University of Amsterdam

Honours project 2007/2008

A Configuration Space Visualization for Robot Path Planning

Authors:
Hessel van der Molen
Wouter Josemans

Supervisor:
Dr. Leo Dorst

September 27, 2008

Acknowledgements

We want to thank our supervisor, Leo Dorst, for his infinite patience and his daughter for being born at exactly the right time to distract him. We also want to thank Ulle Endriss for coordinating the Honours project this year, and our fellow Honours students for their support.

Contents

1 Introduction **3**
1.1 Initial idea . . . 3
1.2 Proposal . . . 4
1.2.1 Robot . . . 5
1.2.2 Task Space & Obstacles . . . 5
1.2.3 Configuration Space . . . 6
1.2.4 User Interface . . . 7

2 Approach **10**

3 Calculating the no-fit polygon **12**
3.1 Polygons . . . 12
3.2 No-fit Polygon . . . 13
3.3 Representing the robot and obstacles . . . 14
3.4 Methods for calculating the no-fit polygon . . . 15
3.5 Splitting Concave Obstacles . . . 17
3.5.1 Detecting concavity . . . 17
3.5.2 Algorithm . . . 18

4 The Configuration Space **20**
4.1 Generating the Configuration Space . . . 20
4.2 Using the Configuration Space . . . 20

5 Path Planning **22**
5.1 Graphs and Scaling . . . 22
5.2 Finding Neighbours . . . 23
5.3 Traversal Algorithm . . . 25

6 Results & Conclusion **29**

7 User's Guide **31**
7.1 Parameter-window . . . 31
7.2 Task Space window . . . 32
7.3 Configuration space-window . . . 35
7.4 Advanced Parameter-window . . . 35

1 Introduction

Students of Artificial Intelligence at the University of Amsterdam have the option to do an honours program, which allows them to do some cool projects that should deepen their understanding of a certain area of AI. The idea is that students work in groups of two or three people for about half a year (6 ECTS), and at the end present their work to other students.

1.1 Initial idea

When we were given the options for Honours projects this fall, the description for this project was:

> Project A: Configuration space visualization for robot path planning Supervisor: Leo Dorst
>
> In your class on robot path planning, you have learned about configuration spaces as a means of thinking about the path planning problem. My future research will lead in the direction of fast transformation of the obstacles to and from configuration space, which is a hard problem.
>
> To develop a good intuition for this, and for educational purposes in the robotics classes, it would be good to have interactive (e.g. JAVA, python) applets, in which students/researcher can define arbitrary robots and obstacle shapes, and see how that affects the configuration space and the reachability of goals. Your task will be to make such a visualization. If there is time, we could augment this with a demonstration of path planning capabilities (A*, or more modern random path planning techniques).
>
> We can start simple, and probably do not want to move much beyond 3 dimensions. You will have to learn about object representation in computer graphics (volumetric versus polyhedral), the use of elementary graphics libraries, and we will have to think about the GUI (graphical user interface). Since I am not an expert in those fields, you will have to acquire a lot of this knowledge yourself – fortunately, it is rather standard. We will likely need linear algebra in many places to understand the literature, and I can help there, and with the choices of representation.

It'll be cool to have the tool!

In other words, the aim was to write a program. This program would need the following functionality:
- The user should be able to specify a robot
- The user should be able to specify a Task Space, including obstacles
- The program should be able to show the user the configuration space for the robot in the Task Space
- If time allows, the user should be able to ask the robot to find a path from 1 point in the Task Space to next, where the robot uses the configuration space to navigate

Some terms need clarification:
Robot: a shape with a point of reference, meaning that if we say "the robot is at location (2,3) in the Task Space", the point of reference lies at that exact point and the shape that is the robot is moved relative to that point.
Task Space: a 2D top-down view of the space the robot should navigate through. The Task Space has a length and a width and contains 0 or more obstacles.
Obstacles: shapes at some fixed position.
Configuration Space: a means of representing the interactions between the parameters of the robot and that of the Task Space. The robot can be represented as a point in the configuration space. If the robot has N degrees of freedom, then the configuration space would be an N-D space. For example, we can define a car (which has 3 degrees of freedom: x,y position and the angle) in the Task Space by drawing a rectangle of that robot with the reference point at x,y at some angle. In the configuration space, we can draw this robot as a single point, being (x,y,angle).

1.2 Proposal

During our first meeting, we established that it probably wouldn't be feasible to write this program for any generic robot. Robots come in many kinds, and they all behave differently. A robot arm, for example, remains stationary and moves her joints, whereas a robot car drives around in a certain fashion. There seemed to be several directions we could take, the most important being:

- We could keep the representation of the robot fairly simple, meaning we wouldn't meddle in more than 3 dimensions. This would give us time to do path planning with this robot. Downside is that this is "nothing new", similar applications have been written.
- We could also choose to represent more complex robots; say a car with a trailer. This would add a degree of freedom, meaning that we would have to think of a way to represent something in 4D. This would probably leave us with little time to focus on path planning.

We agreed that, being AI students, path planning would be more interesting to do. Also, because a 4D space would not be very intuitive, the whole point of the program would be lost. Therefore, we imposed some restrictions on several elements of the program.

1.2.1 Robot

The robot[1] will always have a convex shape. The reason for this is elaborated upon in the section on creating a no-fit polygon. Furthermore, a robot that is to be used in path planning will always have a rectangular shape (i.e. a car-like shape). Cars have a very distinctive way of moving that is limited in some ways by its shape. For the actual path planning, we are going to have to describe these movements. Since these movements are specific for car like objects, they would not be meaningful for shapes other than rectangles.

1.2.2 Task Space & Obstacles

We refer to the Task Space both as simply the "rectangle that limits the robot's world" and this same rectangle including the obstacles that are inside this Task Space. The distinction between these two will often be indicated by the context, unless the distinction is trivial or we explicitly state the proper meaning.

[1]In Artificial Intelligence, when we call something a "robot", we usually mean a more or less autonomous agent that can at least move around the world and has several behavioural patterns that are triggered by some input. In this report, however, when we refer to "the robot" we often mean nothing more than the *shape* (outline) of the robot. One can see that in creating a Configuration Space, we don't need any knowledge of the robot's behaviour, (assuming that the degrees of freedom are (x,y) coordinates for position and θ for rotation) we only need to know shape, size and center of rotation (the point of reference). It is therefore only in path planning context that we can justify talking about an actual robot.

The user will be able to define the width and length (sometimes referred to as height in this report) of the Task Space. The user will not be able to change the *position* of the Task Space; the upper left corner of the Task Space is always at the (0,0) coordinates (which is common in Java implementations). The width and length of the Task Space limit the width and length (again the x and y coordinates) of the Configuration Space.
The presence of obstacles is the heart of the problem we are trying to solve. We want to determine how the robot interacts with the obstacles, and do something useful with these findings, such as the planning of a path. Since the robot is always a convex shape, we can use a relatively simple algorithm to calculate whether the robot collides with obstacles for relatively complex obstacles. Again, the reason for this will be elaborated upon in a later section. The user can define an arbitrary number of obstacles, consisting of an arbitrary number of vertices. We require the user, however, to enter these vertices in a counterclockwise order. Also, the obstacle should not be self-intersecting.

1.2.3 Configuration Space

We wanted to create a Configuration Space that would, after a fashion, help us understand some things about the interaction between the robot and the Task Space. The number of dimensions in the Configuration Space is by definition equal to the number of degrees of freedom a robot has. We assume throughout our implementation that there are exactly three such degrees. Consequently, our Configuration Space has three dimensions. It will come as no surprise then that we chose to represent the Configuration Space in 3D. The Java3D API offered all the aspects we needed, and could be integrated easily, so we chose to use this to display the Configuration Space. This API allows the user to zoom, rotate and pan (among other things) the Configuration Space at will, meaning that they can navigate through it freely and study every aspect of it. Downside of this API is that it is built upon some platform dependent software, which means that the user has to download and install the Java3D runtime environment first. However, the same compiled code should still work on every platform.

1.2.4 User Interface

Combining all above made proposals, we should get an interface which could exists of multiple windows. During one of the first meetings we argued that there should be at least 3 windows: for the task-space, configuration-space, and for some user defined parameters. Since we wanted our implementation to work user friendly on multiple platforms, using three separate windows wouldn't be that great: on a Windows dependent system, multiple windows are represented with multiple links in the Windows taskbar. So using three separate windows results in three different links in the taskbar, which could make the taskbar quite messy. The solution we found lies in the DesktopPane[2] and Internalframe[3] classes provided by Java. Using these classes we are able to create one large window (DesktopPane), within an arbitrary number of small windows (InternalFrames). During further development of the interface we made a conclusion that it would be useful to let the user adjust some more advanced parameters concerning the interface and path planning. These parameters are grouped in a fourth window: the advanced parameter window.

In the implemented interface the Task Space window will be respresented by a lot of buttons to edit the obstacles, the robot and the data used for path planning. The Task Space itself will be represented by a 2D frame, in which a user can move or control the drawn objects by dragging and clicking with the mouse. These "mouse"-functions are implemented by using mouse- and event- handlers, which are provided by java.

The configuration space interface will be built in a single window. This window will consist of a 3D-axis, with the x and y directions horizontal, and the z-direction (rotation of the robot) in vertical position. Again, using some mouse event handlers, the user will be able to zoom, rotate and pan the axis, so he can move through the space.

Both windows for the parameters and advanced parameters will consist of some input fields and select boxes, to choose the right options or to define the wanted parameters for both the task-space, configuration-space or for path planning.

[2]http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JDesktopPane.html
[3]http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JInternalFrame.html

The combination of these interfaces can be found in Figure 1. As can been seen the whole system will consist of 5 GUIs: the 4 windows, described above, and a 5th, which is the main GUI: the Desktop. These GUIs will be built around a central "database" named GlobalSystemData. In this "database" all data will be stored, which is used among multiple methods of multiple GUIs. In this way it is possible to use a single declaration of each parameter or variable. This is useful because now every single GUI has access to exactly the same parameters and their values. Changing one value from a GUI will result in a changement of another GUI. Besides parameters, the GlobalSystemData will also contain instances of each GUI, so it is possible to control a GUI from another GUI. This will be used when some parameters are changed, and thus the TSpace and CSpace needs to be updated.
The GUI_Space will consist of some more methods. The first (MouseEvents3DGraph) will be used to rotate, zoom and translate throughout the C-Space, using the mouse. Using this method, it's possible to see the complete C-Space from a different point, which could help understanding what is happening, or to see some things more clearly. A different method which will be implemented, is a method which uses data (e.g. Robot size/rotation, Obstacle coordinates) from "GlobalSystemData" to calculate the C-Space. Third, there will be a method which will handle all "printing" of coordinates into the C-Space.
Our GUI_TSpace will also consist of multiple parts. First, the GUI will be split into a TSpace and a "ButtonBlock". In the TSpace objects can be drawn, deleted, edited or moved. To do this, some MouseHandlers will be implemented, which will use some variable of "ButtonBlock" to determine which object in which way has to be changed. The "ButtonBlock" is a column of different buttons. There will be two different types of buttons: Modification Buttons and buttons used for pathplanning. "Modification Buttons" are buttons which set variables, to tell the MouseHandlers of the TSpace which object in which way has to be edited, e.g. "Create Obstacle", "Rotate Obstacle", "Add coordinate to Obstacle" etc. The "PathPlan Buttons" are buttons concerned with Pathplanning. They will be used for calculating the optimal path[4] or for displaying the path step by step.

[4]At this point our pathplanning methods will be called.

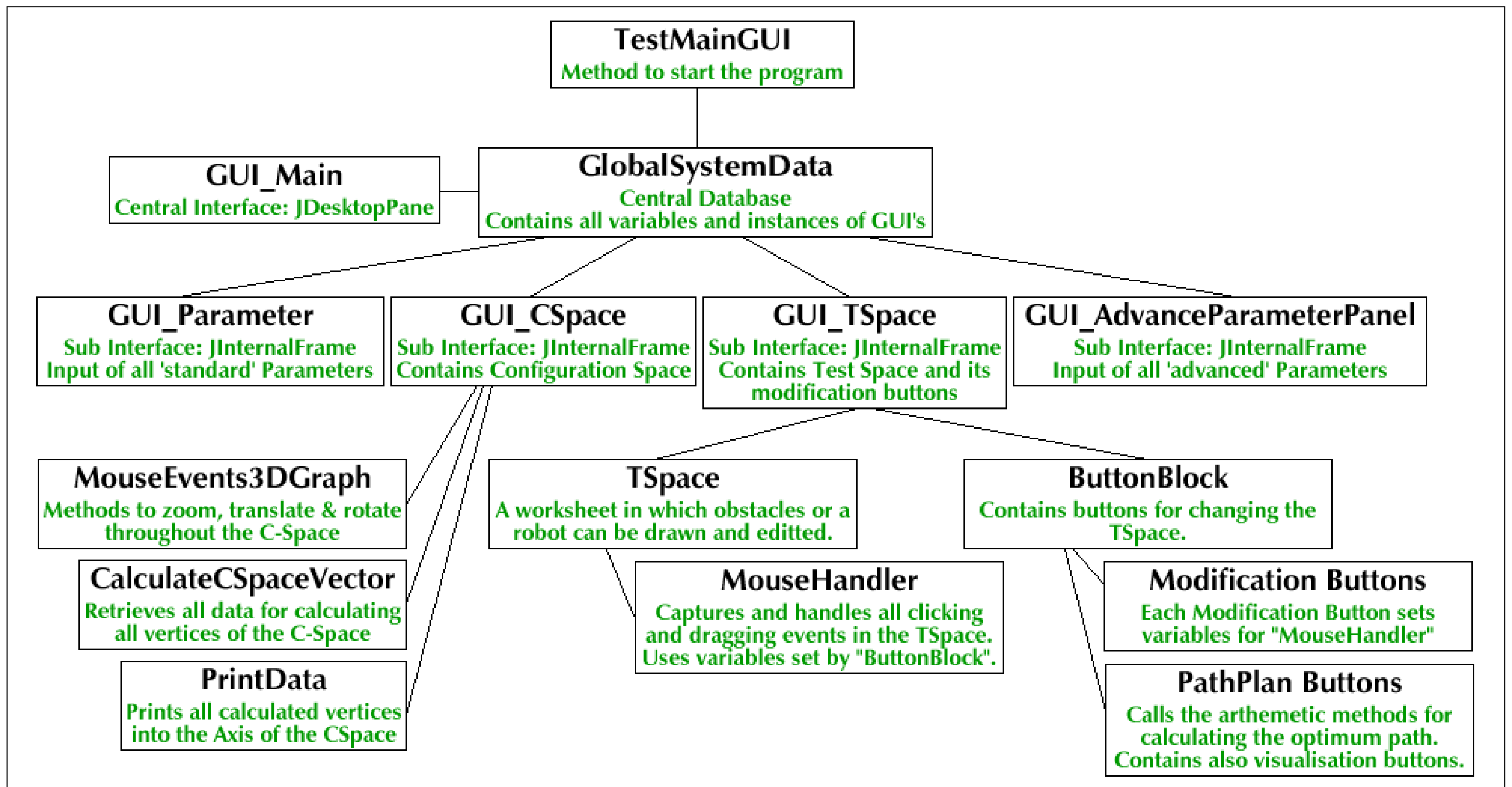

Figure 1: An overview of our Interface

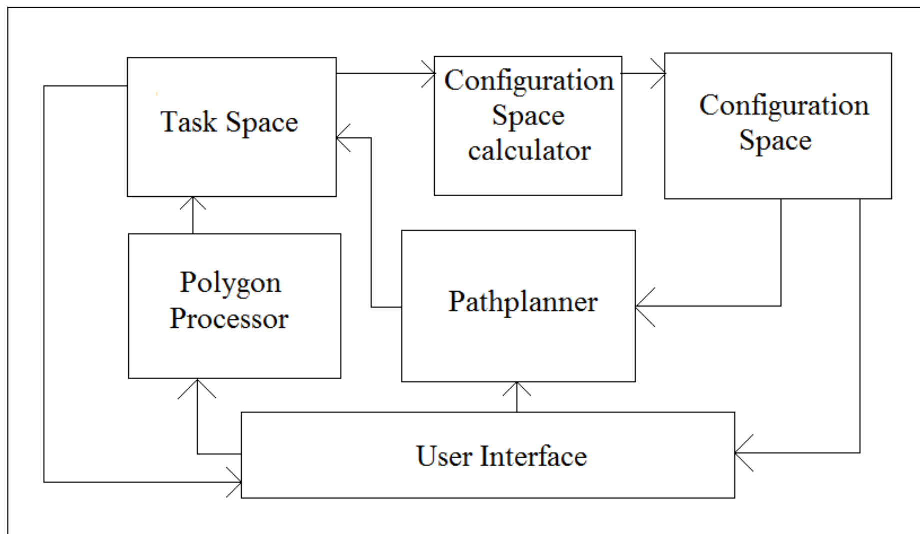

Figure 2: Interaction between components.

2 Approach

In the previous section we have described the necessary components, but this section serves to clarify how they interact. This way, the reader can get a clear picture of how each component fits into the system, and hopefully clarify some choices we have made in the process of building the system.

Figure 2 shows a schematic overview of how the components interact "under the hood". The components are named for their function for clarity and because we have lumped some components together for semantic purposes. When we describe the components in greater detail, we will use the "official" names. At the bottom we have the user interface. The user inputs data for the obstacles and the robots. This data is processed by what is called here the "polygon processor" which processes the user data, which is necessary to apply a certain algorithm. This is explained in section 3.5. This data is passed on to the Task Space, which in turn sends back data to the user interface for a graphical representation of the data (i.e., a draw shape). At the same time, the data of the robot and the obstacles is combined to form the Configuration Space. This is elaborated upon in section 3.4. The Configuration Space sends data to the user interface which is drawn on the user's screen. This is explained in section 4.

Once the user has specified a Task Space and the corresponding Configu-

ration Space is created, the user can start path planning. The user provides the start and goal positions to the path planner, which uses data from the Configuration Space to plan a path without colliding with an obstacle. For a detailed explanation, see section 5. Once the path has been planned, the pathplanner sends this data via the Task Space to the user's screen for a visual representation of the path.

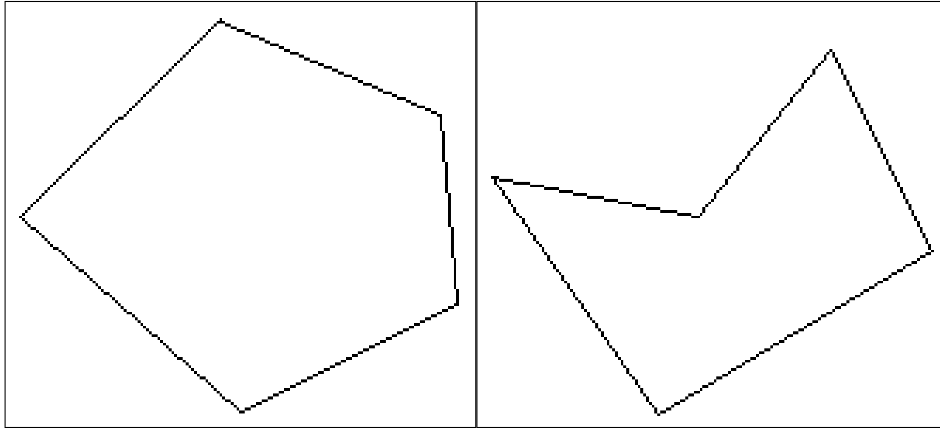

Figure 3: Left: convex polygon, right: concave polygon

3 Calculating the no-fit polygon

At the end of the project, we wanted a car-like robot to be able to navigate a path around obstacles from some start to some goal. Intuitively, there are certain issues that would need to be adressed to make this possible. One of the more obvious is *collision detection*; we cannot allow the polygons representing the robot and obstacles to intersect or even touch, because in a real-life situation this would result in damage to the robot or the obstacles (which one you care about most depends on the situation).

We weren't exactly the first to tackle this problem; there are several algorithms with their own advantages and disadvantages. To understand these advantages and disadvantages, we need to elaborate more on mathematical morphology and some background theory on polygons.

3.1 Polygons

Polygons are used often in computer graphics to draw 2D shapes. A polygon is a sequence of straight line segments that encloses a shape. A polygon can be described by an ordered list of vertices (the points on the "corners" of the polygon), where every two subsequent points denote a line between these points (including a line between the last and first vertices of the list). This representation makes it easy to perform certain operations on polygons, as will become apparent later in this report.

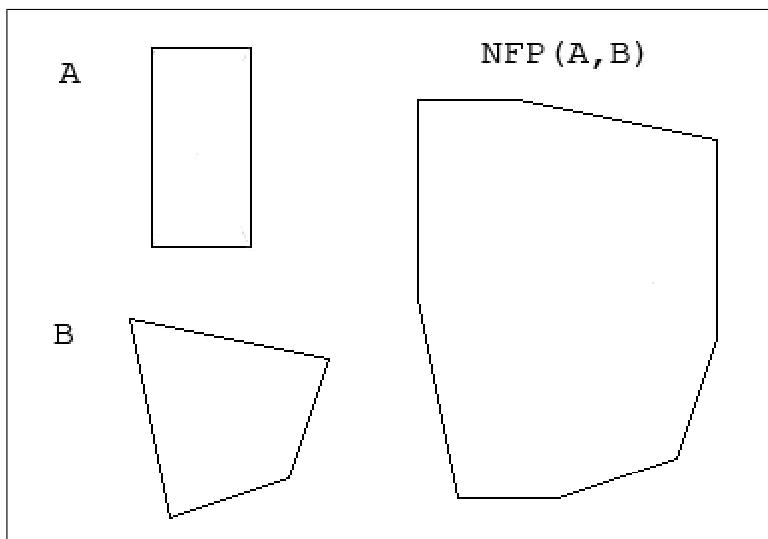

Figure 4: On the right the no-fit polygon of robot (A) and obstacle (B)

As seen in Figure 3, polygons can be either concave or convex (this is the case at least for *simple* polygons, i.e. polygons that do not intersect themselves and have no holes in them). The distinction between the two is important in several of the algorithms for calculating the no-fit polygon we could choose from.
A polygon is *convex* if every inner angle is less than or equal to π radians; equivalently, a polygon is convex if every line segment between two vertices remains inside the boundaries of that polygon. A polygon is *concave* if it is not convex.

3.2 No-fit Polygon

One of the more interesting interactions between obstacles and a robot is collision. We have focused on this particular type of interation throughout this project.
Since we have a static Task Space (stationary obstacles), it is rather intuitive that whether or not a robot collides with an obstacle depends entirely on the shapes of the robot and obstacles. The vertices of the robot are at different (x,y) coordinates for pretty much every angle, so the robot would collide with an obstacle at slightly different locations for every angle.
Since we are interested in exactly when such collisions will occur, we want to

draw a boundary around the obstacle for which a robot would collide with the obstacle if she were to cross it. This boundary is itself a polygon. In [4] this is called the No-fit Polygon, which is what we will call it throughout this report. What kind of polygon (concave or convex; simple or complex) depends on the kinds of polygons that represent the obstacles and the robot. How exactly this depends is explained later in this section.
An example of a no-fit polygon can be seen in Figure 4. In this figure, the no-fit polygon for robot A with obstacle B can be seen. The point of reference for robot A is the center of the robot.

3.3 Representing the robot and obstacles

As we discovered after a fashion, it is very important to consider the representation of every aspect of the program. Choices made early will always influence the way in which you can implement solutions later on, and most of the time in a way that is hard to predict. It should therefore come as no surprise that on some occasions, we were forced to throw away some early work and replace it with an implementation that would better suit our needs later on.
We started out representing the robot as a rectangle and the obstacles as a circle (described by a location and radius). We figured we would start with a very simple, easily implemented case which we would then evaluate. For a circle it is mathematically easy to see if a point lies within the obstacle, so we thought we could try to check at regular intervals of our (x,y,z) coordinates whether or not the robot intersects with the obstacle. It soon became apparent, however, that this was far from an ideal approach. It relied on the distance we chose between sampling points, so the success of the collision detection here depended on the resolution. Additionally, it was computationally extremely expensive, even when checking only in a bounding box around an obstacle. And as we represented an obstacle as a circle, our obstacles were also very limited.
All this gave us reason to choose to represent both the obstacles and the robot as polygons. That is, the shape of our robot is represented by a polygon, but she also has a point of reference which we use to indicate her position in the Task Space. The point of reference does not have any influence on the *shape* of the no-fit polygon, but rather on its position in the Task Space.
Both robot and obstacles should be convex polygons, because our algorithm for calculating the no-fit polygon is limited to convex polygons (why we chose

to do this is explained in the next section). This does not mean, however that the user can only give convex polygons as input; the user can draw any simple polygon, concave or convex. Concave polygons are automatically divided into smaller convex polygons. This is useful because the union of the no-fit polygons of the convex subparts of a concave polygon is equal to the no-fit polygon of that concave polygon.

3.4 Methods for calculating the no-fit polygon

As mentioned previously, there are several algorithms that will help us tackle the problem of calculating the no-fit polygon. The approach we chose was outlined by Ghosh [1] and elaborated on in [4]. Ghosh's approach can deal with situations where both polygons are concave. However, because we chose to represent our obstacles as convex polygons, this would be overkill for our problem. We had several reasons for choosing to represent obstacles as convex polygons.
First of all, the application we have built is a simple one; we do not expect that it will be used for very precise calculations; it will mainly be used to get a reasonably accurate idea of how shapes interact to form a Configuration Space. The user will mainly enter rather simple shapes (such as rectangles), which are usually convex. Furthermore, the user can approximate concave shapes by building them from convex polygons. As mentioned before, even if the user does draw a concave shape, it'll be split up automatically. The algorithm for this is described in a later section.

Second, the algorithm for calculating the No-fit Polygon for just convex obstacles is not as complex as the general algorithm that works for concave polygons, so it is easier for us to implement. Downside is that we have an extra algorithm to run to split a concave obstacle into smaller, convex ones. However, because we expect that the user will enter mainly convex shapes, we chose this approach.
Lastly, the algorithms that deal with concave shapes bring problems of their own. For example, the No-Fit Polygon could self-intersect which would complicate subsequent calculations and rendering. After weighing these problems, we decided to go for the simpler, convex polygon approach.

[4] offers a simple algorithm that works for convex polygons in a computationally cheap way. Our implementation is therefore based on this approach.

A pseudocode representation of our algorithm can be seen in Algorithm 1. All operations in this algorithm are at worst $\mathcal{O}(n)$, except for the edge sorter, which is $\mathcal{O}(n^2)$.

```
   Data: RobotPolygon and ObstaclePolygon
   Result: No-fit Polygon of RobotPolygon and ObstaclePolygon
 1 foreach edge in either polygon do
 2 |   Calculate length and angle;
 3 |   Store length and angle in edge array;
 4 end
 5 Sort the edge array by angle;
 6 Select first edge and remove from edge array;
 7 Add this edge to result;
 8 while Edge array not empty do
 9 |   Take first element e of edge array;
10 |   Append e to the end of the previous edge in Result;
11 |   Remove e from edge array;
12 end
13 Shift the NFP to the correct position by translating all e in Result;
```

Algorithm 1: Calculating the No-fit Polygon

We give as input the vertices of the polygons of the obstacle and the robot. The vertices of the obstacles are represented in a counterclockwise order; which is the usually the default notation. The reason that the robot is represented in a clockwise fashion, is because we need to “dilate” the obstacle with a representation of the point-mirrored version of the robot. This boils down to reversing the order of the edges, because edges are represented by two vertices (vectors). Mirroring the robot means mirroring these vectors (multiplying them by -1), and this results in the same robot, but with edges in clockwise direction.

In Algorithm 1 line 5, we sort the edges by angles. Note that these are the angles that the vectors would make with the vector $[1\ 0]^T$ in counterclockwise direction in a standard axis system (x axis horizontal, y axis vertical) in the interval $[0, 2\pi)$.

3.5 Splitting Concave Obstacles

Because we have chosen for a relatively simple algorithm for calculating the No-Fit Polygon, we had to implement a solution to split concave obstacles into convex ones. This is in itself an interesting problem, since there are some issues that complicate things. Some algorithms for this purpose are discussed in [3]. We will elaborate on our approach in this section because the implementation of the algorithm helps illustrate some interesting problems we ran into.
When considering the problem of splitting concave polygons into convex ones, there is again a tradeoff to be made. One can keep the algorithm for splitting concave polygons relatively simple (for example, by dividing it into triangles), but this will result in a sub-optimal number of convex polygons. Since the splitting of concave polygons typically has a higher purpose,(some calculations can only be performed on convex polygons, or the calculations involved can be either very complicated or very expensive) a close-to-optimal number of convex subpolygons would be a desired solution. An overview of these methods is presented in [5].

3.5.1 Detecting concavity

The following statements both describe the concavity property of a polygon. A polygon is concave if and only if (two statements are equivalent):

- The polygon has at least one vertex with an inner angle greater than 180 degrees
- If some straight line intersecting the polygon encounters more than 2 borders

From the first statement, we know that a triangular polygon can never be concave. As a matter of fact, the triangle is the simplest convex polygon. A check for concavity is not too hard to implement; consider two vertices $v1$ and $v2$ with $v1 = \begin{pmatrix} x1 \\ y1 \end{pmatrix}$ and $v2 = \begin{pmatrix} x2 \\ y2 \end{pmatrix}$. Now let d be $\begin{pmatrix} x1 \\ y1 \end{pmatrix} \cdot \begin{pmatrix} y2 \\ -x2 \end{pmatrix}$. Now we need some additional information; if we have listed our vertices in *counterclockwise* direction then $x1$ is a "concave vertex" if the $sign(d)$ is minus one. Analogously, if we have listed our vertices in *clockwise* direction, then the polygon is concave if there is a notch such that $sign(d)$ is greater than zero. In both cases, we call $v1$ a *notch* (notches are the "vertices that

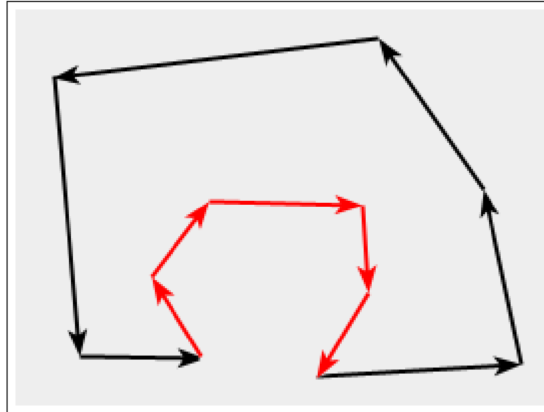

Figure 5: A concave polygon with a pocket. Red arrows are edges between vertices whose direction is reversed.

make the polygon concave"). The situation is complicated by *pockets*, or a sequence of notches. When looking at this sequence individually, we find that the vertices are suddenly ordered in the reverse direction. For an example of this phenomenon, see Figure 5. This is something we need to keep in mind if we want to classify vertices as notches, and this property also complicates the process of checking subpolygons for concavity.

3.5.2 Algorithm

In the end, we settled for a sub-optimal solution. We put a frustrating amount of work into getting an optimal algorithm to work (we came up with it ourselves), but debugging this algorithm took a lot of time. With the deadline approaching swiftly, we decided that a simpler algorithm that did what it was supposed to do in a reasonably efficient way was preferable to an algorithm that had so many special cases that it was nearly impossible to test them all.

We split the obstacles into triangles. We have chosen to do this because splitting into triangles is pretty straightforward. A common approach is the subtracting ears method where "ears" are repeatedly subtracted from the polygon. And ear is a triangle within a polygon. Two sides of the ear are edges that are on the outer bounds of the main polygon, and one side is not. After having split the obstacle into ears, we are going to try to merge these

triangles while preserving convexity. A pseudocode algorithm can be seen in Algorithm 2. Note that the findEar method only finds correct ears; that is, it will exclude triangles that contain other points as ears. It will also exclude triangles that do not lie wholly within the polygon.

```
   Data: Array of (x,y) coordinates inputPolygon
   Result: Array of convex polygons resultPolygons
 1 while sizeOf(inputPolygon) less than 3 do
 2 |   triangle = findEar(inputPolygon);
 3 |   Remove triangle(tip of) from inputPolygon;
 4 |   add triangle to trianglesArray;
 5 end
 6 while trianglesArray not empty do
 7 |   polygon = first element of trianglesArray;
 8 |   while notConcave(polygon) do
 9 |   |   merge polygon with adjacent triangle from triangle array;
10 |   |   if concave(polygon) then
11 |   |   |   Undo last merge;
12 |   |   |   remove triangles of polygon from trianglesArray;
13 |   |   |   add polygon to resultPolygons;
14 |   |   end
15 |   end
16 end
17 return resultPolygons;
```

Algorithm 2: Algorithm for splitting concave obstacles into convex subparts

4 The Configuration Space

The Configuration Space plays a central role in our implementation. It is a useful tool that allows us to model all possible configurations of our robots, and look at problems such as path planning in a different way.

4.1 Generating the Configuration Space

We have seen in the previous section that we can calculate a No-Fit Polygon for an arbitrary convex robot (at a specific orientation) and obstacle. Since we are dealing with a robot who will move in a car-like manner, she will be able to change her orientation whenever she's driving. We assume that the robot can use her manoeuvers to reach any point in the Task Space at any angle at some resolution.

We can essentially calculate the No-Fit Polygon for any double-valued angle, but for the sake of computational efficiency we have to choose some resolution for the intervals at which we draw our No-Fit Polygon. If we treat the angle of the robot (we rotate her around her point of reference) as a third coordinate, we can draw all these No-Fit Polygons in 3D. We usually see an interesting shape emerge[5]. See Figure 6 for an illustration of the Configuration Space. Values on the z axis range from 0 to 360 degrees, with intervals 5 degrees wide. Layers have been coloured to make it easier to distinguish between them.

4.2 Using the Configuration Space

In our Configuration Space, we can represent our robot as a point defined by three coordinates. Consequently, simple point-in-polygon algorithms can calculate whether the robot collides with an obstacle given a position. One can rotate, pan and zoom in the Java3D implementation we provide, making it easy to examine some parts of the space up close.

---

[5] Humans are often biased to see such a figure as a solid shape, this is called closure. It is, however, not completely safe to interpolate between layers in such a way; there are some situations in which it is not unlikely that the shape between layers is very interesting. Imagine a situation where there are two rectangles with parallel sides at some distance from each other that just happens to be the width of our robot. In this case, she can pass between these rectangles only when she's at exactly the same angle as the sides she has to pass between. This would look like a hole in our graph if it happened to fall exactly on the interval of resolution, but it is not likely that we will always be that lucky.

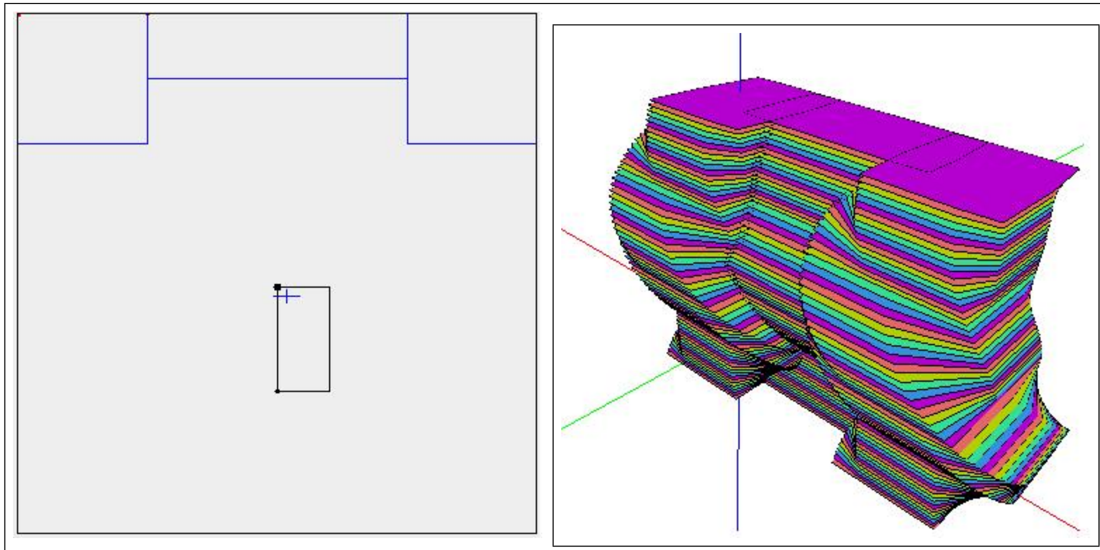

Figure 6: Left: a Task Space, right: corresponding Configuration Space

The calculations performed are limited to the size of the Task Space, but in the Configuration Space viewer we are still able to look past those bounds (the bounds themselves are not shown in the Configuration Space). Our robot's movements in path planning are also restricted to the bounds defined by the Task Space. When we have planned a path (as elaborated upon in a following section), the path is drawn in the Configuration Space. This path draws a straight line between nodes, however, while the actual movement of the robot would be more fluid and curved when not driving straight forward.

5 Path Planning

With the tools implemented as described in the above sections, we can implement a path planning algorithm with relative ease. There are, however, a few issues that we need to tackle first. When we are planning a path, we assume that the robot is actually going to move along this path to the goal. That means that the movements of the robot restrict the possible paths we can take. So to plan a path for a specific robot, we need to represent the movement of the robot. Another problem is in finding some optimal heuristic to guide our A^* implementation.

5.1 Graphs and Scaling

Theoretically, we are searching (a discrete version of) the Configuration Space for a path. The useful thing about the Configuration Space is that we have reduced our robot to a point in 3D space, meaning that we have to find a path using the provided move function such that it does not at any point intersect one of the no-fit polygons.

In practice, however, we search a Graph. Graph search is a well documented process that has a history of being used in robot path planning. A graph consists of *nodes* with one or more *edges* between them. If there is an edge from node x to node y, we can reach node y from node x (in an *undirected* Graph, we can also reach node x from node y). A sequence of edges can be called a path.

We use Graph Search Algorithms for finding a path from a node to an arbitrary node anywhere in an undirected graph. When done by uninformed search algorithms, this can take a long time, depending on the number of nodes. Even with an informed (greedy) search algorithm, finding a path through a large number of nodes can take quite some time (if there is no path to be found, the algorithm will need to explore all possible paths before it can claim that there is no such path, which will usually take more memory than available in standard Java Virtual Machines). Since the nodes of a graph are basically points sampled from our Configuration Space, we need decide on an acceptable interval at which we will take samples. After all, the Configuration Space has three dimensions, which means that computations cam become quite expensive when we sample at short intervals. However, if we make the interval between sampling points too long, the movements of the car will look unnatural. Whereas this can be solved by interpolating

between points, the fact remains that we "jump" a long way between points. Between those two points, a whole lot of nasty things could be happening to our robot and we wouldn't even notice until we do the actual driving and collide with, for example, an obstacle.
What the problem boils down to, essentially, is scaling. Since the user can define the size of the robot, the obstacles, and the Task Space, we cannot simply discretize at some predefined interval. This would give us a resolution that is either too high or too low. This is why we need to scale our discretization to conform to either the size of the robot, the size of the average obstacle or the size of the Task Space. A motivation for choosing one of these can be found in the environment we want to plan a path through. Consider we want to navigate our robot through a desert. Every self respecting desert has a bunch of large rocks of several times the size of the robot here and there. In such a scenario, it would be a better idea to scale to the size of the Task Space or the size of the obstacles (in this case, the respectably huge rocks) than to scale it to the size of robot. At that scale, after all, there is not much interesting happening, so it would be much more expensive than it need be. In the application we have in mind (parking a car between other cars, for example), the scale of the robot will do just fine. If our Task Space consists of a scene where there are two cars with an empty space in between, then our robot and the obstacles are at a relatively similar scale. It is then to be expected that (provided that the obstacles and robot are not too small [6] relatively) we have found a good tradeoff between resolution and compational cost.

5.2 Finding Neighbours

As mentioned before, we need to find a way to model the movements of the robot. For this, we have restricted the robot to be rectangular (i.e., the robot should be longer than it is wide), since a rectangle roughly approximates the shape of a car. We have made this restriction because we know how cars move. Allowing path planning for an arbitrary convex robot would be meaningless, because we have no idea how for example a triangle moves. Note, however, that using the tools implemented in this program we can easily extend the functionality to allow path planning for any convex robot with a

[6] a 3x10 robot in a 400x400 space would be a bit small, a 30x100 robot would be reasonable.

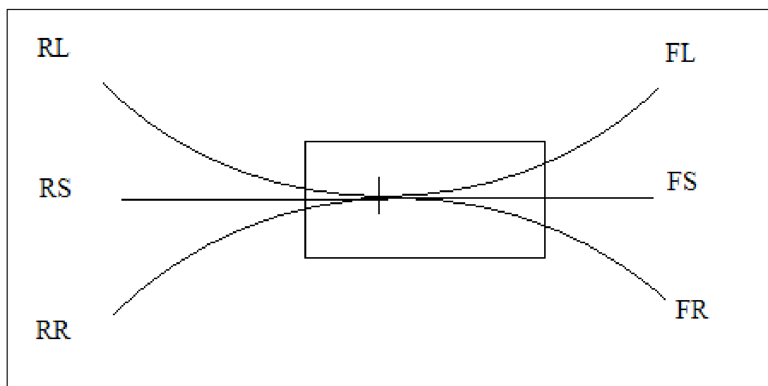

Figure 7: The directions of the neighbours of a car-like robot

well defined (meaningful) move function.

As we have decided that we need to scale our discretization interval to our car, we need to know which parameters are important. First, we need to know some things about car movement.
We will employ a method called "bang-bang control" for our robot. Bang-bang control means that there are only a few possible angles for the wheels to be at. Changes between these positions are made instantly, with no intermediate states. Bang-bang control is sufficient to achieve optimal manoeuvrability [2]. We have chosen to "bang" between wheel angles of 30, 0 and -30 degrees, because this seems to model the movement of cars quite well. Along with a reverse gear, we now have six direction in which we can move: straight forward, forward to the left along the turning circle, forward to the right along the turning circle, straight backward, backward to the left along the turning circle and backward to the right along the turning circle. For a graphical impression of the direction of each neighbour, see Figure 7.

In addition, we have to decide how far we want to drive in each direction. This is obviously correlated with the discretization factor; we want to drive at least far enough for the robot to "reach" the nearest sampling point. We have chosen to define three neighbours in each direction, one at 1 unit of driving distance, the second at 2 units of driving distance and the third at 3 units. We chose to do this to favour longer movements in one direction, but to maintain a decent degree of resolution, as was done in [6].

Note that we have already discretized our Configuration Space in the z

direction; we did this in steps of 5 degrees so we are basically tied to a path planning discretization of a multiple of those 5 degrees. This does give us one less discretization to worry about; we can treat the discretization in z direction as a constant, which helps in determining the discretization for x and y directions. We can calculate how much the change would be in x and y by rotating by the size of the discretization interval in z direction, and use this to determine the minimum distance to be traveled along the turning radius.
The turning radius depends on the angle of the front wheels (in this case, 30 or -30 degrees) as well as the wheelbase (the distance between front and rear axes). The longer the wheelbase, the greater the turning radius. More precisely, we can describe it by the following formula:

$$r = \frac{wheelbase}{tan(30^o)}$$

When we have a turning radius, we need to find the point around which to turn. If we know this point, we can calculate the new position using for example a rotation matrix. We have chosen the turning radius in such way that the circle it describes passes through the point of reference, which is in the center of the rear axis, as was done in [6]. This means that a line, perpendicular to the robot, that intersects this point can be used to calculate the center of the circle described by the turning radius. The calculation of the new x and y coordinates is then only a matter of translating the robot, using the rotation matrix, and translating back. Alternatively, homogeneous coordinates can be used to do all this using one matrix.

5.3 Traversal Algorithm

As mentioned before, the A^* search algorithm is one of the ways in which we can choose to implement our Graph Traversal. The advantage of A^* over other search algorithms is that it will find an optimal path based on the weight function we attribute to the transition between nodes, provided we have an admissible heuristic. An admissible heuristic is one that always gives an estimated cost that is less than or equal to the actual cost. If, additionally, we have a perfect heuristic (one that always estimates the realy cost) we will find the optimal path fairly quickly. The cost of a (partial) path is typically defined as:

$$f(x) = g(x) + h(x)$$

where $f(x)$ is the total cost, $g(x)$ is the actual cost from traversing from the start node to the current node and $h(x)$ is the estimated cost from the current node to the goal node. The simplest admissible heuristic is $h(x) = 0$. This heuristic is a *very* optimistic estimate of the actual cost, because it says the actual cost will always be 0. This means that $f(x) = g(x)$, meaning that we base our cost only on the past nodes, not on any that might be following, so it is basically a greedy version of breadth-first search. With this heuristic, we will find an optimal path. Unfortunately, it will take quite long to find this path, because the algorithm is expanding nodes indiscriminately. So either we have to find a good heuristic, or we have to make sure our search space is reasonably bounded (or we have to be really patient and give Java a lot of extra memory). In [6], $h(x) = 0$ is used. We used this is well, to ensure the correctness of our program. We experimented with the distance in x and y coordinates as a heuristic, but this heuristic only worked well if the optimal path did not contain too many curves, since a curved path would initially be estimated as more expensive. For these kinds of paths, little time was won when using the heuristic.
See Algorithm 3 for pseudocode of our A^* implementation. The open list is implemented using a Priority Queue, meaning that the cheapest subpath is always at the head of the queue.

```
Data: Start, goal
Result: Path from start to goal
1  priorityqueue open ← Start;
2  closed ← {};
3  path ← {};
4  while sizeOf(open) greater than 0 do
5  |   node ← poll(open);
6  |   if pos(n) equals goal then
7  |   |   add n to path;
8  |   |   return path;
9  |   end
10 |   if n already in closed then
11 |   |   continue with next loop;
12 |   end
13 |   add n to closed;
14 |   neighbours ← neighbours of n;
15 |   foreach neighbour in neighbours do
16 |   |   offer n to priorityqueue open;
17 |   end
18 end
19 return null;
```

Algorithm 3: Finding a path in a graph

In our implementation, every node is a linked list containing the path from the start to that node. The node contains additional information such as transition cost and estimated cost as well. The actual cost is the distance the car drove to get from the previous node to the current node.
A planned path is drawn in both the Task Space and the Configuration Space. This can be seen in Figure 8. This situation can be compared to the parking of a car between two other cars. As seen, it is cheapest to first drive past the empty space, turn away to the right slightly and then drive backwards into the slot. There are a few strange moves the car seems to make, this is due to the resolution in which we discretized and to the fact that the robot has some pseudo-realistic moves that aid us in our discretization. Looking back at Figure 7, the first neighbour in the FL direction will end up at the same x and y coordinates as the FS neighbour, but with a different z coordinate (angle). This is not a very realistic move, since it would "sliding" toward that neighbour. Nonetheless, this is required to make the second an third

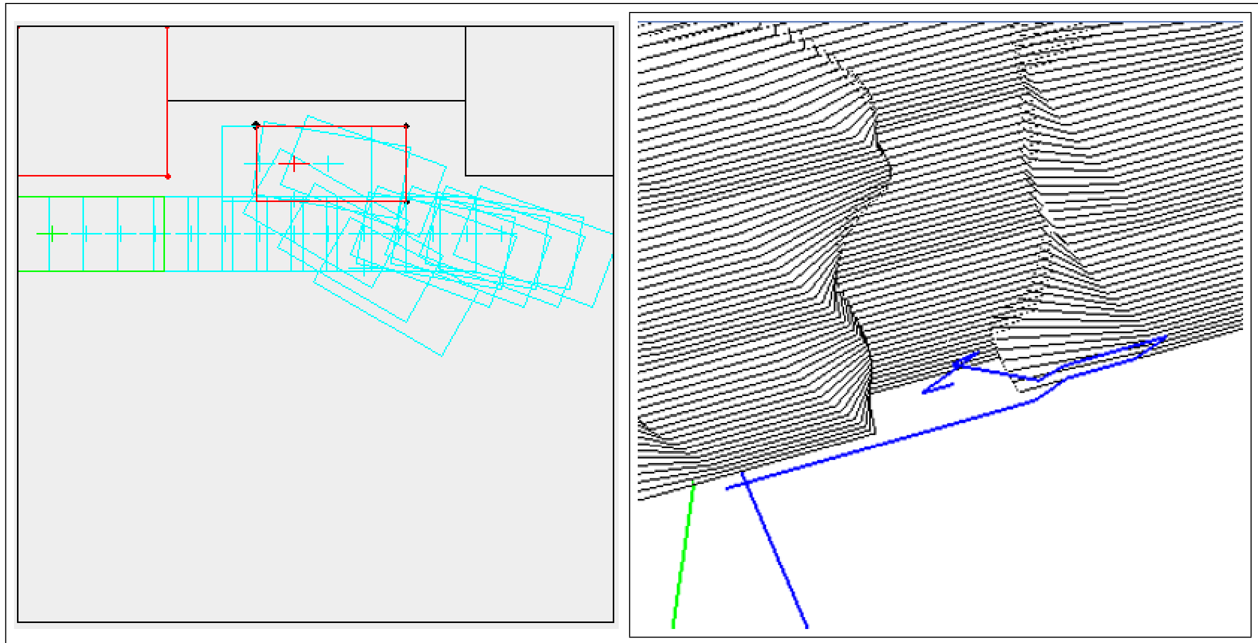

Figure 8: Left: planned path from green to red in the Task Space, right: corresponding path in the Configuration Space (blue and green bars are axes)

neighbours end up in a more realistic spot.

6 Results & Conclusion

We have worked on this project for quite a while, but it was not until the very end that we realized that we had bit off a bit more than we could chew; even though we put a lot of effort into it, we did not complete the program. The preprocessing of the polygons into a Configuration Space may work well, but the path planning by means of this Configuration Space does not work flawlessly. Due to several persistent bugs and our approach for discretizing, the path does not look very much like a path traversed by a car in the real world. Fixing this issue would take an effort that would exceed the scope of this honours project. After consulting our supervisors, we decided that we had in fact reached the pedagogical goal of this project; i.e. learning to plan, set up and coordinate the development of a medium sized software application. Even though the final product does not work as expected, we have learned a lot about the mathematical tools required and about matters such as writing reports and presenting our results.

In retrospect, we could have done the software engineering a lot better if we had had the course Datastuctures before we started out, because in this course we learned many advanced programming skills and conceptual tools that would have aided us greatly in developing our application. For instance, by using hashmaps we could have improved the computational efficiency of path planning.

Scheduling the project was not optimal either; as this was a project alongside our normal courses, we tended to prioritize the latter because there was little pressure to finish this project. Because of this, finishing the project took a bit longer than we had hoped[7].

The division of the work could have been better as well. It soon became apparent that there were two major tasks; one was the computation of the Configuration Space and path planning with its subproblems, and the other was parsing the results in a meaningful way to the user, and providing a user interface. We both took one task and specialized in that area, and after a while we concluded that it would not be efficient to switch tasks. It would take a lot of time to read up on the other's progress, so we decided to both focus on our own task. At that point, this may have been a good choice, but it would have been better if we had both specialized at both tasks. This

[7]Though apparently, as a rule every software project takes longer than expected, even when one takes this rule into account.

would have required a different approach from the very beginning, and the use of tools such as source control.

It would also be al lot better if we had made some (more or better) agreements about some datatypes before programming. Because, now, throughout the program there are a few datatype conversions which wouldn't be needed if we were consistent while programming. For example: in a few parts there are calculations done in degrees, while others are done in radians. These conversions are also a result of the division of the work: at some points neither of us knew how and with which datatypes the other was programming or calculatng.

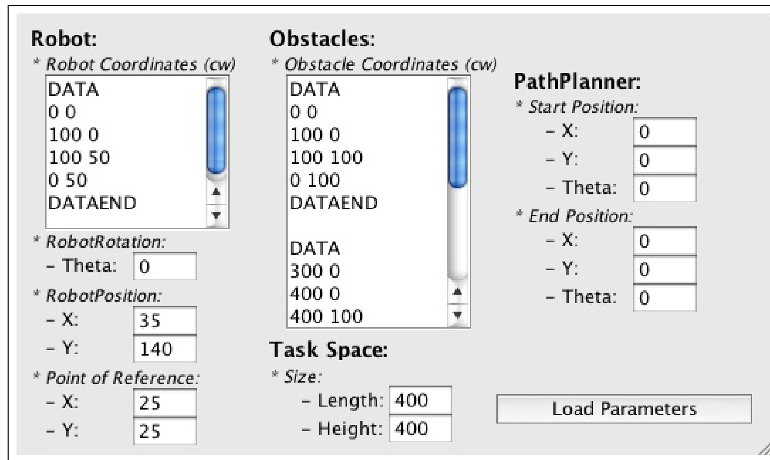

Figure 9: Interface Parameter window

7 User's Guide

Our program is built upon four windows: Parameter, Task Space, Configuration Space and Advanced Parameter. These windows are brought together in one desktop: the main interface. This interface contains one menu bar with four buttons, each one for opening one of the four windows. Furthermore, the main interface consists of the standard close/minimize/maximize functionality. Now, for each window we will explain its functions, and how to use them.

7.1 Parameter-window

The first window is the Parameter Window, which is shown in Figure 9. This window contains all tools to define and position objects at exact places.

The first column defines data that belongs to the robot. Below "Robot Coordinates (ccw)" the user may specify the robot. The given coordinates should start with point [0;0], with each coordinate given in counter-clock-wise (ccw) order. With path planning, the rightmost side of the robot will be the front (so in this example, the front line is from [100;0] till [100;50]). The coordinates should start with the text "DATA", and then given on each row, first the x-coordinate, followed by whitespace, followed by the y-coordinate. When all coordinates are given the user has to type "DATAEND" to indicate

to the program that all robot coordinates are given. Next in the “Robot”-column, we find the RobotRotation. This value defines the rotation of the robot counterclockwise in radians. Third in row, are the values to define the [x;y] position of the robot in pixels. The “Point of Reference” will be placed at the specfied robot position. Last in row, are the [x;y] values of the “Point of Reference”. These values define the place of “Point of Reference” in a robot relative to the first coordinate of the robot (the [0;0] coordinate).

The middle column of the window, defines data about the obstacles or the Task Space. First the obstacles are defined. Just like a robot an obstacle is defined between the “DATA” and “DATAEND” flags. Between the two flags coordinates (in clock-wise order) ar defined with x and y values, similair to the way of defining a robot. To place multiple obstacles in the Task Space, the user have te insert first an empty line. After it, a extra obstacle can be defined using the “DATA”-flags.
For the task-space itself, it is possible to chainge the length or height of it: just enter the number of pixels in the right fields.

The third column defines the start and end values of the robot for path planning. Both start or and values are defined as a [x;y] coordinate (the place at which the point of refernce of the robot will be placed), and a clock-wise rotation of the robot at the specified point in degrees.

When all parameters are set, press “Load Parameters” and both the Task Space and Configuration space will be updated with the new data.

7.2 Task Space window

The Task Space-window (see figure 10) is a graphical version of the parameter window. It supports mouse interactions with the drawn objects. On top of the window there is a “statusbar” which displays the performed action, errors or other information. Below, the window is split into two parts; left all action buttons, and right the Task Space.

All actions buttons are grouped by object actions. The first group is the “Obstacle Data” group. With these buttons it is possible to add a complete new obstacle, to edit an existing one, to move one around, rotate, or duplicate. When you want to add a new obstacle, hit the ”Add Obstacle”-button.

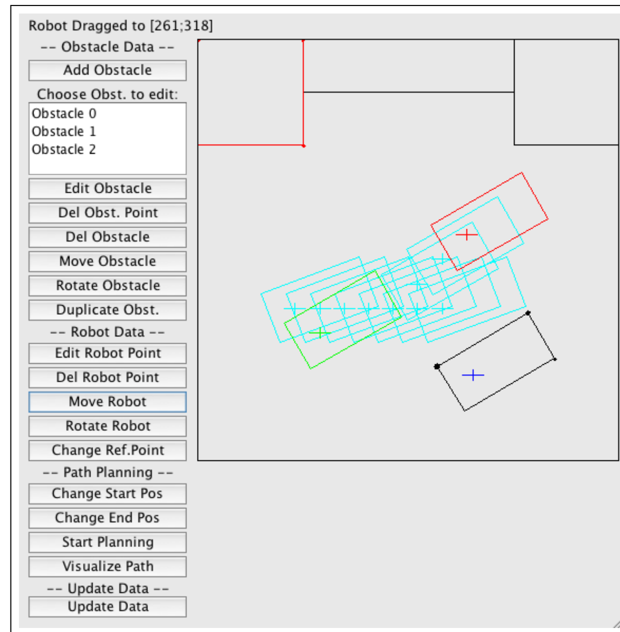

Figure 10: Interface Task-Space window

Clicking somewhere in the Task Space will result in a dot. This dot is the first coordinate of the obstacle. Each new coordinate added will be placed first in the counterclockwise direction of the first dot. For all other actions, the user has to select an obstacle from the list. The selected obstacle will appear green in the Task Space, when editing, it will turn red. By hitting the the "Edit Obstacle"-button it is possible to add new coordinate to the selected obstacle[8], or existing coordinates can be dragged to a new position. When you want to remove some points from an existing obstacle, the "Del Obst. Point"-buttons has to be pressed. Now, by clicking the right coordinate, it will be removed from the obstacle. Of course it could also happen that a complete obstacle has to be removed. By selecting the obstacle from the list and using the "Del Obstacle" button, the obstacle will be removed, after a prompt. A fourth function for editing obstacles is to move them around the Task Space. By using the "Move Obstacle"-button an obstacle

[8]When a obstacle consists of more than 3 coordinates, the first coordinate will turn in a big red blob, the second and third in some smaller blobs. Be sure the size of the blobs are decreasing in counterclockwise order! Otherwise the program could do some strange stuff.

can be dragged. Furthermore it is possible to rotate an obstacle. Rotating will be done around the obstacle's center of gravity. It can be done by using the "Rotate Obstacle"-button. A Last function for edditing obstacle, is to dublicate the selected one (the "Duplicate Obst."-button). The duplicated obstacle will be placed at exact the same location as the original, so it will happen often that after duplication, you'll use the move function.

The second group is the "Robot-Data"-group. This group contains 5 buttons, each for editing the robot. The first one ("Edit Robot Point") is for editing a robot point. It works the same as the edit button for obstacles. The second button (called "Del Robot Point") removes a selected point of the robot. Third in row is the "Move Robot"-button. Just like the name explaines, hitting this button allows you to drag the robot around the Task Space. Just like obstacles, it is also possible to rotate the robot. After hitting the "Rotate Robot"-button the robot can be rotated around its point of reference. The last function build for editing the robot, is to change the point of reference position (using the "Change Ref. Point"-button).

The second last group contains buttons for path planning. The first two buttons are for selecting the right robot configuration (rotation and place) for the start or end position. Selecting a position can be done by dragging (and if needed rotating) the robot to the desired position, hitting one of the two buttons results in selecting the current robot configurations for the start or end position. A start postion will result in a green copy of the robot, the end position will be red. The next button in row is the "Start Planning" button, which - when hit - tries to find a path from the start to end position. If the program can not find a path, it displays a message in the status bar on top of the Task Space window. Otherwise, it wil display the complete path in de Task Space (Cyan colered). Using "Visualize Path" the user is able to see the robot movement per step instead of all steps at once.

The fourth and last group - "Update Data"-group - contains 1 button: for updatating all data from the Task Space to the parameter windows and the Configuration space.

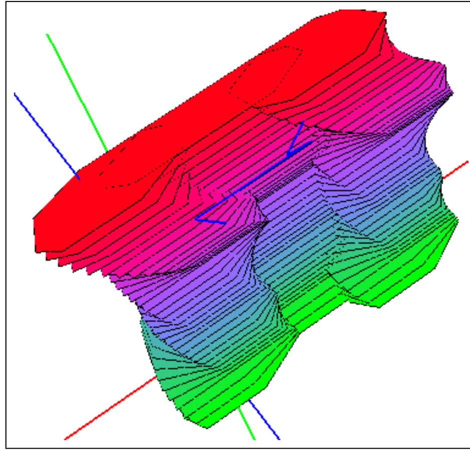

Figure 11: Interface Configuration-Space window

7.3 Configuration space-window

The configuration space is 3D representation of where a robot may or may not drive given some objects and the position, a robot, and the angle of the robot. A screenshot of our implemented CSpace can be seen in figure 11. There are 3 axes, a blue, green and red one. The red one represents the x-axis, green is the y-axis, and blue is the z-axis which represents for the angle of the robot.

With the CSpace it is possible to rotate the graph, to zoom or to translate. All actions are done by using mouse interaction. Using the left-button to drag the mouse cursor around the CSpace results in a rotation. The axis will be rotated around point [0,0,0]. Zooming is done by using the middle-button and translation is done by dragging with the right button.

When a path is planned, it will be shown in the CSpace. In figure 11 you can see a thick blue line (in front of the red layers), which stand for the planned path shown in the TSpace of figure 10.

7.4 Advanced Parameter-window

The fourth and last window used in our program is the Advanced Parameter-window, which is shown in figure 12. With this window it is possible to define

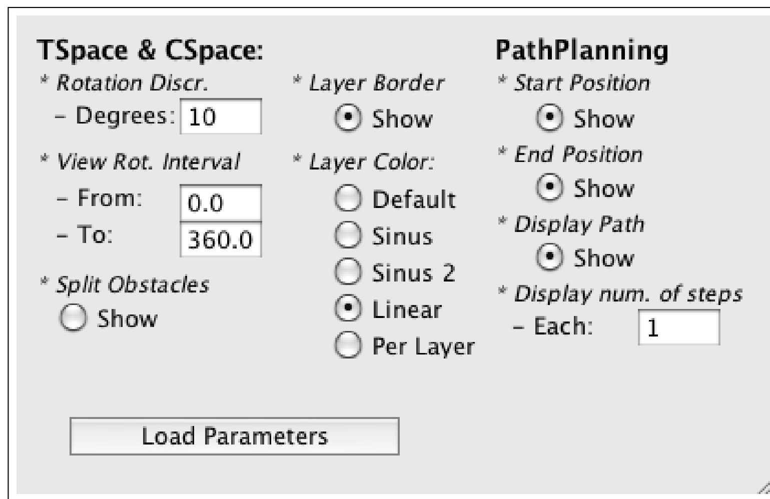

Figure 12: Interface Advanced Paramater window

somewhat more advanced parameters. These parameters are used througout the whole program and can be used to get some more insight in how a particulary problem is solved or represented. The Advanced Parameter-window is split into 2 groups. First there is the "TSpace & CSpace" - group which declares values for the TSpace or Cspace, second there is the "path planning" group, which contains data for path planning.

With the parameters in the first group a few things can be changed. The first thing is "the Rotation Discretization" of the robot. The given value should be in degrees and defines to which angle the angle of the robot will be rounded. Standard this value is defined to 10 degrees, which results in 36 different layers in the CSpace and thus 36 different rotated robot positions. The second parameter is the "View Rot. Interval" which defines the interval in the CSpace of the layers which are shown. A interval of 0 to 360 degrees results in all displayed layers which are in this interval. So with an rotation discretization of 10 degrees, there will be 36 layers shown in the CSpace. Last in the first column is the checkbox to show split obstacles. When checking this box, each time the data in the TSpace is updated, all concave obstacles will be split in convex obstacles and these convex obstacles will be displayed in the TSpace. Figure 13 shows what can be expected when checking this box. Because the concave to convex obstacles will be only updated when hitting the "update"-button in the Tspace, it will happen that when editing

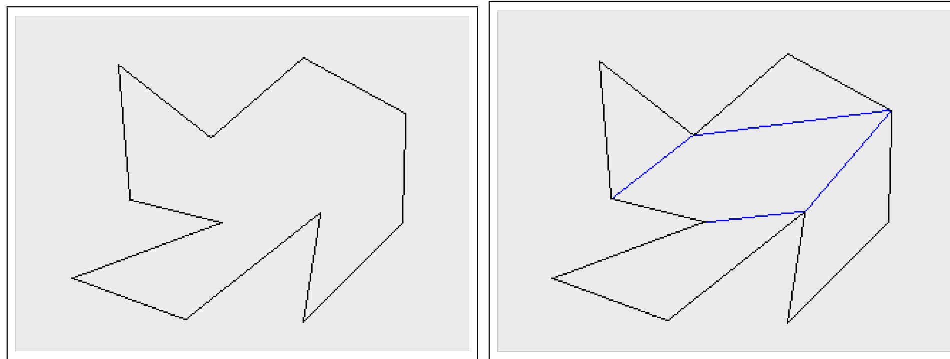

Figure 13: Left: a concave obstacle, Right: obstacle split in convex parts

concave obstacles, the convex result will be temporarily incorrect, this is due to some implementation decisions.

In the second column of the “TSpace & CSpace” - group it is possible to define if the layers in the CSpace do or do not have a (black) border, or to define the type of coloring of the layers in the CSpace. The first choice - “Default” - results in white layers, “Sinus” and “Sinus 2” will color the layers with the color havin a “sinus”-shaped graduation. Linear will result in a linear colored grauduation (an example can be seen in Figure 11) and “Per Layer”, will color each layer, red, blue or green.

Last in this user guide will be treated the “path planning”-column of the Advanced Parameter-window. The first three checkboxes define if the start or end position or the complete planned path has to be shown. The fourth input box defines the step which is shown of each n-steps of a path. Setting this value to 1, all steps in a path will be shown. Setting in to 2, every second step wil be shown, setting it to n, every n-th step will be shown.
When all values in this panel are defined, hitting the “Load Parameter”-button results in an update of the TSpace and CSpace with the new parameters.

References

[1] P.K. Ghosh. A unified computational framework for minkowski operations. 1993.

[2] A.J. Hendriks. Control and tracking of car maneuvers. *Philips Internal Document*, (TN-90061), 1990.

[3] B. Pelegrin J. Fernández, L. Cánovas. Algorithms for the decomposition of a polygon into convex polygons. 1998.

[4] W.B. Dowsland J.A. Bennell, K.A. Dowsland. The irregular cutting stock problem - a new procedure for deriving the no-fit poloygon. 2000.

[5] J.M. Keil. *Handbook of Computational Geometry*, chapter Polygon Decomposition, pages 491–518. Elsevier Science Publishers B.V., 2000.

[6] Karen I. Trovato. *A* Planning in Discrete Configuration Space of Autonomous Systems*. PhD thesis, University of Amsterdam, 1996.