {
        if Orientation::from_coords(self.a, self.b, coord).is_colinear() {
            (coord.x <= max(self.a.x, self.b.x) && coord.x >= min(self.a.x,
↪  self.b.x)
                && coord.y <= max(self.a.y, self.b.y)
                && coord.y >= min(self.a.y, self.b.y))
        } else {
            false
        }
    }
}

/// Represents the orientation of three points.
pub enum Orientation {
    /// Three points are colinear.
    Colinear,

    /// Three points are in clockwise orientation.
    Clockwise,

    /// Three points are in counterclockwise orientation.
    Counterclockwise,
}
impl Orientation {
    /// Computes an orientation from coordinates.
    pub fn from_coords(p: Coord, q: Coord, r: Coord) -> Orientation {
        match (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y) {
            n if n < 0 => Orientation::Clockwise,
            n if n > 0 => Orientation::Counterclockwise,
            _ => Orientation::Colinear,
        }
    }

    /// Returns true if Orientation is Colinear.
    pub fn is_colinear(self) -> bool {
        self == Orientation::Colinear
    }
}
```

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.

```rust
/// Processes the input into it's output by generating the convex hulls.
pub fn process(input: &Input) -> Output {
    let mut hulls = Vec::new();
    // ...
    'generate_all_hulls: loop {
        // ...
        'generate_hull: loop {
            // ...
            // FURTHER PROCESS PRECEDES
            hull = hull.union(&calculate_hull(&polypoints)).cloned().collect();
        }
        hull = calculate_hull(&final_polypoints);
        hulls.push(Hull::from_segment_set(hull.into_iter().collect()));
    }
    // PROCESS FUNCTION FOLLOWS
    // ...
}

/// Calculates the points that lie in the hull of a set of points.
pub fn calculate_hull<S: BuildHasher>(polypoints: &HashSet<Coord, S>) -> HashSet&lt;Segment&gt; {
    let mut hull = HashSet::new();

    if !quick_hull(polypoints, &mut hull) {
        panic!();
    }

    hull
}

/// Calculates the quick hull of a set of points, outputting it into a buffer.
/// Returns true when computation is successful.
pub fn quick_hull<S1: BuildHasher, S2: BuildHasher>(
    input: &HashSet<Coord, S1>,
    hull: &mut HashSet&lt;Segment, S2&gt;,
) -> bool {
    if input.len() < 2 {
        return false;
    }

    let (leftest, rightest) = input.iter().fold(
        (None, None),
        |(mut leftest, mut rightest), &item| {
```

```
            if leftest.is_none() {
                leftest = Some(item);
            }
            if rightest.is_none() {
                rightest = Some(item);
            }
            if item.x < leftest.unwrap().x {
                leftest = Some(item);
            }
            if item.x > rightest.unwrap().x {
                rightest = Some(item);
            }
            (leftest, rightest)
        },
    );
    let leftest = leftest.unwrap();
    let rightest = rightest.unwrap();

    quick_hull_recurse(input, leftest, rightest, Orientation::Clockwise, hull);
    quick_hull_recurse(
        input,
        leftest,
        rightest,
        Orientation::Counterclockwise,
        hull,
    );

    true
}

/// The recursive call component of `quick_hull`.
fn quick_hull_recurse<S1: BuildHasher, S2: BuildHasher>(
    input: &HashSet<Coord, S1>,
    p1: Coord,
    p2: Coord,
    orientation: Orientation,
    hull: &mut HashSet&lt;Segment, S2&gt;,
) {
    let mut divider: Option<Coord> = None;
    let mut max_dist = 0;

    for &coord in input.iter() {
        let dist = Segment::from_coords(p1, p2).coord_distance(coord);
```

```
        if Orientation::from_coords(p1, p2, coord) == orientation && dist > max_dist {
            divider = Some(coord);
            max_dist = dist;
        }
    }

    if let Some(divider) = divider {
        quick_hull_recurse(
            input,
            divider,
            p1,
            Orientation::from_coords(divider, p1, p2).invert(),
            hull,
        );
        quick_hull_recurse(
            input,
            divider,
            p2,
            Orientation::from_coords(divider, p2, p1).invert(),
            hull,
        );
    } else {
        hull.insert(Segment::from_coords(p1, p2));
    }
}

impl Segment {
    /// Returns a value proportional to the distance between the line (extended
    /// of the segment) and the points.
    pub fn coord_distance(&self, other: Coord) -> i64 {
        ((other.y - self.a.y) * (self.b.x - self.a.x)
            - (self.b.y - self.a.y) * (other.x - self.a.x))
            .abs()
    }
}
```

Hulls, Input to Output

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

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

```
    // PROCESS FUNCTION PRECEDES
    Output {
        input: input.clone(),
        hulls: hulls,
    }
}
```

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`

```
fn main() {
    // ...
    // MAIN FUNCTION PROCESSING PRECEDES
    match File::create(output_file) {
        Ok(mut file) => {
            if let Err(e) = match mode {
                "toml" => file.write(&output_to_toml(&output)),
                "json" => file.write(&output_to_json(&output)),
                "png" => file.write(&output_to_png(&output, scale)),
                mode => hard_crash!(1, "Invalid output mode `{}`", mode),
            } {
               hard_crash!(1, "Error Writing to `{}` :: `{}`", output_file,
↪ e);
            }
            if let Err(e) = file.flush() {
                hard_crash!(1, "Error Flushing `{}` :: `{}`", output_file,
↪ e);
            }
        }
        Err(e) => {
            hard_crash!(1, "Error Opening `{}` :: `{}`", output_file, e);
        }
    }
    // MAIN FUNCTION PROCESSING FOLLOWS
    // ...
}
```

```
/// Converts the output to a toml binary encoded text format.
fn output_to_toml(output: &Output) -> Vec<u8> {
    format!("{}", toml::Value::try_from(output).unwrap()).into_bytes()
}

/// Converts the output to a json binary encoded text format.
fn output_to_json(output: &Output) -> Vec<u8> {
    serde_json::to_string(output).unwrap().into_bytes()
}

/// Converts the output to a png binary format.
fn output_to_png(output: &Output, scale: u32) -> Vec<u8> {
    let (x_size, y_size) = Some(output.input.start)
        .into_iter()
        .chain(Some(output.input.end).into_iter())
        .chain(output.input.route.clone().into_iter())
        .chain(
            output
                .input
                .polygons
                .clone()
                .into_iter()
                .flat_map(|polygon| polygon.points.into_iter()),
        )
        .fold((0, 0), |(mut max_x, mut max_y), coord| {
            if coord.x > max_x {
                max_x = coord.x;
            }
            if coord.y > max_y {
                max_y = coord.y;
            }
            (max_x, max_y)
        });

    let mut image: ImageBuffer<Rgb<u8>, Vec<u8>> =
    ↪  ImageBuffer::from_pixel(
        (x_size + 20) as u32,
        (y_size + 20) as u32,
        Rgb {
            data: [255, 255, 255],
        },
    );

    // Draw Polygons
```

```
    for segment in output
        .input
        .polygons
        .iter()
        .flat_map(|polygon| polygon.segments())
    {
        bresenham_line(
            segment.a.x,
            segment.a.y,
            segment.b.x,
            segment.b.y,
            &mut image,
            10,
            Rgb { data: [0, 0, 0] },
        );
    }

    // Draw Hulls
    for segment in output.hulls.iter().flat_map(|hull|
 ↪  hull.segment_set.iter()) {
        bresenham_line(
            segment.a.x,
            segment.a.y,
            segment.b.x,
            segment.b.y,
            &mut image,
            10,
            Rgb {
                data: [255, 0, 255],
            },
        );
    }

    // Draw Polypoints
    for point in output
        .input
        .polygons
        .iter()
        .flat_map(|polygon| polygon.points.iter())
    {
        image.put_pixel(
            point.x as u32 + 10,
            point.y as u32 + 10,
            Rgb { data: [0, 0, 255] },
        );
```

```
    }

    // Draw Route
    for point in &output.input.route {
        image.put_pixel(
            point.x as u32 + 10,
            point.y as u32 + 10,
            Rgb { data: [128, 0, 0] },
        )
    }

    // Draw Start
    image.put_pixel(
        output.input.start.x as u32 + 10,
        output.input.start.y as u32 + 10,
        Rgb { data: [0, 255, 0] },
    );

    // Draw End
    image.put_pixel(
        output.input.end.x as u32 + 10,
        output.input.end.y as u32 + 10,
        Rgb { data: [255, 0, 0] },
    );

    // Scale image up (nearest neighbour)
    let scaled_image: ImageBuffer<Rgb<u8>, Vec<u8>> = ImageBuffer::from_fn(
        (x_size + 20) as u32 * scale,
        (y_size + 20) as u32 * scale,
        |x, y| *image.get_pixel(x / scale, y / scale),
    );

    // Return png data
    let mut buf = Vec::new();
    PNGEncoder::new(Cursor::new(&mut buf))
        .encode(
            &scaled_image.into_vec(),
            (x_size + 20) as u32 * scale,
            (y_size + 20) as u32 * scale,
            RGB(8),
        )
        .unwrap();
    buf
}
```

```rust
/// Standard Bresenham Line Algorithm
fn bresenham_line<G, P>(mut x0: i64, mut y0: i64, x1: i64, y1: i64, g:
    &mut G, pad: u32, color: P)
where
    G: GenericImage<Pixel = P>,
    P: Pixel,
{
    let dx = x1 - x0;
    let sx = dx.signum();
    let dx = dx.abs();

    let dy = y1 - y0;
    let sy = dy.signum();
    let dy = dy.abs();

    let mut err = if dx > dy { dx } else { -dy } / 2;

    let mut e2;

    loop {
        g.put_pixel(x0 as u32 + pad, y0 as u32 + pad, color);
        if x0 == x1 && y0 == y1 {
            break;
        }
        e2 = err;
        if e2 > -dx {
            err -= dy;
            x0 += sx;
        }
        if e2 < dy {
            err += dx;
            y0 += sy;
        }
    }
}
```

3. Tests

3.1. Minimal Input

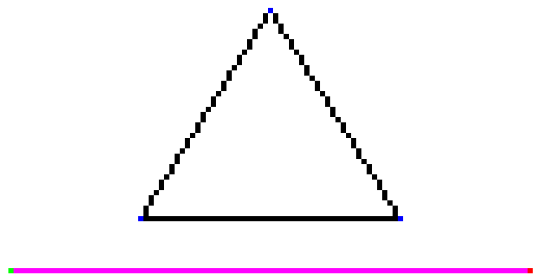
A start point and an end point, no polygons.

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

[end]
x = 100
y = 0
```


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]]
x = 75
y = 40
[[polygon.point]]
x = 50
y = 0
```

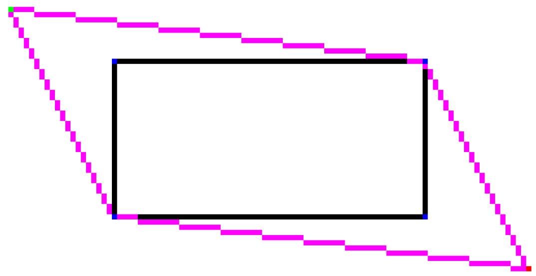


3.3. Simple Intersecting

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
```

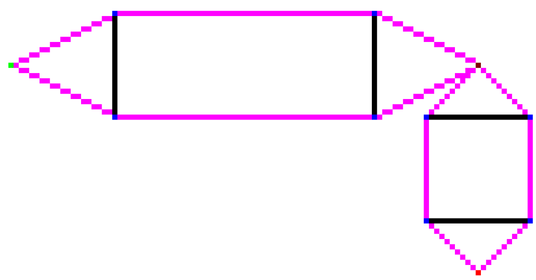
```
    y = 10
    [[polygon.point]]
    x = 80
    y = 10
    [[polygon.point]]
    x = 80
    y = 40
    [[polygon.point]]
    x = 20
    y = 40
```



3.4. Path Route

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

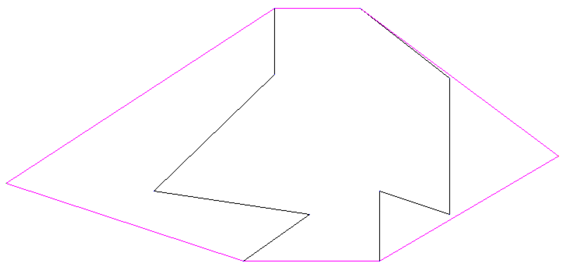
```
[start]
x = 0
y = 10

[end]
x = 90
y = 50

[[route]]
x = 90
y = 10

[[polygon]]
    [[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
```



3.5. 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
```



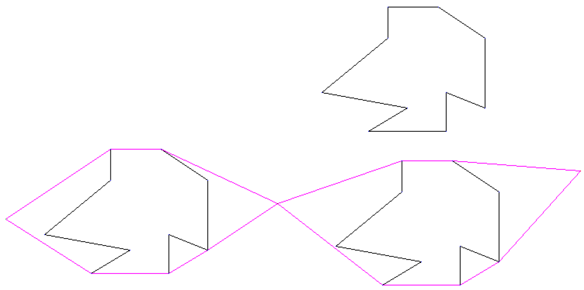
3.6. Example Task 2

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
```

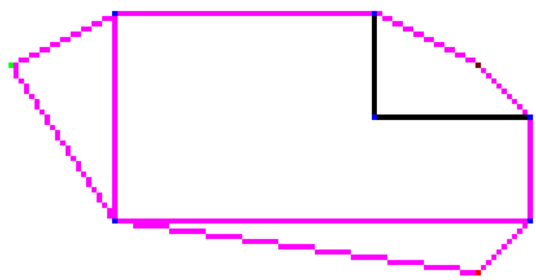
```
    y =  7
    [[polygon.point]]
    x =  532
    y =  7
    [[polygon.point]]
    x =  532
    y =  47
    [[polygon.point]]
    x =  447
    y =  117
    [[polygon.point]]
    x =  557
    y =  137
```



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.

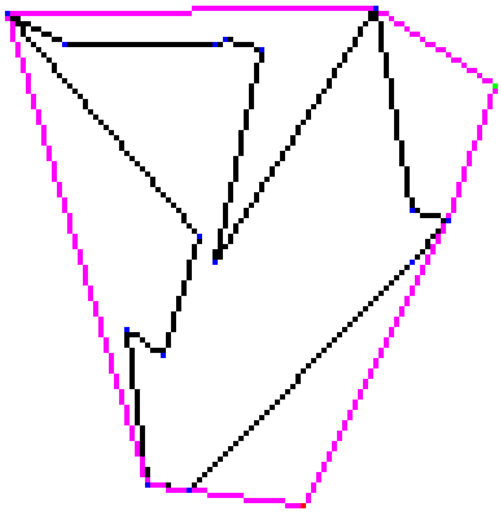
```
[start]
x = 0
y = 10

[end]
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
```



3.8. 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.

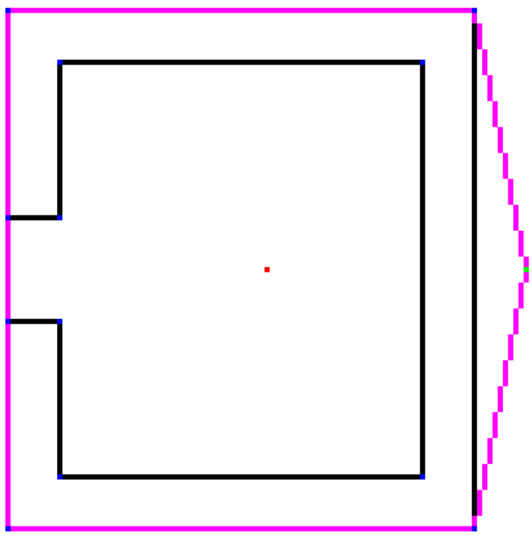
```
[start]
x = 100
y = 17

[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.

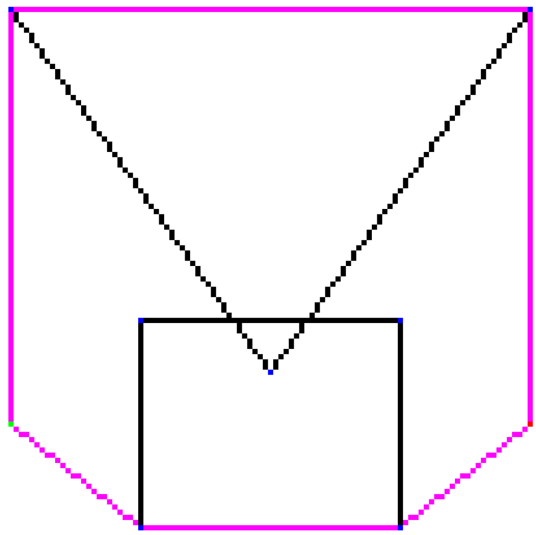
```
[start]
x = 100
y = 50

[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 it's 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

Program Code 1: /Cargo.toml

```
[package]
name = "convex-hull-pf"
version = "0.1.0"
authors = ["Lucille Blumire <llblumire@gmail.com>"]

[dependencies]
clap = "*"
image = "*"
serde = "*"
serde_derive = "*"
toml = "*"
serde_json = "*"
```

Program Code 2: /src/lib.rs

```rust
//! Provides tools for calculating a path around a set of polygons through
    ↪  a set of points using
//! convex hulls.

#![warn(missing_docs)]

#[macro_use]
extern crate serde_derive;

extern crate image;
extern crate serde;

pub mod io;
pub mod process;
pub mod shape;
```

Program Code 3: /src/main.rs

```rust
#[macro_use]
extern crate clap;

extern crate convex_hull_pf;
extern crate image;
extern crate serde_json;
extern crate toml;

use clap::{App, Arg};
use std::io::Read;
use std::fs::File;
use convex_hull_pf::io::input::Input;
use convex_hull_pf::io::output::Output;
use convex_hull_pf::process::process;
use std::io::Write;
use std::io::Cursor;
use image::ImageBuffer;
use image::GenericImage;
use image::png::PNGEncoder;
use image::RGB;
use image::Rgb;
use image::Pixel;

macro_rules! hard_crash {
    ($code:expr, $($arg:tt)*) => {{
        println!($($arg)*);
        $crate::std::process::exit($code);
```

```rust
    }}
}

fn main() {
    let matches = App::new(crate_name!())
        .version(crate_version!())
        .author(crate_authors!())
        .about("Finds a path along a route using a convex hull algorithm.")
        .arg(
            Arg::with_name("INPUT")
                .help("The input to process")
                .required(true)
                .index(1),
        )
        .arg(
            Arg::with_name("OUTPUT")
                .help("The file to output to")
                .required(true)
                .index(2),
        )
        .arg(
            Arg::with_name("output")
                .help("Specify the output mode, \"toml\" or \"json\" or \"png\"")
                .short("o")
                .takes_value(true),
        )
        .arg(
            Arg::with_name("output-scale")
               .help("Specify the output scale, only valid in \"png\" mode")
                .short("s")
                .takes_value(true),
        )
        .get_matches();

    // Unwrap is safe as CLAP handles requirement of value.
    let input_file = matches.value_of("INPUT").unwrap();
    let output_file = matches.value_of("OUTPUT").unwrap();

    let mode = matches.value_of("output").unwrap_or("toml");
    let scale: u32 = matches
        .value_of("output-scale")
        .unwrap_or("1")
        .parse()
        .unwrap_or(1);
```

```
    match File::open(input_file) {
        Ok(mut file) => {
            let mut buf = String::new();
            match file.read_to_string(&mut buf) {
                Ok(_) => {
                    let input = text_to_input(&buf, input_file);
                    let output = input_to_output(&input);
                    match File::create(output_file) {
                        Ok(mut file) => {
                            if let Err(e) = match mode {
                                "toml" => file.write(&output_to_toml(&output)),
                                "json" => file.write(&output_to_json(&output)),
                                "png" => file.write(&output_to_png(&output, scale)),
                                mode => hard_crash!(1, "Invalid output mode `{}`", mode),
                            } {
                                hard_crash!(1, "Error Writing to `{}` :: `{}`", output_file, e);
                            }
                            if let Err(e) = file.flush() {
                                hard_crash!(1, "Error Flushing `{}` :: `{}`", output_file, e);
                            }
                        }
                        Err(e) => {
                            hard_crash!(1, "Error Opening `{}` :: `{}`", output_file, e);
                        }
                    }
                }
                Err(e) => hard_crash!(1, "Error reading `{}` :: `{}`", input_file, e),
            }
        }
        Err(e) => {
            hard_crash!(1, "Error opening `{}` :: `{}`", input_file, e);
        }
    }
}

/// Processes the input text file, turning it into an input serial object.
```

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

/// Processes the input, converting it to the output.
fn input_to_output(input: &Input) -> Output {
    process(input)
}

/// Converts the output to a toml binary encoded text format.
fn output_to_toml(output: &Output) -> Vec<u8> {
    format!("{}", toml::Value::try_from(output).unwrap()).into_bytes()
}

/// Converts the output to a json binary encoded text format.
fn output_to_json(output: &Output) -> Vec<u8> {
    serde_json::to_string(output).unwrap().into_bytes()
}

/// Converts the output to a png binary format.
fn output_to_png(output: &Output, scale: u32) -> Vec<u8> {
    let (x_size, y_size) = Some(output.input.start)
        .into_iter()
        .chain(Some(output.input.end).into_iter())
        .chain(output.input.route.clone().into_iter())
        .chain(
            output
                .input
                .polygons
                .clone()
                .into_iter()
                .flat_map(|polygon| polygon.points.into_iter()),
        )
        .fold((0, 0), |(mut max_x, mut max_y), coord| {
            if coord.x > max_x {
                max_x = coord.x;
            }
            if coord.y > max_y {
                max_y = coord.y;
            }
            (max_x, max_y)
```

```rust
        });

    let mut image: ImageBuffer<Rgb<u8>, Vec<u8>> = ImageBuffer::from_pixel(
        (x_size + 20) as u32,
        (y_size + 20) as u32,
        Rgb {
            data: [255, 255, 255],
        },
    );

    // Draw Polygons
    for segment in output
        .input
        .polygons
        .iter()
        .flat_map(|polygon| polygon.segments())
    {
        bresenham_line(
            segment.a.x,
            segment.a.y,
            segment.b.x,
            segment.b.y,
            &mut image,
            10,
            Rgb { data: [0, 0, 0] },
        );
    }

    // Draw Hulls
    for segment in output.hulls.iter().flat_map(|hull| hull.segment_set.iter()) {
        bresenham_line(
            segment.a.x,
            segment.a.y,
            segment.b.x,
            segment.b.y,
            &mut image,
            10,
            Rgb {
                data: [255, 0, 255],
            },
        );
    }

```

```
    // Draw Polypoints
    for point in output
        .input
        .polygons
        .iter()
        .flat_map(|polygon| polygon.points.iter())
    {
        image.put_pixel(
            point.x as u32 + 10,
            point.y as u32 + 10,
            Rgb { data: [0, 0, 255] },
        );
    }

    // Draw Route
    for point in &output.input.route {
        image.put_pixel(
            point.x as u32 + 10,
            point.y as u32 + 10,
            Rgb { data: [128, 0, 0] },
        )
    }

    // Draw Start
    image.put_pixel(
        output.input.start.x as u32 + 10,
        output.input.start.y as u32 + 10,
        Rgb { data: [0, 255, 0] },
    );

    // Draw End
    image.put_pixel(
        output.input.end.x as u32 + 10,
        output.input.end.y as u32 + 10,
        Rgb { data: [255, 0, 0] },
    );

    // Scale image up (nearest neighbour)
    let scaled_image: ImageBuffer<Rgb<u8>, Vec<u8>> = ImageBuffer::from_fn(
        (x_size + 20) as u32 * scale,
        (y_size + 20) as u32 * scale,
        |x, y| *image.get_pixel(x / scale, y / scale),
    );

```

```
    // Return png data
    let mut buf = Vec::new();
    PNGEncoder::new(Cursor::new(&mut buf))
        .encode(
            &scaled_image.into_vec(),
            (x_size + 20) as u32 * scale,
            (y_size + 20) as u32 * scale,
            RGB(8),
        )
        .unwrap();
    buf
}

/// Standard Bresenham Line Algorithm
fn bresenham_line<G, P>(mut x0: i64, mut y0: i64, x1: i64, y1: i64, g: &mut G, pad: u32, color: P)
where
    G: GenericImage<Pixel = P>,
    P: Pixel,
{
    let dx = x1 - x0;
    let sx = dx.signum();
    let dx = dx.abs();

    let dy = y1 - y0;
    let sy = dy.signum();
    let dy = dy.abs();

    let mut err = if dx > dy { dx } else { -dy } / 2;

    let mut e2;

    loop {
        g.put_pixel(x0 as u32 + pad, y0 as u32 + pad, color);
        if x0 == x1 && y0 == y1 {
            break;
        }
        e2 = err;
        if e2 > -dx {
            err -= dy;
            x0 += sx;
        }
        if e2 < dy {
            err += dx;
            y0 += sy;
```

```
        }
    }
}
```

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

```
//! Provides input and output serialization functionality.

pub mod input;
pub mod output;
```

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

```
//! Provides the input struct.

use shape::coord::Coord;
use shape::polygon::Polygon;

/// The input for deserialization.
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct Input {
    /// The start of the path.
    pub start: Coord,

    /// The end of the path.
    pub end: Coord,

    /// Points that must be passed in order from start to end.
    #[serde(default = "Vec::new")]
    pub route: Vec<Coord>,

    /// The polygons that block the path.
    #[serde(rename = "polygon", default = "Vec::new")]
    pub polygons: Vec<Polygon>,
}
```

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

```
//! Provides the Output struct.

use shape::hull::Hull;
use io::input::Input;

/// The Output of computation.
```

```rust
#[derive(Serialize, Deserialize, Debug)]
pub struct Output {
    /// The input that generated this output, if known
    pub input: Input,

    /// The point to point hulls that make up the outputs along the path.
    pub hulls: Vec<Hull>,
}
```

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

```rust
//! Provides the process function, as well as housing the internals for
    ↪ computing convex hulls.

use io::input::Input;
use io::output::Output;
use shape::orientation::Orientation;
use shape::coord::Coord;
use shape::segment::Segment;
use std::collections::HashSet;
use shape::hull::Hull;
use std::hash::BuildHasher;

/// Processes the input into it's output by generating the convex hulls.
pub fn process(input: &Input) -> Output {
    let mut hulls = Vec::new();

    let mut path = input.route.clone();
    path.push(input.end);
    path.insert(0, input.start);

    let mut path = path.iter();

    let mut origin;
    let mut destination = path.next().unwrap(); // Guarunteed to have value
    'generate_all_hulls: loop {
        origin = destination;
        let destination_o = path.next();
        if destination_o.is_none() {
            break 'generate_all_hulls;
        }
        destination = destination_o.unwrap();

        let mut hull: HashSet&lt;Segment&gt; = HashSet::new();
        let mut final_polypoints;
```

```rust
        'generate_hull: loop {
            let mut polypoints = Segment::from_coords(*origin, *destination)
                .get_intersecting_polygon_coords(&input.polygons);

            for hull_segment in &hull {
                let union = polypoints
                    .union(&hull_segment.get_intersecting_polygon_coords(&input.polygons))
                    .cloned()
                    .collect();
            }
            if polypoints == union {
                final_polypoints = polypoints.clone();
                final_polypoints.insert(*origin);
                final_polypoints.insert(*destination);
                break 'generate_hull;
            }

            polypoints = union;


            polypoints.insert(*origin);
            polypoints.insert(*destination);

            hull = hull.union(&calculate_hull(&polypoints)).cloned().collect();
        }
        hull = calculate_hull(&final_polypoints);
        hulls.push(Hull::from_segment_set(hull.into_iter().collect()));
    }

    Output {
        input: input.clone(),
        hulls: hulls,
    }
}

/// Calculates the points that lie in the hull of a set of points.
pub fn calculate_hull<S: BuildHasher>(polypoints: &HashSet<Coord, S>) -> HashSet&lt;Segment&gt; {
    let mut hull = HashSet::new();

    if !quick_hull(polypoints, &mut hull) {
        panic!();
```

```rust
    }

    hull
}

/// Calculates the quick hull of a set of points, outputting it into a buffer.
/// Returns true when computation is successful.
pub fn quick_hull<S1: BuildHasher, S2: BuildHasher>(
    input: &HashSet<Coord, S1>,
    hull: &mut HashSet&lt;Segment, S2&gt;,
) -> bool {
    if input.len() < 2 {
        return false;
    }

    let (leftest, rightest) = input.iter().fold(
        (None, None),
        |(mut leftest, mut rightest), &item| {
            if leftest.is_none() {
                leftest = Some(item);
            }
            if rightest.is_none() {
                rightest = Some(item);
            }
            if item.x < leftest.unwrap().x {
                leftest = Some(item);
            }
            if item.x > rightest.unwrap().x {
                rightest = Some(item);
            }
            (leftest, rightest)
        },
    );
    let leftest = leftest.unwrap();
    let rightest = rightest.unwrap();

    quick_hull_recurse(input, leftest, rightest, Orientation::Clockwise, hull);
    quick_hull_recurse(
        input,
        leftest,
        rightest,
        Orientation::Counterclockwise,
        hull,
```

```rust
    );

    true
}

/// The recursive call component of `quick_hull`.
fn quick_hull_recurse<S1: BuildHasher, S2: BuildHasher>(
    input: &HashSet<Coord, S1>,
    p1: Coord,
    p2: Coord,
    orientation: Orientation,
    hull: &mut HashSet&lt;Segment, S2&gt;,
) {
    let mut divider: Option<Coord> = None;
    let mut max_dist = 0;

    for &coord in input.iter() {
        let dist = Segment::from_coords(p1, p2).coord_distance(coord);
        if Orientation::from_coords(p1, p2, coord) == orientation && dist >
        ↪ max_dist {
            divider = Some(coord);
            max_dist = dist;
        }
    }

    if let Some(divider) = divider {
        quick_hull_recurse(
            input,
            divider,
            p1,
            Orientation::from_coords(divider, p1, p2).invert(),
            hull,
        );
        quick_hull_recurse(
            input,
            divider,
            p2,
            Orientation::from_coords(divider, p2, p1).invert(),
            hull,
        );
    } else {
        hull.insert(Segment::from_coords(p1, p2));
    }
}
```

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

```
//! Provides tools for manipulating shapes and relationships in the cartesian plane.

pub mod coord;
pub mod polygon;
pub mod segment;
pub mod orientation;
pub mod hull;
```

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

```
//! Provides the Coord struct.

/// A coordinate in the cartesian plane.
#[derive(Serialize, Deserialize, Debug, Copy, Clone, Eq, PartialEq, Hash)]
pub struct Coord {
    /// The x coordinate.
    pub x: i64,
    /// The y coordinate.
    pub y: i64,
}
```

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

```
//! Provides the Hull struct.

use shape::segment::Segment;

/// Represents a Convex Hull
#[derive(Serialize, Deserialize, Debug)]
pub struct Hull {
    /// The segments that constitute a hull.
    pub segment_set: Vec&lt;Segment&gt;,
}

impl Hull {
    /// Constructs a hull from it's segements.
    pub fn from_segment_set(segment_set: Vec&lt;Segment&gt;) -> Hull {
        Hull { segment_set }
    }
}
```

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

```rust
//! Provides the Orientation enum.
use shape::coord::Coord;

/// Represents the orientation of three points.
#[derive(Copy, Clone, Debug, Eq, PartialEq)]
pub enum Orientation {
    /// Three points are colinear.
    Colinear,

    /// Three points are in clockwise orientation.
    Clockwise,

    /// Three points are in counterclockwise orientation.
    Counterclockwise,
}
impl Orientation {
    /// Computes an orientation from coordinates.
    pub fn from_coords(p: Coord, q: Coord, r: Coord) -> Orientation {
        match (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y) {
            n if n < 0 => Orientation::Clockwise,
            n if n > 0 => Orientation::Counterclockwise,
            _ => Orientation::Colinear,
        }
    }

    /// Returns true if Orientation is Colinear.
    pub fn is_colinear(self) -> bool {
        self == Orientation::Colinear
    }

    /// Returns the opposite orientation, Clockwise becomes
    ↪ Counterclockwise, Counterclockwise
    /// becomes Clockwise, and Colinear stays the same.
    pub fn invert(self) -> Self {
        match self {
            Orientation::Colinear => Orientation::Colinear,
            Orientation::Clockwise => Orientation::Counterclockwise,
            Orientation::Counterclockwise => Orientation::Clockwise,
        }
    }
}
```

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

```rust
//! Provides the Polygon struct.

use shape::coord::Coord;
use shape::segment::Segment;

/// Represents a polygon.
#[derive(Serialize, Deserialize, Debug, Eq, PartialEq, Clone, Hash)]
pub struct Polygon {
    /// The set a points that make up the polygon, ordered counterclockwise.
    #[serde(rename = "point")]
    pub points: Vec<Coord>,
}

impl Polygon {
    /// Returns a vector of the segments that make up the polygon.
    pub fn segments(&self) -> Vec&lt;Segment&gt; {
        let cycleiter = self.points.iter().chain(self.points.iter().take(1));
        let cycleiter2 = self.points.iter().chain(self.points.iter().take(2)).skip(1);
        let edges = cycleiter
            .zip(cycleiter2)
            .map(|(&a, &b)| Segment::from_coords(a, b));
        edges.collect::<Vec<_>>()
    }
}
```

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

```rust
//! Provides the Segment struct.

use shape::coord::Coord;
use std::cmp::max;
use std::cmp::min;
use shape::orientation::Orientation;
use std::collections::HashSet;
use shape::polygon::Polygon;
use std::cmp::Ordering;

/// Represents a line segment AB.
#[derive(Serialize, Deserialize, Debug, Copy, Clone, Eq, PartialEq, Hash)]
pub struct Segment {
    /// The left start of the line segment.
```

```rust
    pub a: Coord,

    /// The right end of the line segment.
    pub b: Coord,
}
impl Segment {
    /// Constructs a line segment from it's coordinates AB.
    pub fn from_coords(a: Coord, b: Coord) -> Segment {
        let (a, b) = match (a.x.cmp(&b.x), a.y.cmp(&b.y)) {
            (Ordering::Less, _) | (Ordering::Equal, Ordering::Less) => (a, b),
            _ => (b, a),
        };
        Segment { a, b }
    }

    /// Checks if a point lies on self.
    pub fn contains_colinear_coord(&self, coord: Coord) -> bool {
        if Orientation::from_coords(self.a, self.b, coord).is_colinear() {
            (coord.x <= max(self.a.x, self.b.x) && coord.x >= min(self.a.x, self.b.x)
                && coord.y <= max(self.a.y, self.b.y)
                && coord.y >= min(self.a.y, self.b.y))
        } else {
            false
        }
    }

    /// Checks if self intersects another line segment.
    pub fn intersects(&self, other: &Segment) -> bool {
        if self.a == other.a || self.a == other.b || self.b == other.a || self.b == other.b {
            return false;
        }

        let o1 = Orientation::from_coords(self.a, self.b, other.a);
        let o2 = Orientation::from_coords(self.a, self.b, other.b);
        let o3 = Orientation::from_coords(other.a, other.b, self.a);
        let o4 = Orientation::from_coords(other.a, other.b, self.b);

        if o1 != o2 && o3 != o4 {
            return true;
        }

```

```
        if o1.is_colinear() && Segment::from_coords(self.a, self.b).contains_colinear_coord(other.a)
        {
            return true;
        }

        if o2.is_colinear() && Segment::from_coords(self.a, self.b).contains_colinear_coord(other.b)
        {
            return true;
        }

        if o3.is_colinear()
            && Segment::from_coords(other.a, other.b).contains_colinear_coord(self.a)
        {
            return true;
        }

        if o4.is_colinear()
            && Segment::from_coords(other.a, other.b).contains_colinear_coord(self.b)
        {
            return true;
        }

        false
    }

    /// Returns a value proportional to the distance between the line (extended
    /// of the segment) and the points.
    pub fn coord_distance(&self, other: Coord) -> i64 {
        ((other.y - self.a.y) * (self.b.x - self.a.x)
            - (self.b.y - self.a.y) * (other.x - self.a.x))
            .abs()
    }

    /// Finds the polygons that intersect with a segment.
    pub fn get_intersecting_polygons(&self, polygons: &[Polygon]) -> HashSet<Polygon> {
        let mut intersecting_polygons = HashSet::new();
        'p: for polygon in polygons.iter() {
            for polygon_segment in polygon.segments() {
                if polygon_segment.intersects(self) {
```

```
                    intersecting_polygons.insert(polygon.clone());
                    continue 'p;
                }
            }
        }
        intersecting_polygons
    }

    /// Finds the coordinates of polygons that intersect the segment.
    pub fn get_intersecting_polygon_coords(&self, polygons: &[Polygon]) -> HashSet<Coord> {
        let mut intersecting_polygons = HashSet::new();
        'p: for polygon in polygons.iter() {
            for polygon_segment in polygon.segments() {
                if polygon_segment.intersects(self) {
                    for coord in &polygon.points {
                        intersecting_polygons.insert(coord.clone());
                    }
                    continue 'p;
                }
            }
        }
        intersecting_polygons
    }
}
```

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
```