UNIVERSITÄT ZU LÜBECK

Universität zu Lübeck

Institute for Robotics and Cognitive Systems

Master Thesis

An Approach to Solving Object Displacement Problems

written by

Maximilian Mühlfeld (580070)

Supervisor:

Prof. Dr.-Ing. Achim Schweikard

Lübeck, September 14, 2014

Assertion

I assure that the following work is done independently with the use of only stated resources.

Lübeck, September 14, 2014

Abstract

The scope of this thesis grasps the implementation and testing of algorithms to solve object displacement problems.
To achieve this, the search space is divided into multiple cells to reduce its size. Furthermore, instead of calculating the whole space at once, a local approach is used to only calculate the parts needed for the next search step. This needs a wide range of basic geometric and algebraic algorithms for object representation, collision detection and object translation as well as rotation. On the resulting search space a simple and exchangable graph search algorithm is applied, which gives a path in the object space from the start configuration to the desired target configuration.

Contents

1 Introduction

1.1 Motivation

1.1.1 Path Planning in Robotics

In the field of robotics the path planning of an industrial robot can be programmed as a fixed list of movements to be executed. This has multiple drawbacks as for every change in the enviroment the robot needs to manually be reprogrammed.
But what if this robot is equipped with a camera that detects the shape of objects in the robots enviroment. This would give a set of objects at certain positions, a robot arm in a starting position and a target where the robots endeffector needs to work. If then this puzzle would be solved, the robot could be directed in a different way each time without the actual need to access its software.

1.1.2 Geometric Riddles in Gaming

Solving geometric riddles is an amazingly fun task for a human. This is the reason many games simply consists of such riddles ranking from easy to hard in difficulty. But even the easiest riddle for a human proposes a big challenge for a simple algorithm that searches trough the possible ways of solving it. Even more complicated is the generation of such riddles. Even for the human brain this task can be exceedingly stressful.
Now, if there would be a possibility to check if such a riddle has a solution, there would be the option to generate them randomly and check for feasibility. This would cast off the necessity for the developers to manually create each riddle. Also the consumers, in this case the players, would have a endlessly stream of new and different riddles to solve.

1.2 Idea

For simplification purposes the algorithms work on two dimensional object displacement problems. The following riddles shall provide simple examples of the problem:

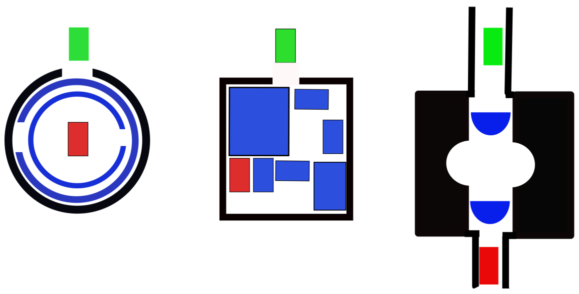

Figure 1.1: *riddle 1 with circles , riddle 2 with boxes , riddle 3 with 2 half-circles*

The riddle is solved, if the red box matches the green one. Blue objects are movable and black ones are stationary. The stationary ones will be referred to as rims hereafter.
As a human the way of finding solutions is quite obvious. The first riddle is solved by rotating the blue objects, the second through translation. The third needs both ways to be solved.
If we want to solve this with an algorithm, we would need to consider each objects collsision with the other objects and the rim. Also we would need to find a way to express the current configuration consisting of position (x,y) and rotation (ϕ) and the direction the main object needs to take. A first idea would be to

1. Generate all valid configurations per object in regard to the stationary obstacles as a configuration space.
2. Create one collision space per object for collision with every other object.
3. Substract the collision space from the configuration space to get a valid space in regard to all obstacles for one object.
4. Divide the space in cells and locate the valid ones.
5. Build a graph out of this starting position by adapting the space after each step.
6. Search in that graph to get a path from the start to the target point.

The target point would then be a simple configuration vector holding each position of each object. A user defined distance function beetween start and target point in that space can then be used as heuristic in the graph search, e.g. $h(start, target) = ||(x_{start} - x_{target}, y_{start} - y_{target})||$.

1.3 State of the Art

In the following part a short overview of the current state of the art algorithms in path planning and collision detecten is given, as these two are the major fields necessary for implementing the final algorithm to solve a geometric riddle. Also a short look is taken into the way complete solutions are present in the possible fields of applications.

1.3.1 Path Planning

The correct answer to the question "Which algorithm is the best for finding the path from start to target" is not as easy as one might suspect. It depends on many factors including which structure one searches on (e.g. graph, net, tree), what knowledge there is about the current position (e.g. distance to target) and what result is needed (shortest path, "good" path, existence of a path).
For this work the focus is on an algorithm working on a graph with non-negativ edges, an absolute knowledge of the position of the target and the current in a two dimensional coordinate system and a need for a "good" path, not necessarily the shortest. Given these conditions, A* [2] is a valid choice. It is a widely used algorithm which calculates a "good" path, not a perfect one. How good the path is depends heavily on the heuristic used. In the given example this would be a weighted sum of the distance traveled in steps plus the absolute distance to the target.
Another choice could be Dijkstra's algorithm [2] itself which is basically A* with the heuristic set to constant zero. This would return the shortest path for all calculations. For the named applications from 1.1 we need the algorithm to be fast, more than to be precise, thus A* is the better choice.
Normally A* would have access to the complete graph to calculate the best way to the target. As this is not the case we need to use an adapted form called RTA* [12]. It introduces a lookahead depth for A*, telling the algorithm to only make a choice using the information gathered by the nodes in the rims to a certain depth of the graph. In our implementation, this value will be set to one, as we move only one object in one direction at a time, thus generating the graph while searching step by step. The drawback is, that the quality of the final path is lower for lower lookahead depths.

Another idea would be to use a potential field method which is quite commen in mobile robotics. This method lays out a grid and for each point a sum is calculated over the repulsive vectors of the obstacles and the attractive target vector [11]. It should be noted though that the algorithm for pathplanning is exchangable. Depending on the choosen method the amount of work needed to adapt the algorithm varies.

1.3.2 Collision Detection

Collision detection in computer science has its home in simulations and computer games as it has in robotics. In this case the focus lies on the way computer games solve collision detection without looking into physical problems that would arise with it.
There are a number of ways this has been solved. In a case where there are not too many objects needed to be considered, pairwise checking is an option. Depending on how the objects are represented, they need to be checked for collision for every step taken. A system was divised by letting only small movements happen in beetween steps [6] to eliminate checking for all objects. Another way around this is to bound objects that lie in a certain proximity of each other together in spheres, where they are only checked in pairs if the spheres containing the objects collide [16].
Also there is a difference in checking if a collision happened, or if it is about to happen. The first option is easy to calculate, because all that is needed is the current position of the objects concerned. If there is the need to calculate the collision beforehand, some information about the movement is needed. As time is not a factor, movement in this scope only refers to position changes in the direction of a vector. Therefore it is possible to calculate the next collision in the direction of the vector to obtain the allowed moving distance.
One way is to start building a spacial binary tree starting from the object, partioning the space along the direction of the movement. Until a given spacial size of an end node is reached (e.g. the bounding box of the moving object) or a obstacle is in the node, the tree keeps on growing. The node directly in front of the last obstacle node would then be choosen as the last safe position to move to. This approach was used in the context of mobile robotic pathplanning [8].
The SWIFT++ algorithm [7] uses bounding boxes around all object. Then a collision check of this bounding box with other bounding boxes in the area is done to remove unnecessary checks. After that simple pairwise checking using the Minkowski sum is done. As this is designed to work with a time axis, steps are supposed to be small, thus it does not suit the needs of this work entirely.
What is needed to prevent a collision, is the distance to the next obstacle. In a work by Ming C.

Lin the distance is calculated using either convex polyhedra and a check for bounds or algreabic functions and check for crossings [13]. In his work this was used to determine if a collision occured, but using the principle ideas one can also calculate the distance to the next obstacle.

1.3.3 Applications

The easiest solution in path planning for industrial robots is to hard-code the correct path. This is mostly done by setting a number of safe points on the path from start to target to avoid a collision. As a matter of fact, this is a good solution for processing objects where only one simple step for a large quantity is needed. In this scenario only few recalibrations are needed. But if the processed objects change more often, each time the machinist needs to recalibrate the robot.
There are already algorithms in work adressing this problem. One algorithm indroduced in the proceedings of IEEE from April 2000 uses Lazy PRM to calculate a collision free path on a grid starting with perfekt path and recalculating edges that include collisions [4].
Earlier work suggests using an approach with local potential field optimization to obtain a good path to the target [3].
The drawback for both algorithms is, that only static obstacles are taken into account. If we use the algorithm introduced in this work to calculate a path, multiple moving objects would be allowed.

Assembly planning describes the process of planning the movements for n objects to fit in n target positions, such that a final object is constructed out of smaller parts. The standart approach uses the finished product as a starting point, trying to find movements to disassemble it into smaller parts. There are multiple algorithms in place to achieve such a plan. A study from Université Libre de Bruxelles suggests using an ordering genetic algorithm in a way, that the sequence of movements is evolved through mutation or crossover and validated until a solution is reached [14]. Another approach would be using motion space representation, where each point in space represents a possible motion of a subassembly [9]. A special interest lies on a geometrical approach to assembly planning [18], as the representation of objects is very close, if not equal in some cases, to the objects in this work. Even though another way of planning the path by decomposing the finished object is presented, it seems possible to adapt the problem of assembly planning to a generic riddle with multiple main objects. As the algorithm in this work is written to compute a path for one main object this is not in the scope of this thesis.

The automated creation of geometric riddles in computer games is a process widely used in the gaming industry. Not only riddles, whole characters and worlds are created at random. One of the first famous games making use of that concept was Nethack [1]. It featured randomized enemy characters in randomized levels. This concept is still in use in todays products. The problem is, that this randomized content is created reversely. For example starting with a valid riddle/ level and then doing only steps from a certain predetermined set of valid transformations one would reach a randomized mutation which can be used as new content.
A newer example is that of Galactic Arms Race, a game using an algorithm called cgNEAT [10]. The main idea of this algorithm is to evolve from existing game content in a matter similar to the evolution of species in biology. This is a huge improvement in contrary to the pseudo-random generated content in older games. But still it needs an initial game content to start from.
The way one could create content with the algorithms in this work is very different. There would be the option to place "totally" random objects into the riddle and afterwards check for the existence of a solution. "Totally" is relativ because there is still the need to look at the given properties of the riddle. For example if the riddles total space should only be 16x16 units big, putting an object the size of 1x30 units in would not be possible.

2 Enviroment and Basics

2.1 Programming Tools

As the target of this thesis is a mere proof of concept, the programming language of choice is matlab [15]. This is a reasonable decision because in the matlab language a huge part of the needed functionality concerning simple mathematical functions is already implemented and easy to use.
Pros:

- easy to use mathematical function.
- fast and easy way to enter data.
- good and simple ways of debugging and locating errors.

Cons:

- slow computation speed.
- needs translation in other language (e.g. c++) for further useage.

As versioning tool "git" [5] is used together with www.github.com as an open source storage platform for the resulting code.

2.2 Basics

For better understanding of the following chapters the central mathematical formulas are repeated here. For the minkowski sum and convex hull [17] should give more information.

2.2.1 Basic Vector Math

If the objects are represented as a list of corner points, vector math comes in handy when describing the borders of the object. Let A be a simple object defined by the following list:

$$A = (0,0;2,0;2,2;0,2)$$

Each pair x,y defines one corner of A and the points are ordered to travel along the border of A counterclockwise. The lines connecting these points will be called the borders of A. They are calculated by substracting the points from each other such that the vectors point along the border clockwise. This leads to:

$$A_{vec} = (2,0;0,2;-2,0;0,-2)$$

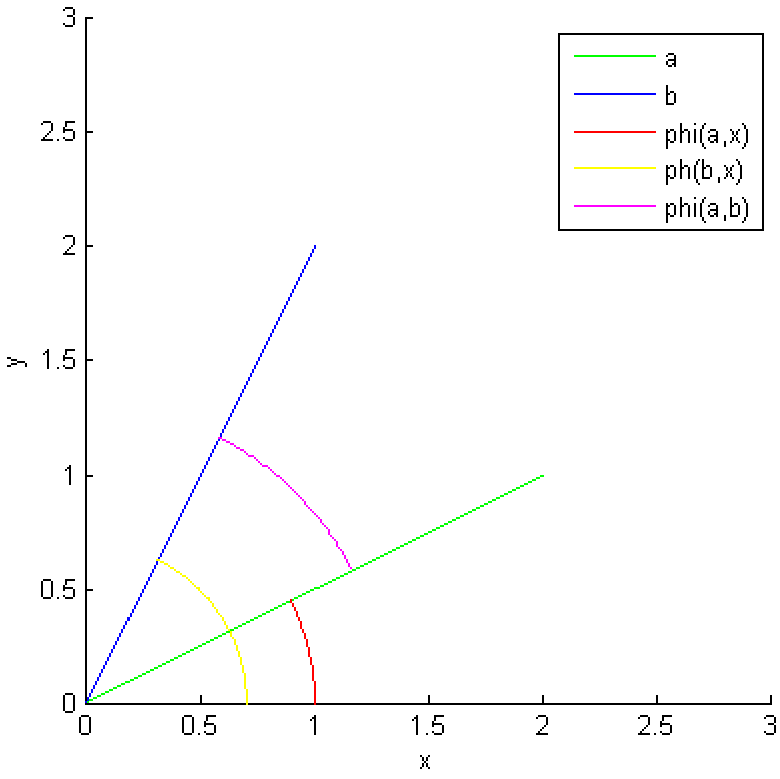

Figure 2.1: *Figure showing the calculation of the angle difference between vectors a and b via reference to x-axis*

Next will be the calculation of the angle beetween two lines a and b. This is done by taking a reference vector, for example the x-axis r=$(1, 0)$. Calculating the angle to that and building the difference in angle results in the relativ anlge difference beetween a and b. This angle tells us which vector lies "below" the other by looking wether its signed or not.
The formula for this is:

$$\begin{aligned}\phi_{a,b} &= \phi_{a,r} - \phi_{b,r}\\ &= tan^{-1}(\frac{a(2)}{a(1)}) - tan^{-1}(\frac{b(2)}{b(1)});\end{aligned}$$

Another vector is necessary as a reference to determine if the difference of the vectors angles is positiv or negativ. This information is needed to build the convex hull.

2.2.2 Minkowski Sum

The minkowski sum is needed to calculate the convex hull beetween two polyhedra A and B. The formula for the Minkowski sum is as follows:

$$A + B = \{\mathbf{a} + \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}.$$

A and B are a set of points in 2 dimensional space defining a convex polyhedron.

2.2.3 Convex Hull

Furthermore we define the center of these objects to be $M_A = (1, 1)$ and $M_B = (0.5, 1)$. The convex hull for A to B is calculated with the minkowski sum of A' and B' with

$$\begin{aligned}A' &= \{-(\mathbf{a} - M_A)|\mathbf{a} \in A\}\\ B' &= \{\mathbf{b} - M_B|\mathbf{b} \in B\}\end{aligned}$$

Basically this means that the points of A are shifted with $-M_A$ and mirrored at M_A before being added to each point of B shifted with $-M_B$ and then reshifted with M_A to get a list C_{temp}

of $4 \cdot 4 = 16$ points.

$$\begin{aligned}
A &= (0,0;2,0;2,2;0,2), M_A = (1,1)\\
B &= (0,0;1,0;1,1;0,1), M_B = (0.5,0.5)\\
A' &= (1,1;-1,1;-1,-1;1,-1)\\
B' &= (-0.5,-0.5;0.5,-0.5;0.5,0.5;-0.5;0.5)
\end{aligned}$$

$$C_{temp} = \begin{array}{lllllllll} (& 0.5, & 0.5; & 1.5, & 0.5; & 1.5, & 1.5; & 0.5, & 1.5;\\ & -1.5, & 0.5; & -0.5, & 0.5; & -0.5, & 1.5; & -1.5, & 1.5;\\ & -1.5, & -1.5; & -0.5, & -1.5; & -0.5, & -0.5; & -1.5, & -0.5;\\ & 0.5, & -1.5; & 1.5, & -1.5; & 1.5, & -0.5; & 0.5, & -0.5) \end{array}$$

$$C_{temp_shifted} = \begin{array}{lllllllll} (& 1.5, & 1.5; & 2.5, & 1.5; & 2.5, & 2.5; & 1.5, & 2.5;\\ & -0.5, & 1.5; & 0.5, & 1.5; & 0.5, & 2.5; & 0.5, & 2.5;\\ & -0.5, & -0.5; & 0.5, & -0.5; & 0.5, & 0.5; & -0.5, & 0.5;\\ & 1.5, & -0.5; & 2.5, & -0.5; & 2.5, & 0.5; & 1.5, & 0.5) \end{array}$$

Each line represents one point of A with the points of B added. By choosing the outer points from C_{temp} and put them in C, such that no point from C_{temp} is still outside C we will get an object C which defines the space around B which the object A can not enter or they will collide. The points can be choosen by searching for the combinations of minimum/maximum x and y coordinates in C.

$$C = (-0.5,-0.5;2.5,-0.5;2.5,2.5;-0.5,2.5)$$

2.2.4 Pathfinding Algorithms on Graphs

Pathfinding in general describes the way of finding a path beetween two points in a graph or net of nodes. The two major algorithms currently in use are A* and Dijkstra's algorithm. As A* is a variant of Dijkstra's algorithm, Dijkstra's will be explained first and afterwards the differences will be highlighted.

Dijkstra's algorithm uses a weighted graph and begins on a starting node which is connected to a set of adjacent nodes. These adjacent nodes are the starting rim. Dijkstra works in these steps:

1. Take unvisited node with shortest distance to the start from rim and check if it is the target node.

2. If not target node, try to add all adjacent nodes to rim.
    - If adjacent node is not in rim, insert the node with its predecessor and distance.
    - If adjacent node is in rim check distance:
        - If adjacent nodes distance is higher than node in rim, discard said node.
        - If adjacent nodes distance is smaller than node in rim, update predecessor and distance.
3. Mark node as visited and repeat.

As long as it is guaranteed that no negativ distance occurs in the search graph, Dijkstra is bound to return the shortest possible path.

A* is a variant which adds a heuristic function measuring the distance beetween the current node and the target. The heuristic and the distance to the start build a weighted sum. This results in a faster exclusion of possibly wrong paths, but it no longer guarantees to find the optimal path.

3 Concept

3.1 Common Defenitions and Representations

3.1.1 Definition of the Configuration Space

As the exact representation of our objects should not matter, we will only define the common points needed for a clear communication of the stated problem.
The algorithms aim is to tell if there is a solution possible and, if so, present it. The object which needs to be moved from a starting configuration to a target configuration will be named main object M. Other movable objects are obstacles named Ob_i and the stationary objects are called rims R_i. This will be combined to the sets $O = \{M\} \cup \{Ob_i | i \in \mathbb{N}\}$ and $R = \{R_i | i \in \mathbb{N}\}$.

Each object O_A contains some data for representing its shape stored under O_A.data. Furthermore the configuration of O_A is given by the vector (x_A, y_A, ϕ_A) and stored under O_A.mid as the middle/reference point for said object where x_A and y_A gives the point around which the object will be rotated by ϕ_A. By substraction of the first two dimensions occupied by R from the possible space O_i per object in O, and, if we divide the space in two, selecting the one in which $O_i.mid$ is located at the start, we get a valid space C_{O_i-R} for the object O_i to be moved in (not taking into account other objects). This space is a simple 3 dimensional space with x_i, y_i and ϕ_i as base.
But as there is the need to check for collision with ALL other objects $O' = \{O_j, | j \neq i \wedge j \in \mathbb{N}\}$ we need to increase the dimensions of all spaces C_{O_i-R} by the number of objects in O'. Also for each set j of dimensions (x_j, y_j, ϕ_j) added, we will need to substract the current position of the corresponding object O_j from the space, such that all collision points are removed from C_{O_i-R}. This will give us the configuration space for object O_i, C_{O_i}.

3.1.2 Building the Configuration Space

To build such a configuration space, every possible configuration of every movable object needs to be calculated. Even those who are NOT valid need to be computed at least once, to check if they are valid or not.
Each object A has three dimensions (x_A, y_A, ϕ_A), with x and y beeing a finite range from $r_x = [x_l, x_h]$ and $r_y = [y_l, y_h]$ defined by the stationary rims of the riddle. The rotation component ϕ is choosen from a set of angles $\Phi = \{\phi | \phi \in r_\phi = [1, 360]\}$. As this would lead to an infinite amount of possible x, y and ϕ, we could allow steps only, e.g. $x, y, \phi \in \mathbb{N}$.
If there are n movable objects we get a total number of possible combinations in the range of $(r_x \cdot r_y \cdot r_\phi)^n$. Under the premise of saving every calculated combination so that we only calculate each set once and assuming that the time t_s needed for collision check and calculating one set is constant, we get the following formula for the time to calculate the complete configuration space.

$$T_conf = (r_x \cdot r_y \cdot r_\phi)^n \cdot t_s$$

Now we define $t_s = 0.1ms$ set $r_x = r_y = 10$ and $r_\phi = 180$ with only two objects (one main object M one obstacle O) meaning $n = 2$ we get

$$\begin{aligned} T_conf &= (10 * 10 * 180)^2 \cdot 0.1ms \\ &= 324000000 \cdot 0.1ms \\ &= 3.24 \cdot 10^7 \cdot ms && = 9h \end{aligned}$$

This means we would need to wait 9h to completely calculate all possible positions on a very raw grid (each step just 1 unit) without even having started to search on it.

So the idea is to interleave search and building of the configuration space in such a way, that only the needed nodes are calculated and checked. But still this would be very slow if we consider implementing it on such a grid. Therefore another approach is needed.

3.1.3 Dividing the Space in Cells

Instead of taking a grid, the configuration space is divided into cells depending on the current positition of the objects. This cell division will heavily decrease the number of search nodes in the space. The drawback on the other hand is, that this division is dependent on the object representation. So instead of computing an independent configuration space for the search, the search needs to use some information from the objects. Thus a combination of a set of collision information for each point in the configuration space is calculated for each search step. This will lead to an integration of search algorithm and object representation into one main algorithm.

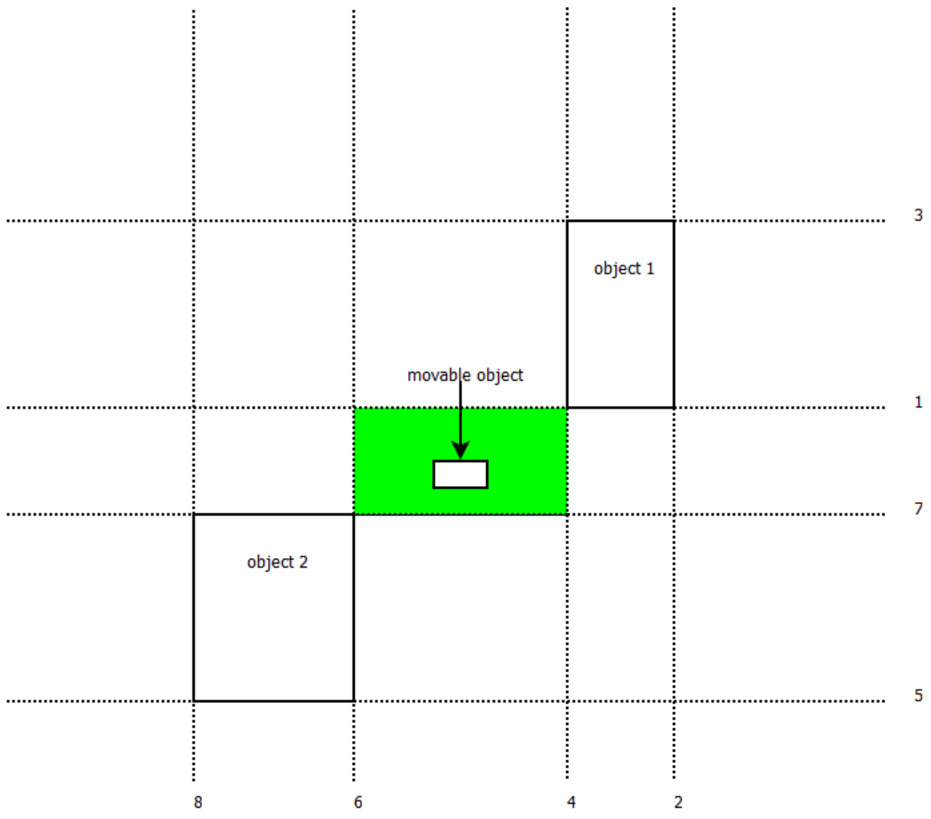

Figure 3.1: *Figure showing two objects and their extended borders*

So to identify in which cell a point p is in, one takes the borders of each object and extends them beyond their normal length. Doing so will split the space in 2 for each border extended. So if we say we have n borders to extend, we can check if the point p is under/over (right/left) of those borders, leading to a binary vector with each position being 1 if p is over the border, 0 otherwise. This vector can be seen as a binary number, thus each cell on the plane can be identified by a single number I. In figure 3.1 the moving objects binary vector would be $(01011111) = 95$.
This only works for one point on an otherwise static field, so for each moving object, their cell position needs to be calculated seperatly. Also rotations propose a problem, because the borders of an rotated object may change the cells, but they dont need to. So if there is the need to

rotate an object, there is no telling the difference beetween object O_1 with configuration $(x, y, 0)$ and O_2 with configuration $(x, y, 1)$ if the cells for all moving objects stay the same.
Thus each possible rotation ϕ_i for each moving object O_i with $i \in \mathbb{N} \wedge i \leqslant n$ is taken into account by adding their value to the cell identifier. One configuration can then be seen as $(I_1, I_2, ..., I_n, \phi_1, \phi_2, .., \phi_n)$.
The exact way of checking for borders to get a simple search graph depends on the implementation. In the following part two ways of representing the objects will be used.

3.1.4 Objects as Point List

One of the possible ways of describing an object in a two dimensional setting would be an ordered list of corner points. Together with an anchor point we can calculate all transformations needed.

Main steps

To see if this representation would work we take a short look at the algorithm described in 1.2 and sketch a solution to each step.

1. Generate $C_{O_i - R}$:By identifying the main outer rim $R_m o$ and computing its inner hull for O_i the space C_{O_i} is build. For all other rims R_j computing the convex hull with O_i and substracting them from C_{O_i} leads to $C_{O_i - R}$.
2. Generate $C_{O_i - O_j}$: Calculate the convex hull from O_i to each other object O_j.
3. Generate C_{O_i}: Substraction of $C_{O_i - R} - C_{O_i - O_j}$ for all $O_j \in O \wedge j \neq i$.
4. Divide search space in cells: By extending the vectors connecting the convex hull in $C_{O_i - O_j}$ we get a seperation of the space in C_{O_i} in multiple parts. Each step is a translation of the object O_i from one cell to another. The neighbour cells can be identified by iterating over the objects O_j and calculating the nearest crossing along (x, y) with the extended vectors of its convex hull.
   Rotations are represented as a jump from one hyperplane to the next in search direction. There are multiple problems with rotation in this representation, that will be discussed later.
5. Construct the search graph: While moving along those cells, we adapt C_{O_i} for each $O_i \in O$ each step. These cells are then added to the graph.

6. Search for target: Again independently of the object representation a search can then be applied to the resulting graph.

So far this representation seems like a good choice in multiple ways with some drawbacks on the other hand.
Pros:

- Simple and intuitive representation of object itself
- Easy and fast to compute concerning the transformation of an object (translation, rotation)

Cons:

- Rotation needs discrete steps along the config dimensions ϕ_i, therefore more exact calculations lead to higher need in computation power.
- The more corners an object has, the more points its convex hull with other object will have. One point more in object O_i can lead to n more points in $C_{O_i - O_j}$ with n beeing the number of points in O_j. As we use the vectors connecting the points in $C_{O_i - O_j}$, n more cells will arise in the search space due to those ghost planes.

In the following this algorithm will be named pList.

3.1.5 Objects as Function List

Another option of describing an object is the representation with functions and definition ranges. Each function is then represented as a list of coefficients a, b, c describing the polynom ax^2+bx+c. Also an anchor point as a reference is needed for rotating the object.

Main steps

The algorithm slightly changes for this representation, solving problems that existed with the point list representation, but introducing new ones. One step of moving an object in one direction would be described as follows:

1. The object O_i is moved function by function. So for each function f_{o_i} describing a border, this function will be moved in the searching direction. By eliminating all functions that are not in the way by looking at the definition/value ranges, a lot of computing time is

saved.
Iterating over all other objects $O_j \in O \wedge j \neq i$ and their functions f_{o_j} and solving $f_{o_i} = f_{o_j}$ then yields a solution with information about the distance in search direction. The new configuration for the object O_i can then be computed.

2. The function with the minimal distance to the object is saved as the closest function together with the new configuration. If no function could be found, the object is moved to the border in the searching direction.

3. The whole object will then be transformed according to the configuration saved, or if the object is already able to be moved to the target configuration in a direct line, the program ends. This is done by connection each corner of the main object with the target area and checking those functions for intersection.

As noted, this representation solves some problems from the previous one but still carries some drawbacks.
Pros:

- The collision set of a point in the configuration space does not need to be computed, as it is defined directly by the anchor points of the objects itself. The outer rim R_{mo} is considered seperate from the objects. Each non-moving object is treated just like a normal object, only that its anchorpoints values in the configuration space along x, y and ϕ are constant.
- With the ability to get rid of functions that are not in the way, the number of search cells goes down due to the absence of ghost planes.

Cons:

- Having direct points in the config space can lead to multiple nodes in the graph due to small movements that would normaly be compromised by cell representation.
- There is no direct way to represent a vertical line with a function. A workaround is to use a function with a extremly high gradient inside a small definition range. The problem is that this can lead to computational errors due to the fact that the optimal gradient would be infinite.
- Rotation still needs discrete steps along the config dimensions ϕ_i, Therefore more exact calculations lead to higher need in computation power.

In the following this algorithm will be known as fList.

Adaption Using Cells as Sodes

As proposed in 3.1.3 an adaption to the algorithm $fList$ has been made in a way, that the ghostplanes still dont interfere with the movement of the objects, but the number of search nodes in the graph decrease for few object in the riddle.
To do so, each point that has been found a valid next step checked for its cell. This cell information is then used as the next node in the search graph. If another step lands in the same cell, the two different positions in the cell are evaluated, and if necessery updated according to the given heuristic, to change the cells distance value to the target.This way all points in the cell amount to the same node in the graph, thus getting rid of many small movement changes.
The drawback on the other hand is, that for many objects in the riddle, the amount of existing cells can lead to unnecessary updates and thus unnecessary movements as the algorithm jumps back to a previous node that now has a better value.
In the following this algorithm will be know as fCell.

4 Implementation

4.1 Algorithms and Functions

4.1.1 Main Search Loop

As proposed in 3.1.2, we create the graph while searching. This requires a certain level of interleaving beetween the search algorithm and the way we work with our objects. In this example, the functions working on the objects are $oneStep(...)$ and $isValid(...)$. All of the rest is needed for the search algorithm, in this case a simple A*.

```
%while target is not in rim
while(~ismember(target,R))
    %select current node from rim
    next = getNodeFromRim();
    %calculate rimnodes of current node
    for i=1:length(directions) % find next node for each search direction
        [possible_next,next_collision_set] = oneStep(next,....);
        if isValid(possible_next,riddle.b,next_collision_set) % check if node
            is valid and...
            if ~isInRim(possible_next,R) % ...not in the rim
                R=[R;possible_next]; %if so, add it
                collision_set{length(D)+1} = next_collision_set; %set his
                    collision information
                D=[D;D(next_position)+0.1]; %enter distance to predecessor
                P=[P;next]; %enter next as predecessor
                H=[H;heuristic(possible_next,target)]; %calculate heuristic
                    value
                V=[V;0]; %mark as not visited
            else                        %... already in the rim
                pn_position=find(possible_next,R); %search the node in the rim
                if(D(pn_position)>D(next_position)+0.1) % if node in rim is
                    further away as new node...
                    D(pn_position)=D(next_position)+0.1; %... update distance
```

```
                        P(pn_position,:)=next; % and chance predecessor
                    end %else do nothing
                end
        end
    end
```

isValid(...) is a simple function testing the given configuration from *oneStep*(...) against configuration space. This is done by simply checking if the configuration point is inside any convex hulls stored in the configuration space.
The functions *oneStep*(...) is now explained in detail for the representation of objects as points list and function list.

4.1.2 Implementation with Point List

As described in 3.1.4 an object can be described as a list of ordered corner points. Starting from there the collision set for each object is the set of convex hulls of its collision with each other object. A cell can then be described as a valid set of collision sets for all objects.
These cells form the search nodes for the pathfinding algorithm.

The Function *oneStep*

$[\mathbf{nextNode}, \mathbf{newCollSet}]$ = *oneStep*(**node, direction, collSet,riddle, jump_over**) is the function that provides us with the next node in a specific direction and an updated collision set for that node. This new collision set fits to the new point in the configuration space at the position of the next node.
Parameters:

- **node**: a point in the configurationspace C used as the starting point.
- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **collSet**: i sets of $n-1$ sized sets giving the collision sets $C_{O_i - O_j}$ per object O_i for all n objects.
- **riddle**: the original set of information for the starting riddle. Needed for recalculation of collSets.
- **jump_over**: flag for signaling if next node should be on the rim of current cell (jump_over

= -1), or in the next cell (jump_over = 1).

Returns:

- **nextNode**: the next point in the configuration space from **node** along the dimension **direction**.
- **newCollSet**: the new collision set for the objects at the configuration **nextNode**.

The fucntion can be divided in two different step types: rotation and translation.
All rotations are executed by changing the dimension **direction** depending on the sign of **direction** by one step. The size of the step is determined by the needed accuracy. The function rotateObject rotates all cornerPoints of the object around its anchor point.

```
...
    tempAdd = zeros(1,length(node)); %build mask for step in rotation
        direction
    tempAdd(abs(direction))=sign(direction)*rotationStep; %roationstep is set
        depending on the needed accuracy
    nextNode = node + tempAdd; % add mask to old node

    for object=1:length(riddle.o)
        riddle.o{object} = changeOneObject(nextNode((object-1)*3+1:object*3),
            riddle.o{object});    %change object according to its new rotation
    end

    for object=1:length(riddle.o) %for each object
        temp = riddle.o;
        temp(object) = [];
        newCollSet{object}= getRims(riddle.o{object}.data,temp,... %generate
            new collision sets
            length(riddle.o{object}.data),riddle.o{object}.mid);
    end
    return; % and return
```

On the other side stands the translation step. As our goal is to identify the next cell in the dimension **direction**, we iterate over all convex hulls stored in **collSet** for the object that needs changing and extend their borders. A jump over/to the nearest border in a given direction equals a jump into the next / to the border of the cell.
First two points are taken from the convex hull (named points) to recalculate the vector. Due to the fact that each object only resides in a 2 dimensional space, by solving a linear equation the

crossing points on the dimension that is NOT our search dimension *direction* can be obtained, so that the point in this direction can be calculated.

```
...
        %calculate line from points
        offset = points(i,:);
        vector = points(mod(i,length(points))+1,:)-points(i,:);

        %solve for x and y points
        x =  vector\(node((object_pos-1)*3+1:(object_pos-1)*3+2) - offset) ;
...

...
        %get point on same x,y coordinate
        %check if line is parallel to searching direction
        if(vector(mod(mod(abs(direction),3),2)+1)<0.001)
            continue;
        end

        %get x to move in y direction and otherwise
        if(mod(abs(direction),3)==1)
            if(x(2,2)<0.001)
                continue;
            end
            p = offset + x(2,2)*vector;%get point on same y
        else
            if(x(1,1)<0.001)
                continue;
            end
            p = offset + x(1,1)*vector;%get point on same x
        end
```

Before we calculate this point, we check if we already reached our target cell by trying to get a non-intercepted connection from the current **node** to the target node taken from **riddle**. The flag needs to be checked for ALL vectors.

```
    %check if point is in same cell as target
        if(inTargetCell)
            %find out if direct way to target is possible
            temp=(node(1:2) - offset)';
            A=[vector', -node_to_target];
            sol = A\temp;

```

```
          %check if lines intersect (aka way to target is free )
          if(sol(1)>=0 && sol(1)<=1 && sol(2)>=0 && sol(2)<=1)
              inTargetCell = inTargetCell && false;
          end
      end
```

If all checks out, we calculate the vector from **node** to the crossing point on the border. If the sign of the vector along the dimension **direction** equals that of **direction** a distance is calculated and, if lower than the current minimum, stored together with a possible new node **nextNode**. This next node is choosen depending on the flag **jump_over** to either be directly in front or behind the point on the border.

```
      %vector from node to temp point on line
      node_to_point = p-node((object_pos-1)*3+1:(object_pos-1)*3+2);
      %get distance to those points if direction is ok
      if sign(node_to_point(mod(abs(direction),3)))~= sign(direction)
          d = inf;
      else
          d = norm(p - node((object_pos-1)*3+1:(object_pos-1)*3+2));
      end

      %save new minimum and new point on line
      if d < min_dist
          min_dist = min(min_dist,d);
          if (jump_over==1)
          nextNode((object_pos-1)*3+1:(object_pos-1)*3+2) = p + (
              node_to_point~=0)*0.001;
          else
          nextNode((object_pos-1)*3+1:(object_pos-1)*3+2) = p - (
              node_to_point~=0)*0.001;
          end
      end

...
```

Now after this loop has finished, we have found either a new minimum in the searching direction or we are inside the same cell as the target.
If we are in the same cell as the target, we just set the main object M to the targets configuration and return.

```
if inTargetCell && object_pos==1
    nextNode(1:3) = riddle.t.mid;
    return
end
```

If we found a new minimum, the collision sets need to be adapted. For that we recalculate the new configuration state **nextNode** of our objects with the help of *changeOneObject*(...).

```
   for object=1:length(riddle.o)
        riddle.o{object} = changeOneObject(nextNode((object-1)*3+1:object*3),
            riddle.o{object});
    end
```

Afterwards we iterate over the newly generated objects and recalculate the collision sets per object with *getRims*(...).

```
for object=1:length(riddle.o)
        temp = riddle.o;
        temp(object) = [];
        newCollSet{object}= getRims(riddle.o{object}.data,temp,...
            length(riddle.o{object}.data),riddle.o{object}.mid);
end
```

This **newCollSet** is then returned together with **nextNode**.

4.1.3 Implementation with Function List

The implementation as function lists 3.1.5 describes the configuration space in the same way as the point list. The collision set on the other hand is defined directly by the objects and their positions. After each step, the object that was moved will be updated and a copy of its set is stored as the new collision set for the configuration.

The Function *oneStep*

[**nextNode**, **collision_set**] = *oneStep*(**node, direction, cur_collision_set,riddle**) is the function that provides us with the next node in a specific direction and an updated collision set for that node. This new collision set fits to the new point in the configuration space at the position of the next node.
Parameters:

- **node**: a point in the configuration space C used as the starting point.

- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **curr_collision_set**: i sets of $n-1$ sized sets giving the collisionsets $C_{O_i-O_j}$ per object O_i for all n objects.
- **riddle**: the original set of information for the starting riddle. Needed for recalculation of collSets.

Returns:

- **nextNode**: the next point in the configuration space from **node** along the dimension **direction**.
- **collision_set**: the new collisionset for the objects at the configuration **nextNode**.

In comparison to the header of the other *oneStep*(...) implementation, only the parameter **jump_over** is no longer needed.
The function can again be divided into two different steps: rotation and translation.
All rotations are executed by changing the dimension **direction** depending on the sign of **direction** by one step. The size of the step is determined by the needed accuracy. The function *rotateFunc*(...) recalculates the function parameters after the rotation.

```
...
    tempAdd = zeros(1,length(node)); %generate mask
    tempAdd(abs(direction))=sign(direction)*rotationStep; %rotationStep
        depends on the needed accuracy
    nextNode = node + tempAdd; %adapt old node

    %rotate object to fit new configuration
    curr_collision_set{object_pos} = rotateFunc(nextNode((object_pos-1)*3+1:
        object_pos*3),curr_collision_set{object_pos});
    collision_set = curr_collision_set; % save new collision set
   return; %and return
```

On the other side stands the translation step. Due to the fact, that there are no cells, the goal is to move the object in the **direction** until it collides with either the border or another object. This is done by checking all elements in the **curr_collision_set** for collision depending on the **direction**. If none could be found the border of the **riddle** is used.
As the object to be moved is defined by **direction**, all other objects need to be checked for a possible collision. Therefore a loop iterates over all functions in all objects in **curr_collision_set**,

checking if the moving object can be moved to the target without collision and calculating the configuration node if a collision exists in **direction** by calling $moveToFunc(...)$

```
...
  if(object_number==object_pos) %skip if object is moving object
        continue;
    end
    %pick object
    object = curr_collision_set{object_number};

    %pick function from object
    for function_number=1:length(object.coeff)
        func=object.coeff{function_number};
        def=object.def{function_number};
...

...
        %% get nextNode in search direction. if function not in the way, dist
            = inf.
        [tempNode,dist] = moveToFunction(node,direction,curr_collision_set{
            object_pos},riddle.b,func,def,object.above{function_number});

        %save new minimum ( closest function in the way ) and nextNode
        if abs(dist) < abs(min_dist)
            min_dist = min(abs(min_dist),abs(dist))*sign(dist);
            nextNode=tempNode;
        end
...
```

In the same loop a check is done to see if the function lies in the direct way of the object to the target in **direction**. Before the loop a function is constructed connecting the corners of the moving object with the corners of the target. The whole test is discarded if the moving object is not the main object.

```
...
inTargetCell = object_pos==1; %only check for target cell if main object is
    moved
if(inTargetCell) %if main object, calculate connection to target
    object = curr_collision_set{object_pos}; %pick main object
    %connection = cell(length(object.coeff),1); % init connection array
    px = object.def{1}(1);
    for i=1:length(object.coeff)
        if px==object.def{i}(1); %pick next x-coordinate ( move along border )
```

```
            px=object.def{i}(2);
            tx=riddle.t.def{i}(2);
        else
            px=object.def{i}(1);
            tx=riddle.t.def{i}(1);
        end
        %calculate y-coordinate
        py = object.coeff{i}(1)*px*px + object.coeff{i}(2)*px + object.coeff{i
            }(3);
        ty = riddle.t.coeff{i}(1)*tx*tx + riddle.t.coeff{i}(2)*tx + riddle.t.
            coeff{i}(3);
        connection.coeff(i)={[ (py-ty)/(px-tx) , py - (py-ty)/(px-tx)*px ]}; %
            set function
        connection.def(i)={[px tx]}; %set definition range
    end
end
...
```

Then inside the loop this function is check for collision with the choosen function *func*. After that if a node with the closest distance was choosen the collision set is recalculated. If not *moveFunc*(...) moves the object to the border of **riddle**. And if the object can be moved to the target without collision, the target will be set as the next node and the function returns.

```
...
%% build new collision set from chosen node
curr_collision_set{object_pos}=moveFunc(min_dist,direction,curr_collision_set{
    object_pos});
collision_set = curr_collision_set;

if inTargetCell && object_pos==1
    nextNode(1:3) = riddle.t.mid;
    return
end
```

The Function *moveToFunction*

[**nextNode**, **dist**] = *moveFunc*(**node, direction, object, border, func, def, above**) checks if the function **func** defined in the range **def** lies in the way of the **object** trying to move in the **direction** or if the **border** needs to be used. The function calculates the configuration point **nextNode** together with the travelled distance **dist**.

Parameters:

- **node**: the current node the movement should be executed from.
- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **object**: the object which wants to be moved.
- **border**: the border of the riddle.
- **func**: the function which could be laying in the way.
- **def**: the definition range of the function.
- **above**: wether the object the function **func** borders lies above or below **func**

Returns:

- **nextNode**: the next configuration point after moving the **object**.
- **dist**: the distance the **object** has been moved.

The function *moveFunc*(...) iterates over all functions in **object** and checks if there exists a crossing point beetween them and the function **func**. This point is then saved together with the distance and the point with the minimal distance is returned.

Along x-Direction

First the values of the function **func** are calculated and if the movement is along the x-coordinate, compared to those of the **object** functions. If no common range exists, the functions can not collide.

```
...
%calculate funcValue at ends of defenition range
funcValue_min = (def(1)^2)*func(1) + def(1)*func(2) + func(3);
funcValue_max = (def(2)^2)*func(1) + def(2)*func(2) + func(3);

%check for max/min
if funcValue_min>funcValue_max
    temp = funcValue_min;
    funcValue_min=funcValue_max;
    funcValue_max=temp;
end
```

```

for obj_function_number = 1:length(object.coeff)
    %get current function and range
    obj_func = object.coeff{obj_function_number};
    obj_def = object.def{obj_function_number};


    if(mod(abs(direction),3)==1)
        %  move along x coordinate (right/left)
        %% calculate function values
        objValue_min = (obj_def(1)^2)*obj_func(1)  + obj_def(1)*obj_func(2) +
            obj_func(3);
        objValue_max = (obj_def(2)^2)*obj_func(1)  + obj_def(2)*obj_func(2) +
            obj_func(3);

        %% get the y-range used by both function ( inner borders )
        min_y = (objValue_min>funcValue_min)*objValue_min + (objValue_min<=
            funcValue_min)*funcValue_min;
        max_y = (objValue_max<funcValue_max)*objValue_max + (objValue_max>=
            funcValue_max)*funcValue_max;

        if(min_y > max_y) % if no common range exists, jump to next function
            if(sign(direction)==-1)
                diff_min = border(1,1) - obj_def(1);
                diff_max = border(1,1) - obj_def(2);
            else
                diff_min = border(3,1) - obj_def(2);
                diff_max = border(3,1) - obj_def(1);
            end
...
```

If a common range exists, the x values at the edges of the overlapping y values are calculated as $funcX_min$ and $funcX_max$. The same calcuations are also made for the current **object** function to calculate $objX_min$ and $objX_max$. The difference beetween the **object** function and the **func** values are used to determine the distance the **object** needs to be moved towards the function **func**. The minimum distance is then saved together with the new configuration point.

```
...

            diff_min = funcX_min - objX_min; %calculate distances
            diff_max = funcX_max - objX_max;
        end

        if(diff_min==0 || diff_max==0) %check if object lies besides function
            if(sign(direction)==sign(above)) %if object is moving away from
                function
                continue;
            end
        end

        %% choose smaller distance...
        if(abs(diff_min)<abs(diff_max)&&abs(diff_min)<abs(dist))
            nextNode(abs(direction)) = node(abs(direction)) + diff_min; %...
                and nextNode;
            dist = diff_min;
        elseif(abs(diff_max)<abs(dist))
            nextNode(abs(direction)) = node(abs(direction)) + diff_max; %...
                and nextNode;
            dist = diff_max;
        end
    end
...
```

Along y-Direction

If the movement is along the y-coordinate a simple look at the definition ranges of the **object** functions rules out all functions that are not in the way. If that is the case, the border is taken as the nearest function.

```
...

% move along y coordinate ( up/down)
else
        %% get the x-range used by both function ( inner borders )
        min_y = (def(1)>obj_def(1))*def(1) + (def(1)<=obj_def(1))*obj_def(1);
        max_y = (def(2)<=obj_def(2))*def(2) + (def(2)>obj_def(2))*obj_def(2);

        %% check if function lies in the way
        if(min_y > max_y) % if no common range exists, take borders
            %% calculate function values
            objValue_min = (obj_def(1)^2)*obj_func(1) + obj_def(1)*obj_func(2)
                 + obj_func(3);
            objValue_max = (obj_def(2)^2)*obj_func(1) + obj_def(2)*obj_func(2)
                 + obj_func(3);
```

```

    ...
```

If there is a common range, the *objValue*s are calculatet with the common function range and the distance is stored together with the new configuration point.

```
        elseif(min_y<=max_y) %start to find collision

            %% calculate function values
            objValue_min = (min_y^2)*obj_func(1) + min_y*obj_func(2) +
                obj_func(3);
            objValue_max = (max_y^2)*obj_func(1) + max_y*obj_func(2) +
                obj_func(3);
    ...

    ...
            diff_min = funcValue_min - objValue_min; %calculate distances
            diff_max = funcValue_max - objValue_max;

            if(diff_min==0 || diff_max==0) %check if object lies besides
                function
                if(sign(direction)==sign(above)) %if object is moving away
                    from function
                    continue;
                end
            end
        end

        %% choose smaller distance...
        if(abs(diff_min)<abs(diff_max) && abs(diff_min)<abs(dist))
            nextNode(abs(direction)) = node(abs(direction)) + diff_min; %...
                and nextNode;
            dist = diff_min;
        elseif(abs(diff_max)<abs(dist))
            nextNode(abs(direction)) = node(abs(direction)) + diff_max; %...
                and nextNode;
            dist = diff_max;
        end

end
```

5 Results

If we apply the finished algorithm to various test data sets, we can get an impression of what the program is capable of. For each riddle beeing solved, the algorithms are executed a number of times to calculate an average runtime.
In both cases A* was used as the search algorithm on the resulting graph. As a distance measurement the heuristic in form of the 2-norm of the vector between current position and target was added to the stepcount with each step weighting 0.1 units. The starting distance to beetween main object and target 10.1980 units.

5.1 Measurements

The algorithms are described as pList for the implementation with point lists and fList for the implementation with functions. The adaption of the algorithm to work with cells as node identification is called fCell.

5.1.1 Rotation and Translation

First the algorithm is tested against rotation and translation with one obstalce (blue). This object has a fixed position and only rotations are allowed.
The same riddle is then used and solved simply by translation with no rotation allowed for the obstacle. The main object (green) can rotate and move in both cases inside the border (black) to get to the target area (red).

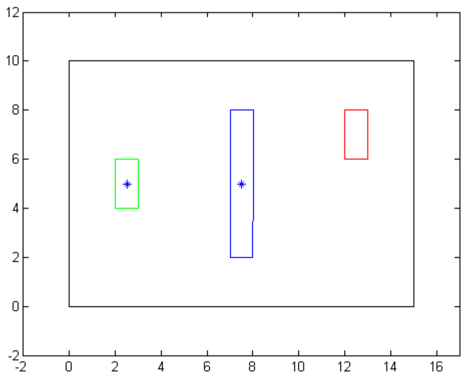

Figure 5.1: *Riddle with main object, border, target and moveable obstacle.*

	rotation			translation		
Algorithm	fastest	slowest	medium	fastest	slowest	medium
pList	18.8819	22.8457	21.0269	0.1211	0.1272	0.1231
fList	0.1176	0.1635	0.1296	0.0291	0.0504	0.0367
fCell	0.3404	0.4012	0.3698	0.2894	0.4112	0.3358

Table 5.1: *Time in seconds for rotation riddle and translation riddle.*

5.1.2 Small Riddles

Two simple small riddles with 2 and 4 moveable obstacles.

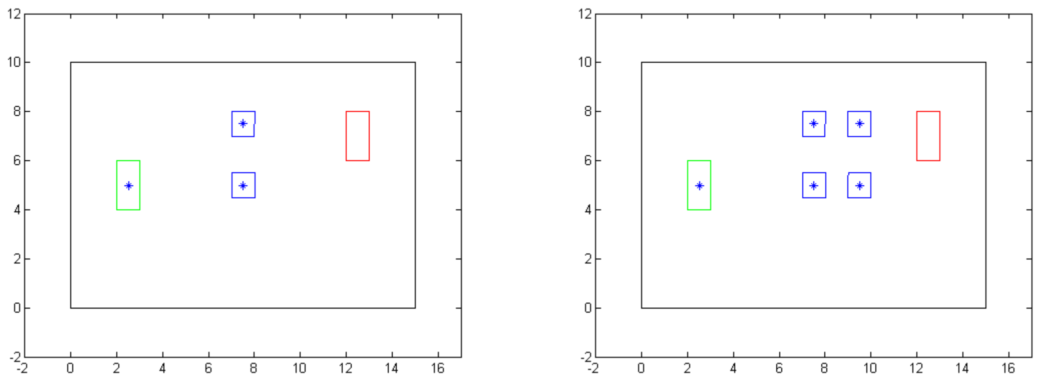

Figure 5.2: *Riddle with two and four movable obstacles.*

	2 objects			4 objects		
Algorithm	fastest	slowest	medium	fastest	slowest	medium
pList	4.7801	5.1930	4.9305	85.1726	90.9629	89.3621
fList	0.6242	0.8082	0.6747	7.3608	8.1557	7.7242
fCell	0.1362	0.1591	0.1472	13.2560	14.0114	13.5623

Table 5.2: *Time in seconds for small riddle with 2 and 4 obstacles.*

5.1.3 Medium Riddles

Bigger riddle with 6 and 8 moveable obstacles.

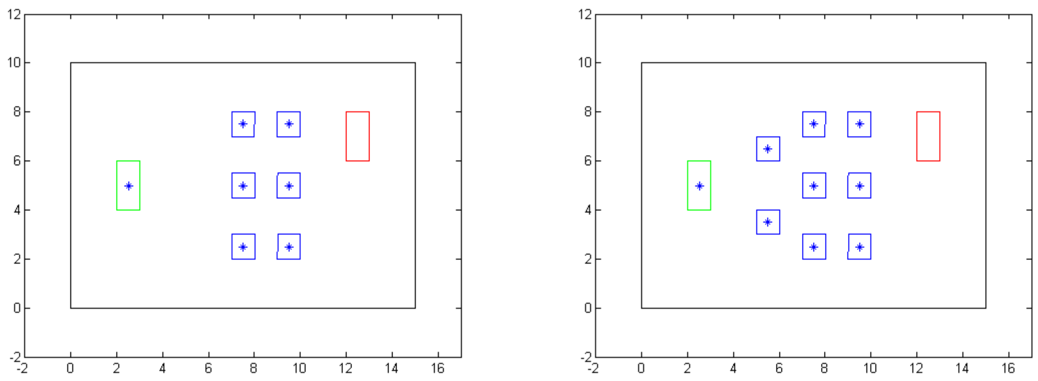

Figure 5.3: *Riddle with six and eight moveable obstacles.*

	6 objects			8 objects		
Algorithm	fastest	slowest	medium	fastest	slowest	medium
pList	395.0670	415.7986	403.9329	1790.5	1972.5	1855.4
fList	32.6039	34.4411	33.7001	30.4606	33.9077	32.1049
fCell	49.1205	51.2846	50.3155	95.5828	106.6468	100.63497

Table 5.3: *Time in seconds for medium riddle with 6 and 8 obstacles.*

5.1.4 Counterheuristic Riddles

This riddle is build to test the search algorithms heuristic in combination cell identification.

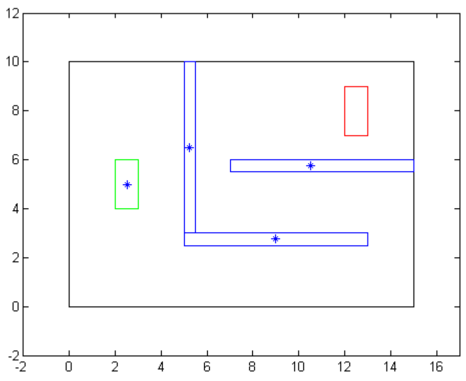

Figure 5.4: *Riddle which needs backward step to be solved.*

	6 objects		
Algorithm	fastest	slowest	medium
fList	34234	35599	34917
fList with cells	24.036	27.8225	25.001

Table 5.4: *Time in seconds for counterheuristic riddle.*

5.2 Interpretation

As we can see the algorithm working with functions is much faster. But still it scales with an increasing amount of objects to work with. If we plot these two against each other we see an exponential grow depending on the number of objects involved for the pList algorithm. The fList algorithms still scale with the number of objects, but are more dependent on the initial positioning of the objects.

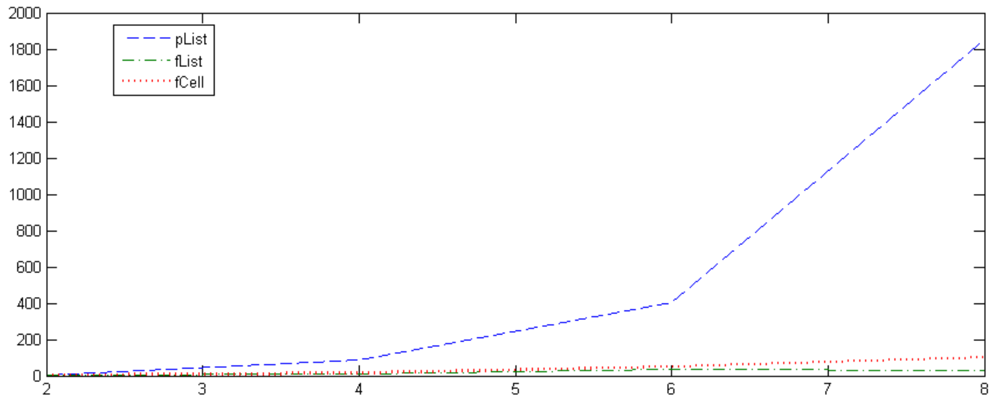

Figure 5.5: *Plot of the tree algorithms showing the time needed to finish depending on the objects involved.*

It can easily be seen, that the algorithm pList with point list representation scales extremely bad with an increasing number of objects. This is due to the fact that not only the objects in the direct way beetween target and starting point are considered, but also the extended vectors of objects that are not in the way. This leads to increased number of cells to search through.

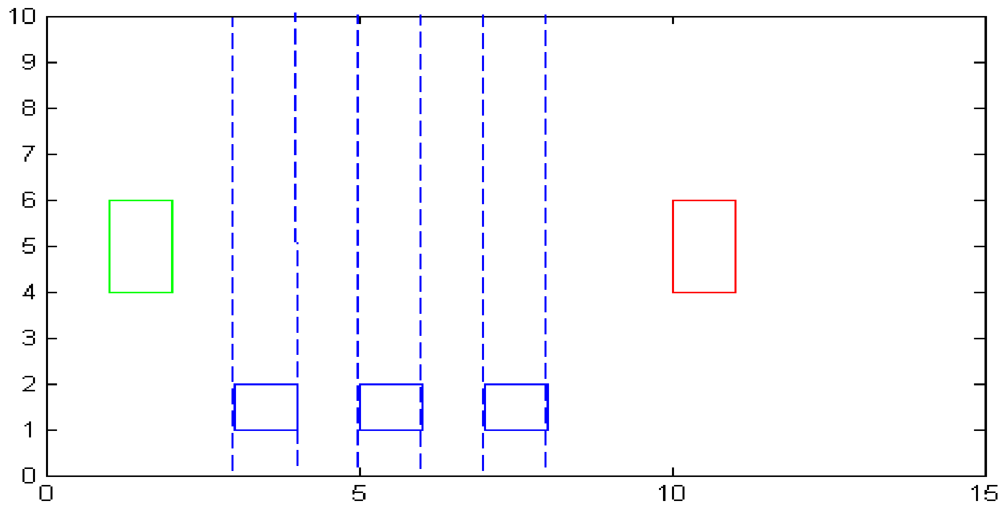

Figure 5.6: *Plot showing the ghostplanes between main object and target.*

In this plot we can see that there are 6 ghostplanes, extending from the borders of the obstacles in blue, beetween the main object in green and the target in red. Due to this $6 \cdot 2 = 12$ unnecessary steps need to be calculated before the final step, as per plane we calculate a position in front and behind before evaluating which position is better.
If this scenario is given to the algorithm $fList$ ($fCell$) with functions, it would simply check if the obstacles on the bottom are in the way. And as that is not the case, no unecessary steps would arise and the algorithm woud finish in 1 step.

Concerning the counterheuristic riddle the data shows that the algorithm $fCell$ is around 1400 times faster than the simpler $fList$ algorithm. The algorithm with $pList$ is not listed here, as it takes more than $16h = 57600$ to finish. The calculation was abonded at that time.
Comparing the two algorithms $fList$ and $fCel$l, it can be seen that $fList$ is faster for normal calculations with many objects, but fails if the heuristik is bad in the pathplanning.
$fCel$l on the other hand reduces the number of cells, thus decreasing the number of steps for the pathplanning. This shortens the path in a way, such that fewer steps in the wrong direction need to be taken to find the target. So for a general purpose algorithm, $fCell$ would be the best choice as it takes only a small penalty for more objects added compared to $fList$, but can deal better with hard riddles.

6 Discussion

Through applying a local approach to building the configuration space, the riddles are starting to become solvable in an acceptable timeframe. Still, the time it takes to solve a riddle can vary strong depending on the number of objects, their start configuration and their shape. Furthermore there are still some unsolved problems in general and depending on the used implementation

6.1 General Problems

6.1.1 Rotation Steps

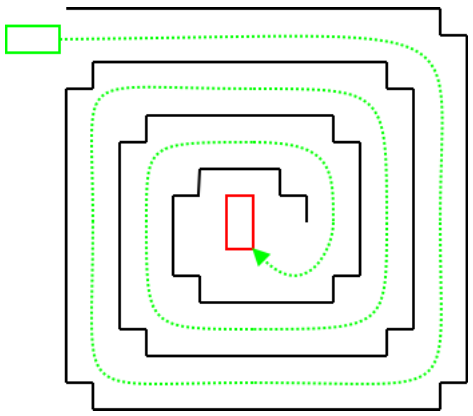

Figure 6.1: *Figure showing a main object (green) with a spiral like obstacle (black) and a target area (red).*

As each object can be rotated a total 360° to get back to its starting orientation, the rotation dimension is finite. If we take a look at figure 6.1 it can be seen that the object can nevertheless be moved and rotated more than 360°. What happens is that as soon as the rotation would be bigger than 360° the rotation is set back to 0°. This can be done due to the fact that the collision

sets containing an object with the configuration (x_1, y_1, ϕ_1) is equal to that of $(x_1, y_1, \phi_1 + 360)$. The problem is, that for algorithm P with points lists convex hulls are needed to check for collision of moving objects. But the objects hull including the rotation axis would be non-convex as the object would be moved along the rotation axis in a sinuid fashion. Therefore the rotation is done on a grid to build a stack of hyperplanes containing the configuration of each object. For every rotation a new hyperplane is created with only the rotating objects rotation changed.
The downside to that is, that in order to build a stack, there is the need to define a stepsize in which to move up and down the stack and change the rotation. This leads to another scaling factor which needs to be set so that adequate accuracy can be achieved. One has to choose either less accuracy for less calculations or more calculations for higher accuracy.

6.2 Problems with Point List Implementation

6.2.1 Invisible Plains

An invisible plain is a plain extending from the vector of one object between two of its points. This plain splits the search space in two parts. Due to the fact, that we use this effect to split the space in cells, which we use for nodes in our search graph, changes to these plains affect our graph.

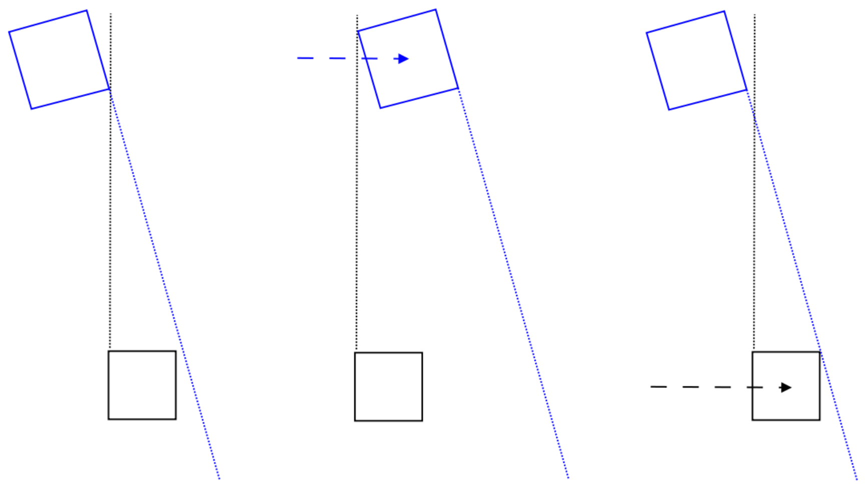

Figure 6.2: *Two objects which planes interfere*

If we take a look at this example, we can see how the black object O_1 dictates the cell border for blue object O_2 on the right. If we move O_2 over the border, it now takes over the cell border for object O_1. Without a given heuristic which object is better to be moved to the right, this will end in alternating movements for both objects.
But if we choose O_1 to be the main object, which has a target on the right, the heuristic would tell our search algorithm that the point with O_1 beeing right of O_2 is the better point, and search from there on further, meaning O_1 beeing moved over the right cell border.

6.3 Problems with Function List Implementation

6.3.1 Vertical Lines

Due to the nature of a function, vertical lines can not be represented. This leads to the rule that all objects in the riddle can not have a vertical line even after rotating. To still be able to draw an object with a vertical line, a function is defined with a very high gradient and setting the starting point at the x-coordinate of the vertical line. This leads to a near vertical line, that is still a function, and as such can be used in the calculations.
The drawback is clearly the loss of preciseness. How strong this loss is depends on the initial gradiant one gives the function. A seemingly good idea would be to use the max value of the variables used in calculations (e.g. double $= 1.7977 \cdot 10^{308}$). But if we do this without being sure that this value will never be used in a way, such that the solution of a calculation could be greater than the gradiant, we would get an overflow.
Thus, a value needs to be found that leaves enough room for calculations and still keeps the precision loss as small as possible. The exact number depends on the system the algorithm is used in. Also one needs to cap the gradiant after each rotation, so that it does not exceed the given max value.

7 Conclusion

7.1 Applications and Further Usage

As already noted in 1.1 there are multiple appliances in the field of robotics and the gaming industry. The exact use is open due to the generalised representation of the problem as a simple riddle to be solved. Thus every practical problem that can be expressed as a geometric riddle in theoretically n-dimensional space can be solved with these algorithms.
Also there is the possibility to further improve the algorithms. One could allow non-convex moving objects by, for example, representing them as a set of convex objects. Or for algorithm $fList$ with function list representing the rotation not as a set of planes, but as a continuously function which would then give the possibility to check for a collision while rotating and as such removing the need to rotate on a set grid of angles.
Another general thing would be a try on implementing the algorithm in a way such that parallel computation is possible. At the moment this would only be possible for each of the n objects movements making the algorithm idealy $n \cdot 3 \cdot 2$ times faster (2 translational dimensions and 1 rotational dimension with increasing/decreasing directions). But as this parallelization depends on the programming language and enviroment used for applying the algorithm it is hard to tell how much of an increase in speed could be obtained.

7.2 Summary

In the creation of this thesis tree algorithms for solving object displacement problems were build, implemented and tested. One as a simple approach with an easy geometric way to understand the representation of world objects and two using an analytic approach making computation easier but losing the simple method of representation.
All algorithms showed an improvement if compared to a solution using a fixed grid to calculate all possible combinations of objects involved in both accuracy and speed. Due to the nature

of the problem, rotations still propose a big problem which needs to be solved for access to a faster solution. Also all problems are bound to have convex moveable objects only. This need for convex objects could be cut if concave objects are build out of multiple convex objects that move together.
From the current data, the algorithm $fCell$ emerged as the all-around best solution to general purpose riddles.This algorithm can be applied to an amount of very different problems, due to the fact that in their core they solve a collision detection problem combined with pathfinding which is needed in many applications.

List of Figures

List of Tables

Bibliography

[1] Nethack.
URL: http://www.nethack.org/index.html.

[2] Aho, A. V., and Ullman, J. D. *Foundations of Computer Science*, vol. C Edition. Computer Science Press, 1995.

[3] Barraquand, J., Langlois, B., and Latombe, J.-C. Numerical potential field techniques for robot path planning, 1989.

[4] Bohlin, R., and Kavraki, L. E. Path planning using lazy prm. In *Proceedings of the 2000 IEEE International Conference on Robotics & Automation* (April 2000), IEEE.

[5] Chacon, S. *Pro Git*, July 2009.

[6] Cohen, J. D., Lin, M. C., Manocha, D., and Ponamgi, M. I-collide: An interactive and exact collision detection system for large-scale enviroments, 1995.

[7] Ehmann, S. A., and Lin, M. C. Swift: Accelerated proximity queries using multi-level voronoi marching, 2000.

[8] Fujimura, K., and Samet, H. Path planning among moving obstacles using spatial indexing, 1988.

[9] Halperin, D., Latombe, J.-C., and Wilson, R. H. A general framework for assembly planning: The motion space approach, 2000.

[10] Hastings, E. J., Guha, R. K., and Stanley, K. O. Evolving content in the galactic arms race video game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG09)* (2009), IEEE.

[11] Hwang, Y. K., and Ahuja, N. A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on 8*, 1 (1992), 23–32.

[12] Korf, R. E. Real-time heuristic search: New results*. In *AAAI-88 Proceedings* (1988), AAAI.

[13] Lin, M. C. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, University of California at Berkley, 1993.

[14] Lit, P. D., Latinne, P., Rekiek, B., and Delchambre, A. Assembly planning with an ordering genetic algorithm, 2001.

[15] MathWorks, Inc. *Matlab Language Reference Manual.*

[16] Palmer, I. J., and Grimsdale, R. L. Collision detection for animation using sphere-trees., 1995.

[17] R&kg, A.-W. Encyclopedia of mathematics.

[18] Wilson, R. H. *On Geometric Assembly Planning.* PhD thesis, Stanford University, March 1992.