Czech Technical University in Prague
Faculty of electrical engeneering

Bachelor thesis

Matej Grajciar

Any-angle path-planing algorithms

Department of Cybernetics
Bechelor thesis supervisor: **RNDr. Kulich Miroslav Ph.D**

Praha, 2012

Prehlásenie

Prehlasujem, že som svoju bakalársku prácu vypracoval samostatne. Všetky podklady (literatúru, SW, zdroje atď.), ktoré som použil sú uvedené v priloženom zozname.

V Prahe dňa ....

podpis

Abstrakt

Táto bakalárska práca skúma výhodnosť any-angle plánovacích algoritmov (Basic Theta*, Lazy Theta*, Phi*, Incremental Phi*). Algoritmy sú testované v rôznych reprezentáciách (mriežka, triangulačná sieť, tetrahedrónová sieť). Všetky zmienené algoritmy sú naprogramované a otestované v rozsiahlych experimentoch. Výsledky any-angle algoritmov sú porovnané s výsledkami najznámejšieho štandardného algoritmu A*.

Abstract

This thesis examines the profitability of the any-angle path-planning algorithms (Basic Theta * Lazy * Theta, Phi * Incremental Phi *). Algorithms are tested in different representations (Grid, triangulation mesh, tetrahedral mesh). All of the mentioned algorithms are programmed and tested in large scale experiments. The results of any-angle algorithms are compared with the results of the most known standard algorithm A*.

Prodakovanie

Rád by som podakoval vedúcemu práce RNDr. Miroslavovi Kulichovi Ph.D za odborné vedenie a poskytnutie cenných rád, ktoré boli pre mňa velmi prínosné ako pri písaní práce tak do života.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The aim of this thesis is to examine the potential contribution of any-angle algorithms to the path-planning.

The objective of path-planning algorithms is to find the shortest path from a start position to a goal position. Due to complexity of the path-planning problem the environment is discretized into a graph, in that are the algorithms computing the path.

While the standard algorithms plan the path only along the edges of the graph, any-angle path-planning algorithms plan the path trough free space (the path is not constrained to the edges) so the path-planning of any-angle algorithms runs in a continuous environment. This property of any-angle algorithms causes that the final path is shorter and more realisticly looking than path found by standard algorithms.

This thesis accrues from the work of Sven Koenig [11] about the any-angle path-planning algorithms. His work describes these algorithms and compares them with the best known standard algorithm, A*. In this thesis, the algorithms are not tested only in the grid representation, as in the work of Sven Koenig but are also tested in other commonly used environment representations for $\mathbb{R}^{\mathbf{2}}$ and $\mathbb{R}^{\mathbf{3}}$ environment.

The objective of this thesis is to examine the advantages of any-angle path-planning algorithms for other environment representations than a grid, while achieving similar results for a grid environment representation as in Sven Koenig's work.

The aim of this thesis is not to develop a perfectly optimized algorithm, but to compare the generated paths and the computation times of any-angle and standard algorithms.

The structure of the thesis is as follow, in Chapter 3 are described the environment representations. The individual algorithms are presented in Chapter 4 and the results of experiments from these algorithm are discussed in Chapter 5.

Chapter 2

Problem definition

The objective of this thesis is to solve a path-planning problem for mobile robotics, where a robot must move from a start position to a goal position in the shortest time. The path-planning problem and it's solution is composed of two parts. The first subproblem is called *generate-graph* [11]. It's aim is to discretize a continuous environment into a graph. The discretized environment where robot moves is called workspace. A workspace is a operation space of a robot. A workspace is divided into two kinds of objects, free space and obstacles. The obstacles are locations in the workspace, trough which is the robot unable to move. There exist many workspace representations : Grids, Voronoi diagrams, Framed Quadtrees, Triangulation mesh, Tetrahedral mesh etc. These representations differ in space complexity, the speed of creation, algorithm heftiness etc. Workspace is transformed into a weighted graph. This graph consists of a finite set of vertices V (points in a workspace), a finite set of weighted edges E (visible paths in workspace), and function ε that assigns to each edge two end vertices (neighbors) [5].

This graph is needed as an input for the subproblem *find-path* [11]. The objective of the *Find-path* subproblem is to search the minimum weighted path from start point to the goal point. Several algorithms can be used to compute the path. Probably the most known algorithm is A*. Other algorithms are typically improvements of A*. A* algorithm is fast enough but the path is constrained only to the graph edges, so the final path looks unrealistic and is longer than necessary. The main property of each any angle algorithm is the line-of-sight check. The line-of-sight check test if exists a visible straight path between two points in the workspace. However Theta*, a variant of A*, propagates information along the graph edges without constraining the path to them. Theta* finds shorter and more realistic looking paths than A*. Theta* is extended to Lazy Theta algorithm, which is much faster than Theta*. Phi* algorithm is a variant of Theta* with slightly differences, such as storing additional data of each point in the graph. This information is further used to speed up the path-planning. Incremental Phi* is an incremental version of Phi*, that us designed for path-planning in changing environment. Incremental Phi* recomputes any-angle paths faster than repeated single shot Theta* because it adjust the path during the movement of a robot.

Chapter 3

Workspace Representation

As mentioned before the workspace can be represented in many ways. The following representations, are used in this thesis: grid, triangulation mesh and tetrahedral mesh.

Grid workspace is a matrix $\mathbb{A} = [n, m]$ the elements of matrix can have value 0-unblocked or 1-blocked. Around each cell of matrix are four vertices, in a grid defined as the matrix $\mathbb{A}$ contains $[n + 1, +1]$ vertices. Each vertex has four crossbar neighbors and eight neighbors (Figure 3.2) [13]. The green dots are the eight neighbors of the black vertex and the green lines are the visible path between them. The blue triangles connected with the black vertex trough the blue lines are the vertex's four crossbar neighbors with visible paths. The edges of a graph G (created from grid representation) are the edges between visible neighbors in grid. This edges represent the visible path for a robot.

Grids are used because it is a fast way to represent a workspace. Terrain can be easily discretized into grid by laying the grid net over the terrain and labeling all cells that are partially or completely obstructed as blocked [11]. The discretization into grid is inaccurate. To achieve a precise discretization into grid, the size of cells must be as small as possible, which causes a higher time complexity by path-planning. In Figure 3.1 is shown an example of robot' workspace. The example of creating a grid from a workspace is shown in Figure 3.2.

A triangulation mesh can be considered as a graph, that is created by the Delaunay triangulation. A triangulation mesh comprises a set of triangles, these triangles are connected by their common edges. However the triangulation mesh represents a workspace for a fraction of the memory than the grid representation, the speed of mesh creation is very high in comparison to grid. The memory saving is caused due to less space complexity. The triangulation mesh does not contain so much points as grid, which is consequence of the polygonal map representation. This representation not only saves the memory but also speed-ups the path-finding algorithm. Polygonal map is defined by one or more polygons, one polygon refers to boundaries of the workspace and other polygons represent the obstacles inside the workspace. A simple polygon is a finite chain of line segments called edges, their endpoints called vertices and without self-intersection. In Figure 3.3 can be seen a polygonal map created from the workspace shown in Figure 3.1 and also the triangulation mesh created from this polygonal map.

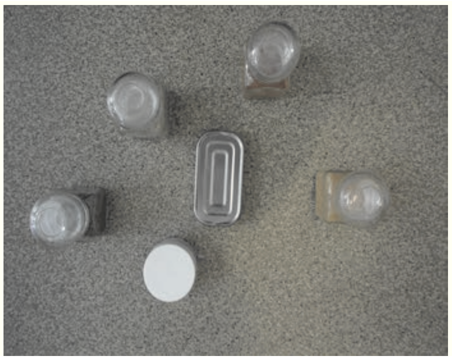
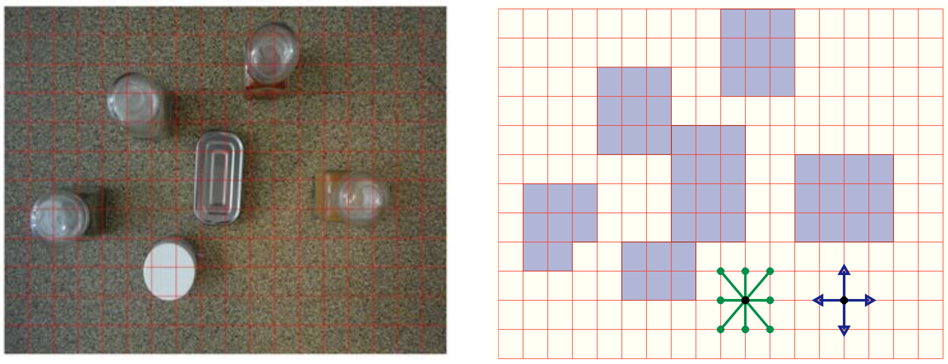
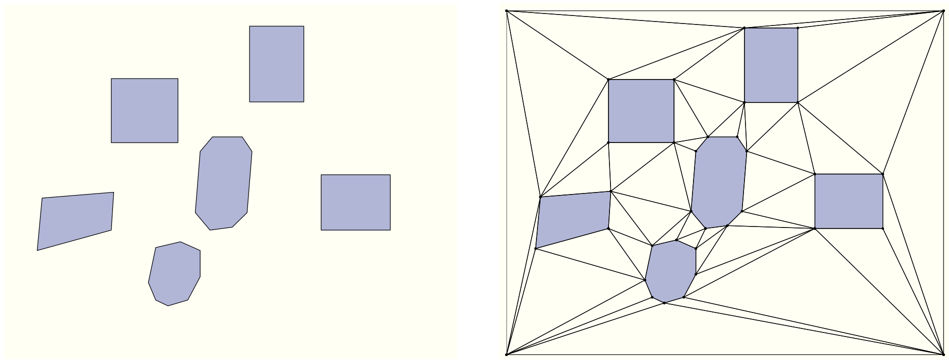


Figure 3.1: A photography of an environment



(a) The environment with grid lines (b) A grid representation of the environment

Figure 3.2: Transformation of workspace into a grid



(a) A polygonal map of the environment (b) A triangulation mesh of the environment

Figure 3.3: Transformation of workspace into a triangulation mesh

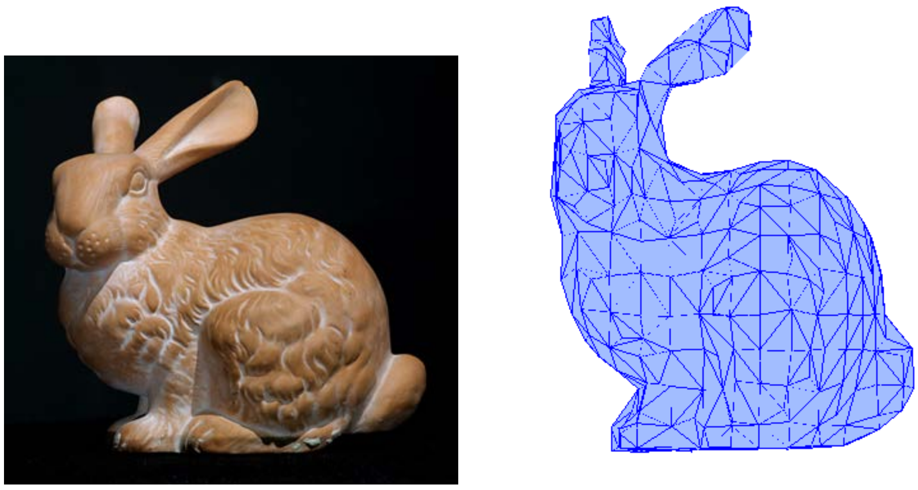

(a) A photography of a clay bunny (b) A polyhedral map of the bunny

Figure 3.4: Transformation of workspace into polyhedral map

A tetrahedral mesh is an equivalent for the triangulation mesh in $\mathbb{R}^3$ space. Both meshes share same attributes in comparison to grid. A tetrahedral mesh is created from a 3D triangulation mesh by process called tetrahedralization. This triangulation mesh consists of points end edges, which represent the constraining facets of the workspace. These facets are either the boundary facets or the obstacle facets, these facets constrain the free space of robot's motion. Even if the grid representation may be used in $\mathbb{R}^3$,it's space complexity is too high. Figure 3.4 shows a real life object represented as a 3D triangulation mesh. The image and the map in this Figure are taken from The Stanford 3D Scanning Repository [3].

3.1 Grid representation

After an environment is discretized into grid, the planar graph is formed of all vertices and the visible edges. The neighbors of current vertex are not saved, but computed by each expansion again.

The line-of-sight check is inspired by Bresenham algorithm. It is checked trough which cells passes the line segment formed by two points. The line-of-sight check does not draw the current pixel (cell in grid), but checks if the curresponding cell in grid is blocked or unblocked. If all cells along the line are unblocked, the path between points is marked as a visible path. The Bresenham line algorithm is an efficient algorithm to render a line with pixels. The long dimension is incremented for each pixel, and the fractional slope is accumulated [15].

3.2 The triangulation mesh for $\mathbb{R}^2$ space

The triangulation mesh represents $\mathbb{R}^2$ space. The triangulation is the division of a surface or a plane polygon into a set of triangles, relative to the restriction that each triangle side is entirely shared by two adjacent triangles. It was proved in 1925 that every surface has a triangulation, but it might require an infinite number of triangles and the proof is difficult [14].

The triangulation of a polygon means partition of into the non-overlapping triangles using the diagonals only. The triangulation maximizes the minimal angle of each triangle. The triangulation theorem says that every simple polygon admits the triangulation. Every triangulation of an n-gon has exactly $n - 2$ triangles [12].

The triangulation mesh is converted into a planar graph as an input for algorithm. Edges of created triangles stands as edges in the planar graph. They also represent the visibility between vertices. The information of the vertex neighbors is no longer computed by each expansion newly. The neighbors of each vertex are known on the basis in which triangles is given vertex located. Both of two points,which are associated with the expanded vertex to the current triangle, are considered as the vertex neighbors.

The main intention by developing the line-of-sight check for the triangulation mesh is the test of the path's visibility without usage of the obstacle knowledge. The algorithm is much faster when the check whether path crosses an obstacle is avoided. Thus it is used the knowledge of the adjacent triangles and it is assumed that none of the obstacles has a shape of triangle. While the evaluation, if the path crosses any edge of obstacle polygon, can be very slow on large maps and will use additional memory space. It is tested, whether the path lies in triangles, because it is set that the adjacent triangles are not obstacles so the path is passing trough unblocked space. The line-of-sight algorithm is build on this basis.

In this section,the used line-of-sight algorithm will be described. This algorithm tests whether the path between the *goal* vertex and the *parent* vertex is visible or not. The *actual* vertex is the closest neighbor of the *goal* vertex along the path between *goal* and *parent*. The loop is repeated unless the path crosses a new triangle edge(line **2, 9, 12, 18**) or there are no edges of the triangles to check (the path crosses an edge of an bstacle, is unblocked). The *goal* vertex and *actual* vertex are always the vertices of first triangle along the path between *goal* and *parent*. There are two possible triangles in space, that contains both *goal* and *actual* vertices. After these two triangles are found (line **4**) by determination of the *goal* and *actual* mutual neighbors. The two determined vertices are stored in the *neighbors* stack(line **4**). It is tested whether an edge of the triangle $\triangle(\textit{sucess actual child})$ intersects the line segment *goal and parent* (line **9, 12**). If there is an intersection of these two line segments, the *actual* and *child* vertices are moved to the next triangle (line **10, 13**). The inner loop stops and *neighbors* are reloaded. The hasLineLineIntersection() function tests whether the line segment *goal and success* (line **9**) or the line segment *parent and success* (line **12**) intersects the line segment *goal and parent.* Every *success* vertex that has already satisfied the hasLineLineIntersection function during the algorithm is removed from *neighbors* to prevent the endless loop. This is repeated until there are no triangle edges left to check, or the *start* vertex is found (line **7**).

Script 1: Line-of-Sight algorithm for triangulation mesh

```
   Data: Triangulation mesh
   Input: Goal,actual and parent vertex
   Result: Returns true if the path between two points is visible, false if it is blocked
1  child = goal;
2  repeat
3      continue = false;
4      neighbors = $actual.neighbors \cap child.neighbors$;
5      while neighbors.notEmpty() and continue == false do
6          success = neighbors.front();
7          if success = start then
8              return "path is visible";
9          if hasLineLineIntersection( $\overline{start\ goal}$ , $\overline{actual\ sucess}$ ) then
10             child = success;
11             continue = true;
12         else if hasLineLineIntersection( $\overline{Start\ Goal}$ , $\overline{child\ sucess}$ ) then
13             actual = success;
14             continue = true;
15         end
16         neighbors.delete(v2);
17     end
18 until continue ;
19 return "Path is not visible";
```

3.3 The tetrahedral mesh for $\mathbb{R}^3$ space

The tetrahedral mesh is an improved triangulation mesh to be used in the $\mathbb{R}^3$ space. The tetrahedralization divides the environment into tetrahedrons, the pyramid like geometrical objects. Even if the tetrahedralization is based on the triangulation, it does not share every advantages of the triangulation. The computation time of creating tetrahedral mesh is much longer. The vertices of the tetrahedral mesh must belong to given convex hull. The intersection of two tetrahedrons is either empty or a vertex or an edge or a facet. The number of the created tetrahedrons is no longer firmly determined as by triangulation. Every polyhedron with N vertices can be triangulated with $O(N^2)$ tetrahedral [12] [7].

The graph created from the tetrahedral mesh must be no longer planar. It contains all the points contained in tetrahedral mesh and the new created edges of polyhedrons stands as the visible paths between these points. The information of vertex neighbors is known on the basis in which tetrahedrons is given vertex located. All of the three points, that create with the current vertex a tetrahedron, are considered as vertex neighbors.

Line-of-sight check for tetrahedral mesh is developed with the same intention as for triangulation mesh This intention is to avoid the test of the path's visibility without usage of the obstacle knowledge. The essential precondition is, that none of obstacles has a shape of a tetrahedron and the tetrahedrons are considered as an unblocked space. The algorithm tests if the path crosses a facet of tetrahedron, and thus lies in a tetrahedron, and thus is visible and unblocked. The evaluation if the path crosses any obstacle facet is not used,because it can be very slow with rising number of the blocked facets and will also use an additional memory space.

In this section,the used line-of-sight algorithm will be described. This algorithm checks the visibility between the *goal* vertex and *parent* vertex in tetrahedral mesh. The *actual* vertex is a closest neighbor of *goal* vertex along the path between *goal* and *parent.* The loop is repeated unless the path crosses a new tetrahedron facet(line **2, 11, 24, 21**) or there are no facets of the tetrahedrons to check (the path crosses an facet of an obstacle, is unblocked). The *goal* vertex and the *actual* vertex are always the vertices of first tetrahedron along the path between *goal* and *parent.* All triangles that contain the two *goal* and *actual* vertices and the third *success* vertex (line **4**), represents one of the three facets (the base) of a tetrahedron. The *neighbors* queue and the *neighbors2* stack represents the all possible facet in tetrahedron with the base facet $\triangle(success\ actual\ child)$. It is tested which facet of the $tetrahedron(success\ actual\ child\ roam)$ is intersected by the line segment *goal and parent* (line **11, 14**). If there is and intersection of the facet and the line segment, *actual* and *child* vertices are moved to the next tetrahedron (line **12, 15**). The inner loop stops and *neighbors* and *neighbors* are reseted. The hasLineLineIntersection() function tests whether the facet *child, success, roam* (line **11**) or the facet *parent, success, roam* (line **12**) is intersected by the the line segment *goal and parent.* Every *roam* vertex that has already satisfied the hasLinePlaneIntersection() function during the algorithm is removed from the *neighbors* queue to prevent the endless loop. This is repeated until there are no tetrahedron facets left to check, or the *start* vertex is found.

Script 2: Line-of-Sight in 3D algorithm

```
Data: Tetrahedral mesh
Input: Goal,actual and parent vertex
Result: Returns true if the path between two points is visible, false if it is blocked
continue = false;
repeat
    neighbors = $actual.neighbors \cap child.neighbors$;
    while neighbors.notEmpty() and continue == false do
        success = neighbors.front();
        neighbors2 = $success.neighbors \cap neighbors$;
        while neighbors2.notEmpty() do
            roam = neighbors2.front();
            if roam = start then
                return "path is visible";
            if hasLinePlaneIntersection( $\overline{start\ goal}$ , $\triangle(actual\ success\ roam)$) then
                child = roam;
                continue = true;
            if hasLinePlaneIntersection( $\overline{start\ goal}$ , $\triangle(child\ success\ roam)$) then
                actual = roam;
                continue = true;
            neighbors2.delete(roam);
        end
        neighbors.delete(success);
    end
until continue ;
return "Path is not visible";
```

Chapter 4

Path-planning algorithms

In this chapter, four any-angle path-planning algorithms are described. All these algorithms are planning the path in graphs produced from various workspace representations. There exist several any-angle path-planning algorithms, that slightly differs in the computation time and the path length.

First, A* algorithm is presented as a base algorithm for following algorithms. A* finds long and non-realistic looking paths, because the path is constrained only to graph edges. The path of A* can be improved with post smoothing. A* PS (with post smoothing) adjusts the found path on end of A* algorithm to be smoother and more realistic looking. This adjustment shortens the former path created by A* at an increase in runtime [11].

Theta* is the name for family of any-angle path-planning algorithms based on same principles. All Theta* algorithms adjusts the path during the algorithm unlike A* PS that smooths the path after it is found. Following algorithms belong to Theta* family: Basic Theta*, Angle-Propagation Theta*, Lazy Theta*, Lazy Theta*-R, Lazy Theta*-P[4]. From Theta* family are presented Basic Theta*, as basic any-angle path-planning algorithm that adjusts the path during algorithm, and it's improvement Lazy Theta*, that runs significantly faster than Basic Theta*.

All algorithms mentioned above are designed to find path in static a environment, but the environment used in praxis rarely static. In most cases,the environment is dynamic, new obstacles arise or the some obstacles disappear. The re-computation of a single shot algorithms like Theta* or A* by each change of environment, mainly in large environments, can be very slow and inconvenient. Algorithm Incremental Phi*, which is presented, adjusts the path only in places, where the path is blocked instead of running the path-planning from scratch. Phi* is presented as algorithm which computes the initial path which is lately adjusted by the Incremental Phi* algorithm[9].

4.1 A*

A* is a basic algorithm widely used to solve informed path-finding problems in various workspaces. This algorithm is well described in almost every books or articles handling with path-finding. A* was first described by Peter Hart, Nils Nilsson and Bertram Raphael in year 1968. [6]

A* is correct, complete and optimal. Correctness means that the final found path from the *start* vertex to the *goal* vertex will be unblocked. Completeness means that a path from the *start* vertex to the *goal* vertex is found, if one exists. Optimality means that it is guaranteed to find the true shortest paths [11].

It is mentioned that A* is a well know algorithm, but in comparison of the other algorithms to A*, the main steps are described to be easier recognized the differences. Given the fact, that A* is very effective and fast, it also provides the ability to improve itself in many ways. On basis of this A*'s property, any-angle path-planning were developed from the A*. They differ from A* in the time complexity, the length of found path and the shape of path.

At the line **1-3** begins the initialization of the *start* vertex. It is the only vertex which *parent* is itself. *Open* is a priority queue, which sorts the unexpanded vertices inside ascending, using the f value as the key. Expanded vertices are stored in the *Closed* queue, to prevent A* from re-expanding them. By expansion, it is computed the valuation function f for every *child* vertex of the *actual* vertex (line **10**). *Childes* are the visible neighbors of *actual. Open* is resorted at every start of the while loop (line **on 5**). Function *front()* means that the first vertex in *Open* queue, with the lowest f value, will be saved as the *actual* vertex and is deleted from the *Open* queue. This vertex is supposed to be the one closest to the *goal* vertex. All expanded vertices are added to the *Open* queue (line **15**), and the *actual* vertex is set as their parent (line **19**) unless the *child* has already an another *parent* and the path trough it is shorter than the path trough *actual* (line **17**). After the goal vertex is found (line **6**), the path is re-construed recursively by the *parent* property. It is found the parent of the *goal* vertex. Then the parent of *goal's* parent etc, untill it reaches the *start* vertex and the path is complete.

All algorithms presented use the same valuation and the same heuristic function as A* :

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

If A* should find the optimal solution first, the next statement must be true:

$$\forall(n) : 0 \leq h(n) \leq h*(n)$$

This statement means that heuristic function is valid.
$h(n)$ is the straight-line distance from the n vertex to the *goal* vertex.
$g(n) = g(n-1) + c(n, n-1)$ where $c(n, n-1)$ is the value of the current edge E identified by the two vertices n and its *parent* $n-1$.

The time complexity of A* depends on the heuristic. The number of expanded vertices is polynomial in this case, because the search space is a graph, there is only one goal vertex and the heuristic function $h(x)$ meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

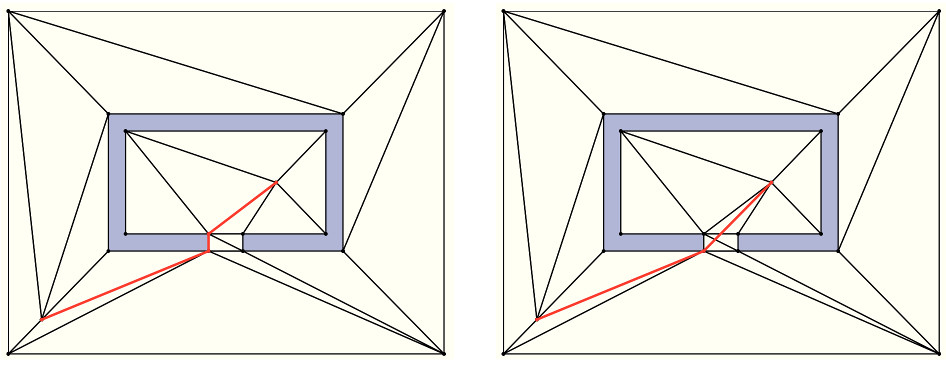
Script 3: A* algorithm

```
Data: Graph
Input: two vertices start and goal
start.g = 0;
start.parent = parent;
open.add(start);
while open.notEmpty() do
    actual = open.front();
    if actual.isGoal() then
        return "path found";
    end
    close.add(actual);
    neighbors = actual.expand();
    foreach child $\in$ neighbors do
        if child $\notin$ closed then
            if child $\notin$ open then
                child.g = $\infty$;
                child.parent = actual;
                open.add(child);
            end
            if $actual.g + length(actual, child) < child.g$ then
                child.g = actual.g + length(actual,child);
                child.f = child.g + length(child,goal);
            end
        end
    end
end
return "no path found";
```

$h^*(x)$ is the optimal heuristic, the true distance from vertex x to goal vertex. This means the error of $h(x)$ will not grow faster than the logarithm of h^* [8]. $h(x)$ function is used as the value of weighted edge in a graph, that means the straight-line Euclidean distance between two points. This next basic pseudo code describes the exact algorithm.

4.2 Basic Theta*

Basic Theta* is a version of A* for any-angle path planning that propagates information along graph edges without constraining the paths to graph edges. The motivation of using Basic Theta* is to find shorter and realistic looking paths in the grids and the meshes. Basic Theta* adjusts the path during the algorithm by allowing to be the *parent* any vertex not only the neighbor. To allow any vertex to be the *parent*, current vertex there must be



(a) *Constrained path* (b) *Not constrained path*

Figure 4.1: Constrained and not constrained path example

a visible path between them. For visibility verification are used the line-of-sight checks. This ability also extend the algorithm time complexity, that is longer than as by A*. As mentioned in the beginning, Basic Theta* differs from A* in the use of *Not constrained path* instead of *Constrained path* as A* does. *Not constrained path* is a path that is not only constrained to the edges in graph but is placed in the continuous free space. The difference between the use of *Not constrained path* (by Basic Theta*) and the use of *Constrained path* (by A*) can be seen in the Figure 4.1.

Basic Theta* is correct and complete but not optimal. It is not optimal, because the parent p of a vertex v has to be either a visible neighbor of the vertex v so $p \in v.neighbors$ or the parent of a visible neighbor p Such condition is not always the case for true shortest paths[11].

The function line-of-sight is the only difference in the pseudo code and can be seen on (line **19-24**). At this point, the algorithm decides which path will it use if the *Not constrained path* or *Constrained path*.
Constrained path is only constrained to the initial graph edges, this means that the *parent* of a vertex can be only it's neighbor.
Not constrained path means that any vertex can be the *parent*, so the final path will not lead only along the edges, but freely trough the unblocked space. This decision is always defined on the basis, if there is a visible path between child and child's grandparent (*actual's parent*), this means whether the line-of-sight check is true, or not. If the line-of-sight check is true, this *Not constrained path* path is used and the expediency of *Constrained path* is not checked (to this refers the if else statement). The possible *Not constrained path* is shorter than the *Constrained path*, due to the *triangleinequality*. In case there is a *Constrained path* from the *parent* to the *child* that leads trough the *actual*m that means the path connect the vertices via $parent \Rightarrow actual \Rightarrow child$. And there exist a *Not constrained path* from the *parent* to the *child*, it is chosen the *Not constrained path* and the parent of *child* is *parent* instead of its neighbor *actual*. This new *Not constrained path*, connecting the vertices via$Parent \Rightarrow Child$, must meet the requirement. $parent \Rightarrow child$ path must be

shorter than path via *parent* ⇒ *actual* ⇒ *child*. This requirement is satisfied due to the $triangleinequality$. The triangle $\triangle parent, actual, child$ is constructed and the following statement about triangle length of edges is true:

$$\overline{parent, actual} + \overline{actual, child} \leq \overline{parent, child}$$

If it is assumed that the child node is added to open list at the first time, so there is no former path to it, which can be shorter. The weighted function $f(child)$ of *Constrained path* will be:

$$f(child) = \overline{child, goal} + actual.g + \overline{child, actual}$$

$$actual.g = parent.g + \overline{actual, parent}$$

$$f(child) = \overline{child, actual} + parent.g + \overline{actual, parent} + \overline{child, actual}$$

The weighted function $f(child)$ of *Not constrained path* will be:

$$f(child) = \overline{child, goal} + parent.g + \overline{child, parent}$$

After comparison of both these functions $f(child)$ we can see that the that the *Not constrained path* is in this case shorter than the *Constrained path* mentioned before.

4.3 Lazy Theta*

Lazy Theta* is a version of Basic Theta*. Lazy Theta* is developed to achieve less number of line-of-sight checks and thus shorter the computation time of the algorithm. The reduction of line-of-sight checks are achieved by delaying them, until they are completely necessary. This delay causes some mistakes in order of vertex expansion, but the computation time is shorter than by Basic Theta* and the path is complete and correct. The nature of line-of-sight check remains identical as in Basic Theta*. This means that it still uses the triangle inequality as main property.

The expanded vertices *childes* are first all considered to have a visible path to the *actual* also the f value is updated and the line-of-sight check is executed first when the *child* is chosen as *actual*. The benefit of this procedure lies in the fact that only at such vertices, that are truly close to the goal vertex, the line-of-sight check is proceeded. On the other hand, a little error occurs during the computation of the f value on path that might not exist and thus the order in *Open* queue might be wrong. This mistake causes the expansion of unwanted vertices, that do not have generally the lowest f value of all in *Open* queue. But it saves the computation time of line-of-sight check per vertices, that are farther from goal vertex, in other words they are in the end of *Open* queue.

Script 4: Basic Theta* algorithm

```
Data: Graph
Input: two vertices start and goal
start.g = 0;
start.parent = parent;
open.add(start);
while open.notEmpty() do
    actual = open.front();
    parent = actual.parent;
    if actual.isGoal() then
        return "path found";
    end
    close.add(actual);
    neigbors = actual.expand();
    foreach child ∈ neighbors do
        if child ∉ closed then
            if child ∉ open then
                child.g = ∞;
                open.add(child);
            end
            parent = actual.parent;
            if line-of-sight(parent,child) then
                //Not constrained path
                if parent.g + length(parent, child) < child.g then
                    child.g = parent.g + length(parent,child);
                    child.parent = parent;
                    child.f = child.g + length(child,goal);
                end
            else if actual.g + length(actual, child) < child.g then
                //Constrained Path
                child.g = actual.g + length(actual,child);
                child.parent = actual;
                child.f = child.g + length(child,goal);
            end
        end
    end
end
return "no path found";
```

Script 5: Lazy Theta* algorithm

```
Data: Graph
Input: two vertices start and goal
start.g = 0;
start.parent = parent;
open.add(start);
while open.notEmpty() do
    actual = open.front();
    parent = actual.parent;
    if Not-line-of-sight(parent,actual) then
        //Constrained Path
        localparent = actual.localparent;
        actual.g = localparent.g + length(localparent,actual);
        actual.parent = localparent;
    end
    if actual.isGoal() then
        return "path found";
    end
    close.add(actual);
    neigbors = actual.expand();
    foreach child ∈ neighbors do
        if child ∉ closed then
            if child ∉ open then
                child.g = ∞;
                open.add(child);
            end
            parent = actual.parent;
            localparent = child.localparent;
            //Not constrained path
            if parent.g + length(parent, child) < child.g then
                if length(actual,child) <= length(localparent,child) then
                    child.localparent = actual;
                end
                child.g = parent.g + length(parent,child);
                child.parent = parent;
                child.f = child.g + length(child,goal);
            end
        end
    end
end
return "no path found";
```

4.4 Phi*

Phi* is a version of Basic Theta* however it is strictly used only at grids. This algorithm was developed to describe the initial state of *dynamic workspace* sufficiently. When the single-shot algorithms like Theta* can be very slow by recomputing the path from scratch by each newly blocked cell, Incremental Phi* algorithm provides the new re-construed path much faster. To describe Incremental Phi* must be described Phi* first, because Phi* computes the initial shortest path in workspace without new obstacles. This path and properties of Phi* are used as the input for Incremental Phi*, that adjust the Phi* path. Incremental Phi* will be described later.

The most suitable representation of a dynamic environment is grid,because new obstacles are represented as newly blocked cells in grid. The blockage status of some cells in grid can be changed very fast, but the triangulation mesh must be triangulated for every new change of the environment. Thus triangulation mesh represents a very slow representation of a dynamic environment.

Phi* is complete and correct but not optimal.

Phi* gathers the local parents of vertices to be made incremental.[11] The main difference is, that Phi* uses angle ranges to constrain the direction. This property is also used to find out the best local parent along the path. The best local parents are not identified as in Lazy Theta* algorithm. Local parents are stored for further use and are for additional change of path, because on every new child vertex is executed the line-of-sight check. This means, that by every expanded vertex determine the algorithm properly whether use *Not constrained path* or *Constrained path*. Phi* also maintains two additional values for each vertex and that is the lb - lower bound and ub - upper bound angle.

The first difference can be seen on (line **19**) where even more conditions are added to choose the *Not constrained path*. There must a visible path between *parent* and *child* vertex. Additionally the angle $\angle(actual, parent, child) \in < actual.lb, actual.ub >$ are computed. This represents the condition, that the *child* must lie inside the constrained area, which is defined by value of angles stored in *actual* lower and upper bound. Function angle(actual,parent,child) is defined as the smaller angle $\angle(actual, parent, child) \in [-180°, 180°)$ between the ray from *parent* through *actual* and the ray from *parent* through *child*. The angle is positive if and only if the *parent* - *actual* ray is counterclockwise of the *parent* - *child* ray. $\angle(child, parent, axis) \neq k*45°($ where k is an integer), stands there to avoid unwanted execution of *Not constrained path* when the *Constrained path* is sufficient. Function angle(child,parent,axis) is the smaller angle formed by the ray *parent* - *actual* and the vertical line through *parent*. *Not constrained path* is no longer than *Constrained path* due to the triangle inequality, but they are equally long if $\angle(child, parent, axis) = k*45°(kisaninteger)$. In this case it is better to define the values of *child* vertex via *Constrained path* to save computation time. The bounds of new created constraining angle are updated (line **25-26**). *Crossbar-neighbors* are 4 neighbors on the grid lying on the crossbar centered on current vertex. *Crossbar-neighbors* of a vertex also satisfies the condition that the distance between them and the vertex is the lowest of all neighbors, they are 4 for 2D grid and 6 for 3D grid. At (line **25**), the angle $\angle(child, parent, crossbarneighbor)$ of all crossbar-neighbors is minimized. At (line **26**), this angle is maximized. This sets the next values of lower and upper bound of *child*. At (lines **34-36**) the local parent of *child* is set and *child* lower and upper bound is initialized to initial values.

4.5 Incremental Phi*

Incremental Phi* as a version of Phi* is usable only at grids. Incremental Phi* provides faster results by recomputing paths than repeated single shot Phi* or Theta* algorithms. As mentioned before the initial path is computed by Phi* and its recomputed for each newly blocked cell in *dynamic workspace* by Incremental Phi*.

Same as Phi* can be Incremental Phi* only used on grids.

Incremental Phi* first detect whether the newly occurred obstacle lies on a path. It detects whether corner C of newly blocked cell is in the *Closed* or the *Open* list. Incremental Phi* recomputes the g value and reset the parent of each visible neighbor of all vertices that have as their local parent C and lies in *Closed* list.

The efficiency of using incremental Phi* instead of single shot algorithms depends on the number of newly blocked cells C. By small amount of newly blocked cells C is Incremental Phi* algorithm much faster than repeated singe shot algorithms.

This code is only extension for Phi* algorithm where ComputePath function at (line **1**), is the whole algorithm Phi*. First the initial shortest path is computed and after new cells becomes blocked the path is recomputed, with former knowledge. *AlongPath* and *ChangedPath* are queues of vertices. ResetVertex() (line **9**) resets all information about current vertex. After the reset, the new best parent of current vertex is found. This is steps are executed to found new parent with visible path. Set *Eightneighbors* at (line **17**) is a set of all 8 neighbors of a vertex on grid, regardless to their's visibility. Function recomputePath() is shown in the Phi* algorithm at (lines **19-39**), it rebuilds the former shortest path if it becomes blocked by new obstacle.

Script 6: Phi* algorithm

```
Data: Graph
Input: two vertices start and goal
start.g = 0;
start.parent = parent;
open.add(start);
while open.notEmpty() do
    actual = open.front();
    parent = actual.parent;
    if actual.isGoal() then
        return "path found";
    end
    close.add(actual);
    neighbors = actual.expand();
    foreach child $\in$ neighbors do
        if child $\notin$ closed then
            if child $\notin$ open then
                child.g = $\infty$;
                open.add(child);
            end
            parent = actual.parent;
            //Recompute Path function,also used in incremental Phi*
            if line-of-sight(parent,child) and angle(actual,parent,child)
            $\in$ [actual.lb, actual.ub] and angle(child,parent,axis) $\neq k * 45^\circ$ $k \in$ Integer
            then
                //Not constrained path
                if $parent.g + length(parent, child) < child.g$ then
                    child.g = parent.g + length(parent,child);
                    child.parent = parent;
                    child.f = child.g + length(child,goal);
                    child.localparent = actual;
                    l = min(angle(child,parent,crossbarneighbors));
                    u = max(angle(child,parent,crossbarneighbors));
                    child.lb = max( l, actual.lb - angle(actual,parent,child) );
                    child.ub = min( u, actual.ub - angle(actual,parent,child) );
                end
            else if $actual.g + length(actual, child) < child.g$ then
                //Constrained Path
                child.g = actual.g + length(actual,child);
                child.parent = actual;
                child.f = child.g + length(child,goal);
                child.localparent = actual;
                child.lb = $-45^\circ$;
                child.up = $-45^\circ$;
            end
        end
    end
end
return "no path found";
```

Script 7: Incremental Phi* algorithm

```
Data: Graph
Input: two vertices start and goal
ComputePath;
if New cells blocked then
    foreach corner C of newly blocked cell do
        if ($C \in$ Closed or $C \in$ Open) and $C \neq$ start then
            AlongPath.add($C$);
            while AlongPath.notEmpty() do
                first = AlongPath.front();
                ChangedPath.add(first);
                Open.remove(fist);
                Open.remove(fist);
                first.g = $\infty$;
                first.lb = $-\infty$;
                first.ub = $\infty$;
                first.localparent = null;
                first.parent= null;
                foreach vertex $v \in$ first.eightneighbors do
                    if v.localparent = first then
                        AlongPath.add(v);
                    end
                end
                while ChangedPath.notEmpty() do
                    second = ChangedPath.front();
                    foreach vertex $v \in$ second.neighbors do
                        if $v \in$ Closed then
                            recomputePath();
                        end
                    end
                end
            end
        end
    end
end
return "no path found";
```

Chapter 5

Experiments

In this chapter, properties of the presented algorithms are studied. The properties are the computation time and the length of a found path by the particular algorithms. The Results are obtained from three workspaces (grids, triangulation meshes, tetrahedral meshes). All computations were performed on Notebook MSI GX610P, processor AMD® Mobile Turion64 X2 2.0GHz, operation system Linux ubuntu 2.6.38-10-generic 64bit.

5.1 Experiments on grid workspace in $\mathbb{R}^2$ space environment

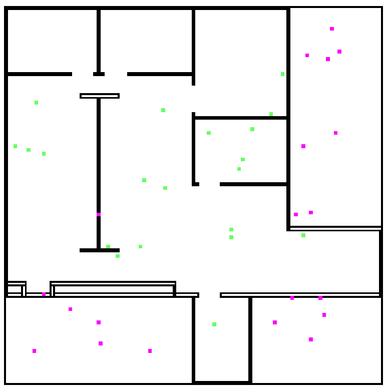
In this section, experimental results from path-planning on a grid workspace in $\mathbb{R}^2$ space are shown and discussed.

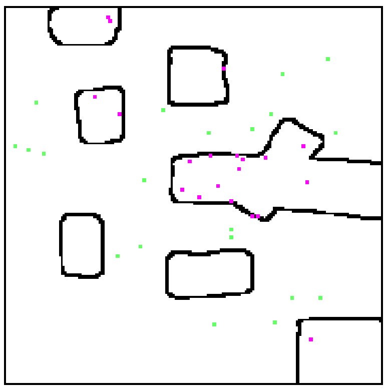
The maps used in this experiment are from player stage [2]. Next Figure 5.1 shows the used maps with the reachable (green) and the non-reachable (purple) vertices, that are used in experiments.

Following experiment consist of a complex algorithm test. Twenty reachable vertices are chosen from each map. The algorithms has to find the path between a pair of two different vertices. For each pair of the vertices is the algorithm ran ten times to achieve a high accuracy. Totally, each algorithm is ran 3800 times on each map. The average and relative path length and computation time are displayed in the Tables 5.2, 5.1. The relative values are relative to the lengths and the times of A* algorithm.

All path results for each pair of two different reachable vetices are compared, and it is proven that any-angle path-planning algorithms never find longer path than A*. Basic Theta* and Lazy Theta* find approx. 7% shorter paths than A* in avarage, which is similar to results of Sven Koenig's work [11].

The decrease of the path length causes the increase of the runtime shown in 5.2. It can be seen that the computation time of Lazy Theta* is only one and a half times higher than time of A* when the time of Basic Theta* is two times higher. This big difference of computation time between Basic and Lazy Theta* is caused by the number of line-of-sight checks that are executed during the program. Phi* does not have better results


(a) autolab map


(b) cave map

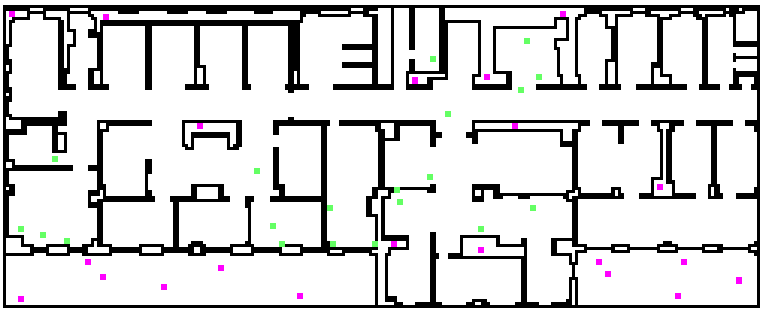

(c) hospital map

Figure 5.1: Path of Basic Theta*

in path length nor in computation time. As was mentioned Phi* is a base algorithm for Incremental Phi*, described in next section 5.4. Phi* is supposed to compute the initial path and maintain the specific properties for vertices for Incremental Phi*.

In the next experiment is compared and discussed the computation time of a non-existing path (Table 5.3). Twenty reachable vertices were chosen and twenty not reachable. The *Start* vertex is always one of the reachable vertex this means it lies in a free space and the *goal* vertex lies in an obstacle. This experiment compares the time complexity of the line-of-sight check. The number of expanded vertices by the former experiments (with existing path) differs. Algorithm in this case has to expand each vertex in a free space which causes the number of expanded vertices be same by all algorithms. The difference in the runtime between Lazy and Basic That* is so high, because Basic Theta* executes more line-of-sight checks than Lazy Theta*. Basic Theta* performs one or more line-of-sight checks per vertex while Lazy Theta perform always only one. From results of Tables 5.2 and 5.3, it can be seen that Lazy Theta* expands more vertices in free space than necessary because Lazy Theta* updates the g value incorrectly due to optimistic assumption of a non-blocked path. This causes that some vertices are expanded by Lazy Theta*, while Basic Theta* does not expand them.

algorithm	A*		Basic Theta*		Lazy Theta*		Phi*	
map	l_A	%	l_{BT}	%	l_{LT}	%	l_P	%
autolab	97.3	1	91.1	93.6	90.9	93.5	95.6	98.3
cave	109.8	1	103.9	94.6	103.7	94.5	106.4	97.0
hospital	113.8	1	105.8	92.0	105.7	92.9	111.4	97.9

Table 5.1: Path length

algorithm	A*		Basic Theta*		Lazy Theta*		Phi*	
map	t_A	%	t_{BT}	%	t_{LT}	%	t_P	%
autolab	16.8	1	33.4	198.5	26.1	155.1	34.7	206.1
cave	33.1	1	62.1	187.3	49.5	149.3	64.3	194.1
hospital	46.9	1	88.2	188.0	69.6	148.4	94.1	200.5

Table 5.2: Computation time (milliseconds), existing path

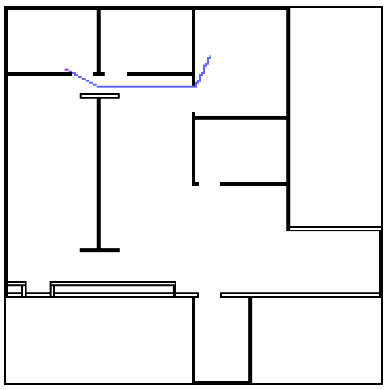
algorithm	A*		Basic Theta*		Lazy Theta*		Phi*	
map	t_A	%	t_{BT}	%	t_{LT}	%	t_P	%
autolab	92.7	1	411.6	443.9	191.0	206.0	258.4	278.7
hospital	209.1	1	1025.8	490.5	459.2	219.6	570.2	272.6
cave	275.5	1	1200.2	435.6	566.8	205.8	743.5	269.8

Table 5.3: Computation time (milliseconds),non existing path

The next experiments shows the unusual cases of any-angle algorithms behavior. These cases are obtained from set of experiments with random position of the *start* vertex and the *goal* vertex.

The first set of Figures 5.2, 5.3 and 5.4 refers to optimality of Theta* algorithms. In these Figures can be seen that the paths differ, so the any of Theta* algorithm is not always guaranteed to find the shortest path. Lazy Theta* can sometime find shorter paths, because it assumes that the path is not blocked, thus the path leads straight from the *start* vertex to the *goal* vertex and the path is only adjusted along the obstacles. In some cases the path found by Basic Theta* is shorter than the path found by Lazy Theta*, because Basic Theta* updates newly expanded vertices properly.

Figures 5.5 displays cases when the * algorithm is slower than the Lazy Theta* algorithm, despite Lazy Theta* is more complex . This property of Lazy Theta* occurs always when the path is not blocked by any obstacle, because A* expands far more vertices. Lazy Theta* expands only vertices along the straight path from the *start* vertex to the *goal* vertex. Also the computation time of Basic Theta* is sometime lower in this cases, but not as short as time of Lazy Theta* algorithm because Basic Theta* performs more line-of-sight checks than Lazy Theta* thus slows the computation time.


(a) Path of Basic Theta*

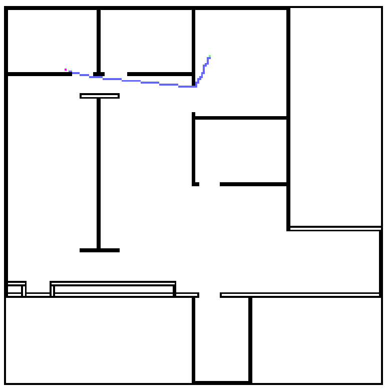
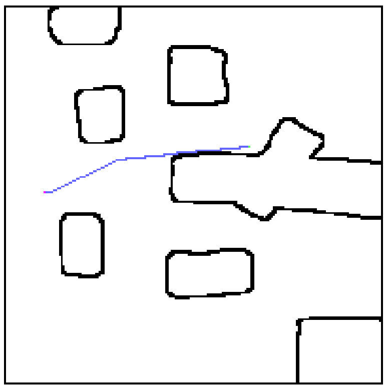

(b) Path of Lazy Theta*

Figure 5.2: Autolab map


(a) Path of Basic Theta*

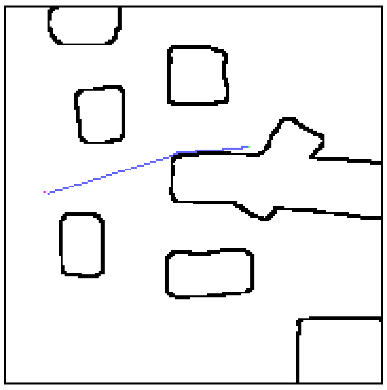
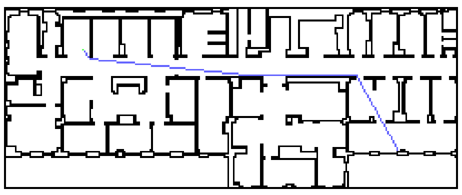

(b) Path of Lazy Theta*

Figure 5.3: Cave map


(a) Path of Basic Theta*

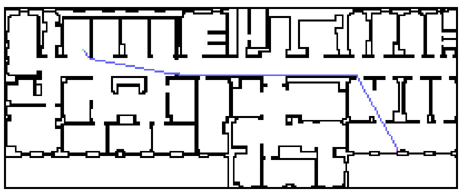
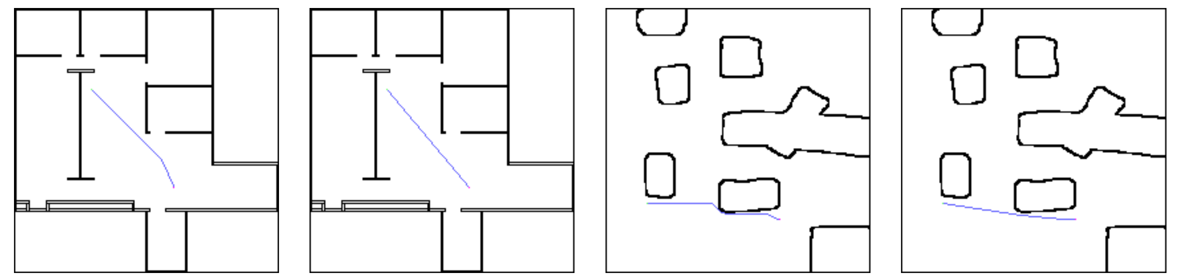

(b) Path of Lazy Theta*

Figure 5.4: Hospital map



(a) A* path in autolab map (b) Lazy Theta* path in autolab map (c) A* path in cave map (d) Lazy Theta* path in cave map

Figure 5.5: Comparision of A* a Lazy Theta* paths, when Lazy Theta* has significantly shorter runtime than A*

5.2 Experiments on triangulation mesh workspace in $\mathbb{R}^2$ space environment

The objective of this experiments is to examine the results of any-angle path-planning algorithms and compare them with the grid results.

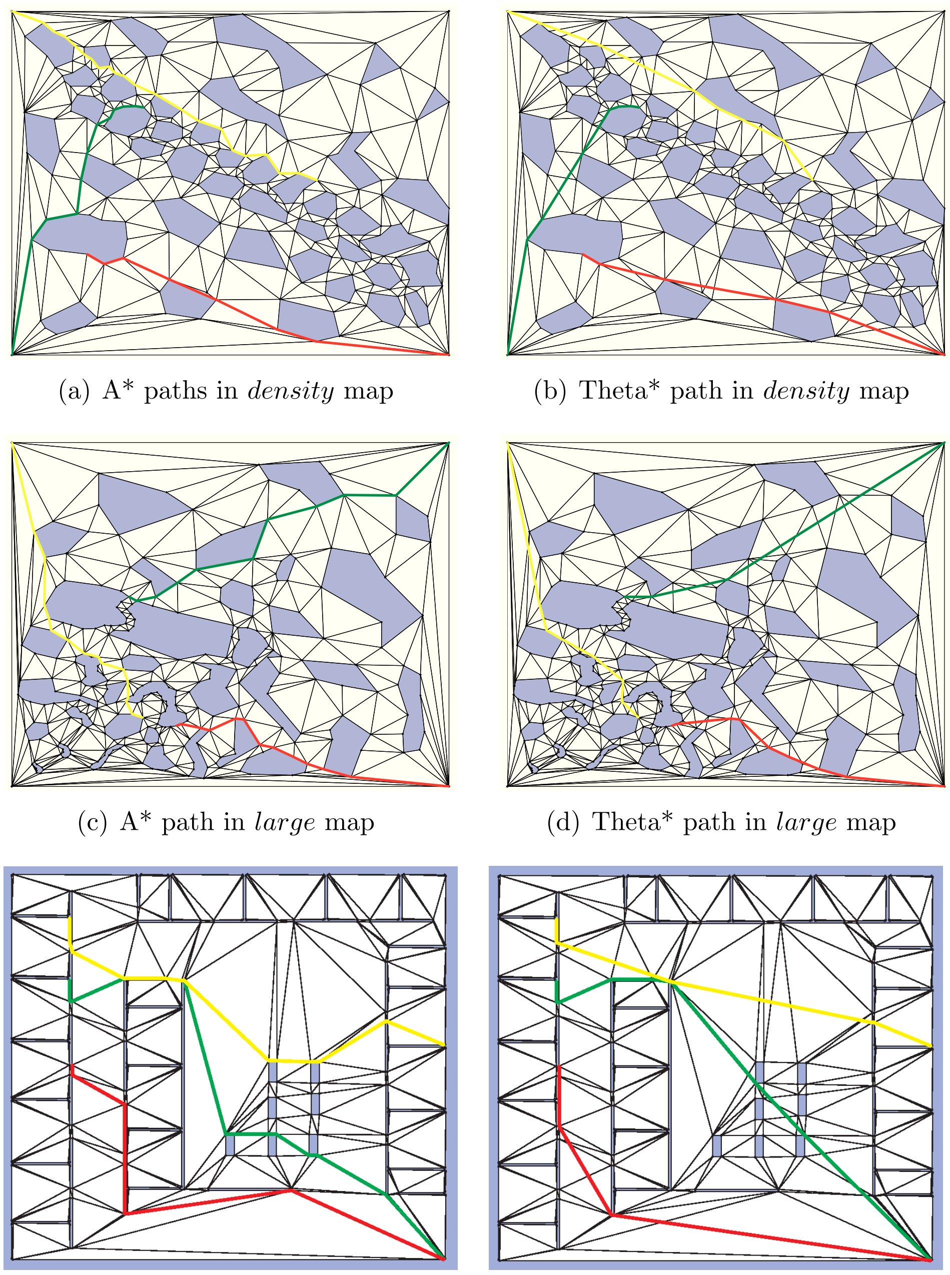
The maps used in this experiment are from Mr. Kalisiak and from Intelligent and Mobile Robotics Group [1]. Next Figure 5.6 shows the used triangulated maps (large, jari, density) with example paths of A* and Theta*. The maps differs in number of vertices and in the shape and the number of the obstacles. As expected Theta* generates shorter and smoother path than A*. In Figure 5.6 can be seen generated paths from three unique pairs of the*start and goal* vertices and the paths generated by A* and Theta* (red, green, yellow).

Experiments are done at all three maps. The *start* vertex is fixed in the right bottom corner of the map and the *goal* vertex is randomly chosen from one of the map points. On each map are planned paths with fifty *goal* vertices. Every planned path is repeated ten times to achieve accurate computation time.

Table 5.6 shows the average and the relative path length. As it can be seen, any-angle algorithms find in the triangulation mesh paths only approx 2.5% shorter than A*. This low value of the any-angle algorithms main property to find shorter paths than A* is caused by the workspace representation. The shortest path is located near the triangle edges. In despite of the grid, where the vertices are symmetrically placed in the space and many vertices can be placed in free space. Vertices in triangulation mesh are placed only on edges of obstacles.

Another difference from the grid workspace representation can be seen between paths of Basic and Lazy Theta*. When on the grid Lazy Theta* founds slightly shorter paths than Basic Theta*, on triangulation mesh it is otherwise, because Basic Theta* expands the values of vertices more properly.

Table 5.7 displays the results of any-angle algorithm's runtimes in triangulation mesh. The difference between relative computation times on grid and triangulation mesh is caused


(a) A* paths in *density* map (b) Theta* path in *density* map

(c) A* path in *large* map (d) Theta* path in *large* map

(e) A* path in *jari* map (f) Theta* path in *jari* map

Figure 5.6: Difference in path between any-angle algorithm and A* on var-density map

by time complexity of line-of-sight checks.

For the triangulation is used external C library Triangle developed at University of California at Barkley [10].

algorithm	A*		Basic Theta*		Lazy Theta*	
map	l_A	%	l_{BT}	%	l_{LT}	%
large	3276.6	1	3230.1	98.5	3239.8	98.8
density	1591.0	1	1552.2	97.5	1553.0	97.6
jari	2125.5	1	2037.9	95.9	2042.8	96.1

Table 5.4: Path length

algorithm	A*		Basic Theta*		Lazy Theta*	
map	t_A	%	t_{BT}	%	t_{LT}	%
large	410	1	1440	3.5	720	1.7
density	910	1	3030	3.3	1450	1.6
jari	1190	1	4090	3.4	2000	1.7

Table 5.5: Computation time (milliseconds)

5.3 Experiments in $\mathbb{R}^3$ space environment

In this section, experimental results from path-planning in $\mathbb{R}^3$ space are shown. In the beginning of this section, example paths are shown to prove that any-angle algorithms can work properly with all their properties also in tetrahedral mesh.

Following figures are created by plotting the object facets in matlab and then plotting the path (red line) with it's vertices (red dots) into same Figure. The path is planned from the chosen *start* vertex and the chosen *goal* vertex.

In Figure 5.7 an example environment is displayed, this is an simple 3D image of gear's part. Figures 5.7, 5.8, 5.9 do not represent an complex environment, but in these Figures are shown the main properties of A* and Theta* algorithms. All three figures are displayed from two views to provide better projection of an image. An example path of A* is shown in Figure 5.8. The same *goal* and *start* vertices are used for Theta* algorithm. In Figure 5.9 can be seen the key difference between A* and Theta* algorithms. Theta* algorithms plan the path trough the unblocked space unlike A*, which path is only constrained to graph edges.

The next set of Figures 5.10 and 5.11 provides a better comparison of the different path construction between A* and Theta* algorithms. This *bunny* environment can be considered as a complex environment. The 3D tetrahedral mesh of *bunny* environment comes from The Stanford 3D Scanning Repository [3]. On these Figures can be better seen the usage of *Not constrained path* by Theta* algorithms where it significantly shortens and smooths the path. In Figure 5.10, it may seem that A* do not use the optimal path trough object like it is in Figure 5.8 and plans the path along the triangulated hull, this happens because the created tetrahedrons inside bunny are not symmetric like 3D grid and thus the map trough the object is not optimal. This property of tetrahedral mesh can appear as very inconvenient but it has any negative effect on any-angle algorithms.

Results shown in Table 5.6 are obtained from randomly chosen *goal* vertices with fixed *start* vertex. Totally there were chosen fifty *goal* points for every map. Input map is is the *bunny* map obtained form The Stanford 3D Scanning Repository [3]. This map is compressed with different resolutions, *res*4 is the biggest compression.

Because tetrahedral mesh has similar properties as triangulation mesh except the dimension, results are corresponding to results from triangulation mesh and differs in same way results from triangulation mesh do. An exception is by using Basic Theta* when the computation time reaches double relative value than in triangulation mesh 5.7. Because the values of computation were so different the relative time were computed separately for each map and than averaged into one value displayed in Table 5.7. The relative runtimes has increased compared to relative runtimes from triangulation mesh, due to higher time complexity of line-of-sight checks. The relative value of path lengths (Table 5.6) rises from 2.5% (triangulation mesh) to circa 5% for any-angle algorithms compared to A*. This decrease of relative path length in comparison with triangulation mesh is because in tetrahedral mesh is much more edges in the unblocked space and are not only along the blocked facets.

algorithm	A*		Basic Theta*		Lazy Theta*	
map	l_A	%	l_{BT}	%	l_{LT}	%
bunny-res2	0.097	1	0.091	93.8	0.092	94.8
bunny-res3	0.109	1	0.103	94.5	0.105	96.3
bunny-res4	0.144	1	0.137	95.1	0.138	95.8

Table 5.6: Path length

algorithm	A*		Basic Theta*		Lazy Theta*	
map	t_A	%	t_{BT}	%	t_{LT}	%
bunny-res2	9178.4	1	76456	833.0	14033.2	152.9
bunny-res3	551.6	1	4945	896.5	1027.8	186.3
bunny-res4	596.4	1	5370.2	900.4	1118.4	187.5

Table 5.7: Computation time (milliseconds)

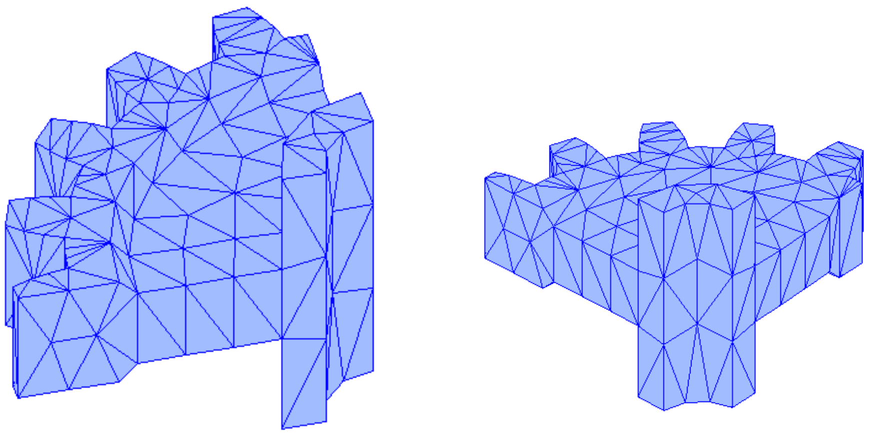


Figure 5.7: *Gear* environment

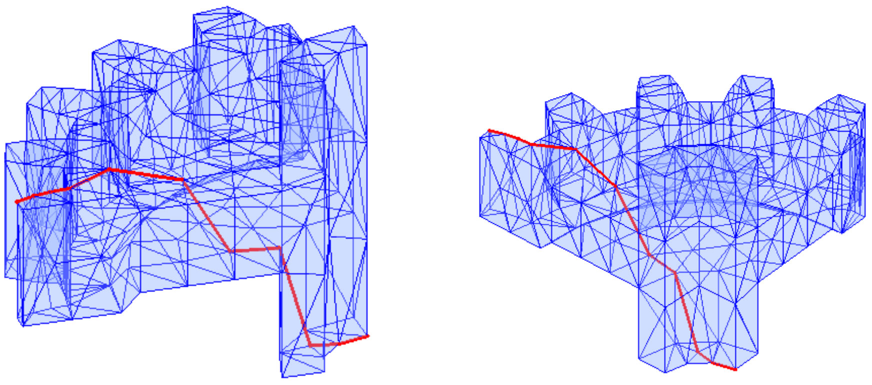


Figure 5.8: Example path of A* in *Gear* environment

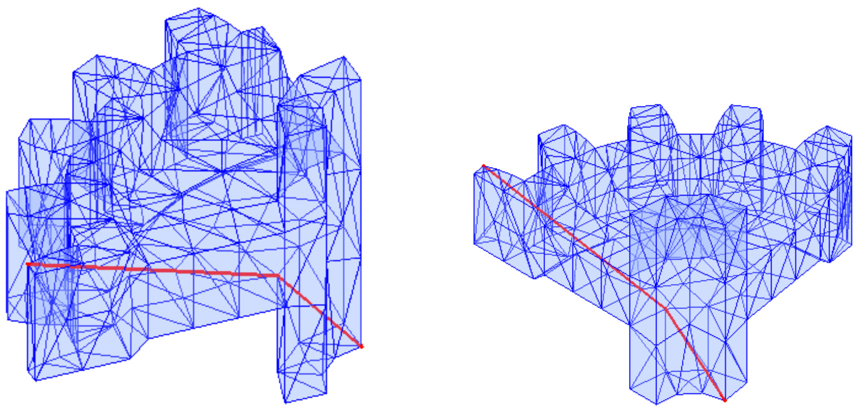
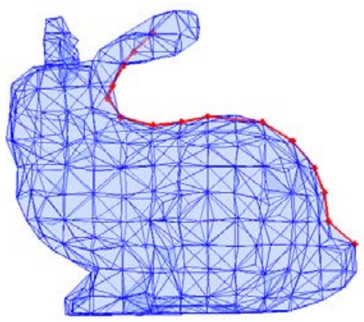
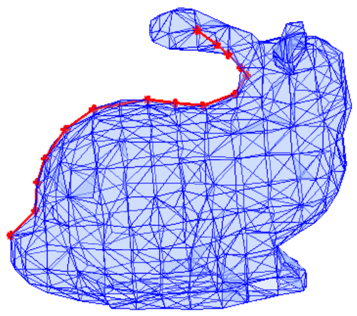


Figure 5.9: Example path of Theta* in *Gear* environment


(a) Front view


(b) Back view

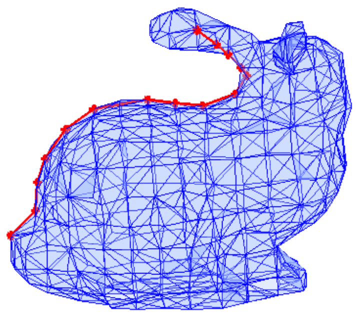
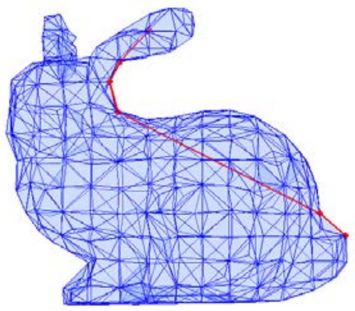
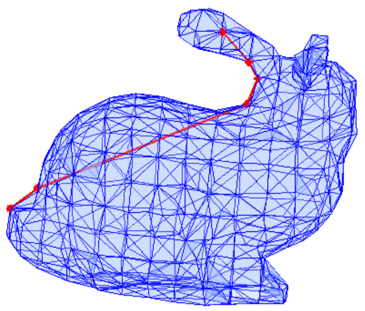

(c) Rear view

Figure 5.10: Example path of A* in *Bunny* environment


(a) Front view


(b) Back view

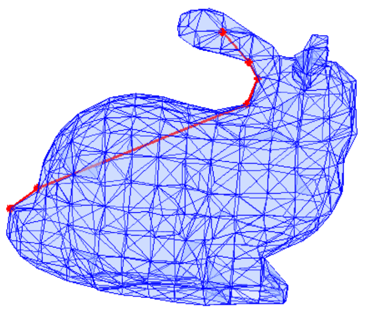

(c) Rear view

Figure 5.11: Example path of Theta* in *Bunny* environment

5.4 Experiments in $\mathbb{R}^2$ space dynamic environment

The experiments in the dynamic environment has the objective to test whether the Incremental Phi* algorithm, especially designed for dynamic environment, achieves shorter runtime than single shot algorithm Basic Theta*. As mentioned Basic Theta* recomputes the path for every change (state) of environment while Incremental Phi* is only adjusting the path if new obstacle of environment appears along the assumed path. Incremental phi* is designed to handle new appearing obstacles in environment. When an obstacle disappears (moves away or change position), the path must be newly recomputed via single shot algorithm because Incremental Phi* cannot is not designed for such cases.

Experiments are done in "hospital" map , where in the middle of path four new obstacles (purple squares) are located into the map sequential. Figures 5.13 and 5.14 shows example paths that are adjusted (Incremental Phi* path) or newly recomputed (Basic Theta* paths). Basic Theta* was computed on this map without new obstacle, and then again recomputed with on for every new obstacle occured in the environment.

In Figure 5.15 is shown a graph that contains the average computation times for sequential adding of obstacles. One of the objectives was to compare the results with results from work of Sven Koenig [9]. In this case it is irrelevant to compare the relative time results because the nature of experiments in work of Sven Koenig is different from this. But as it can be seen Incremental Phi* has significantly shorter runtime as single algorithm Basic Theta*.

Incremental Phi* is not optimal and must not always find a path around new obstacle, if the obstacle is big enough 5.12. This happens because of Incremental Phi*'s nature, when the algorithm needs the property of local parents for each vertex, the path will be replaned trough, and this property is maintained with the expansion of current vertex. Because Incremental Phi* is also a version of A* and inherits A*'s properties, when A* expands only verices along the path. It may happen that if the obstacle is too big and vertices around the obstacle are not expanded, thus the local parent property is not maintained, the path must be not found even if one exists. On the other hand it is nepravdepodobne that a new obstacle will have that big size.

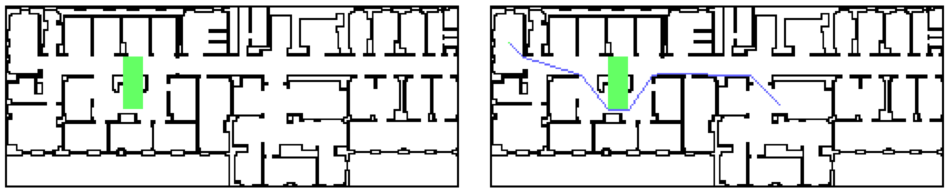

(a) Path of Incremental Phi* (no path found) (b) Path of Basic Theta*

Figure 5.12: *Hospital* environment with a new obstacle of a big size

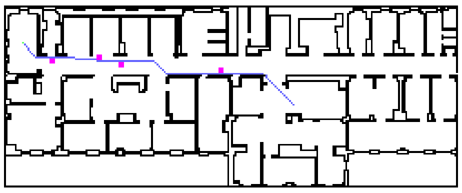


Figure 5.13: Example path of Incremental Phi* in *hospital* environment with all four new obstacles

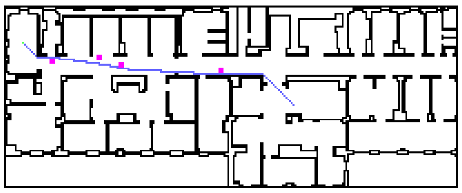


Figure 5.14: Example path of Basic Theta* in *hospital* environment with all four new obstacles

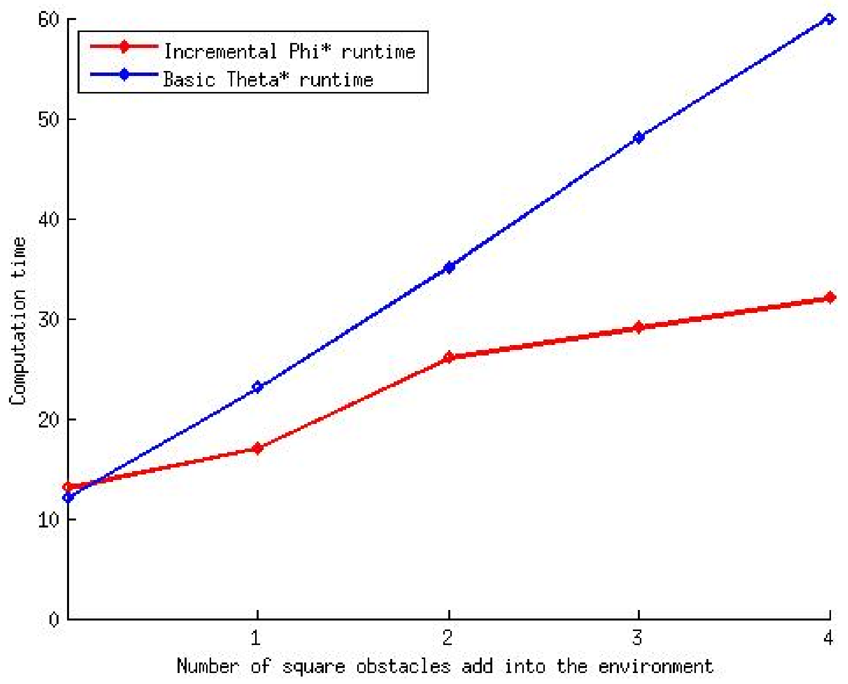


Figure 5.15: Avarage runtime for rising number of the obstacles

Chapter 6

Conclusion

The objective of this thesis is to show and discuss the advantages of any-angle path-planning algorithms for other workspace representations than grid, while achieving similar results for the grid workspace representation as in the Sven Koenig's work [11]. As it can be seen in the Experiments Chapter 5, the relative results of the any-angle algorithms are very similar to the results from Sven Koenig's work. This fact indicates, that the presented implementation is correct.

Following algorithms are implemented and they are tested in high amount of experiments : A*, Basic Theta*, Lazy Theta*, Phi*, Incremental Phi*.

By exploring the potential of any-angle algorithms in different workspace representation than a grid (namely triangulation mesh and tetrahedral mesh), following has been proven :
The any-angle algorithms finds shorter and smoother paths in comparison to the standard algorithms at the cost of increased runtime.
The any-angle algorithms provides better results in runtime and path length in unblocked areas of environments than standard algorithms.
Time complexity of the any-angle algorithms depends on the time complexity of line-of-sight check.

The usage of any-angle algorithms can be convenient in more complex path-planning problems, with high requirements on the shortest path (for example Exploration problem or the Inspection problem). The any-angle algorithms can be used in every representation where is able to execute a line-of-sight check. They can be also used in non Euclidean space.

It can be seen a great potential in any-angle algorithms because the plan in continuous space, which is important for some complex path-planning problems.

Bibliography

[1] Motion planning maps. http://imr.felk.cvut.cz/planning/maps.xml.

[2] The player project. http://playerstage.sourceforge.net/.

[3] The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[4] A.Nash S.Koenig C.Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d, 2010. http://idm-lab.org/bib/abstracts/papers/aaai10b.pdf.

[5] Doc.RNDr.Marie Demlova Csc. Prof.RNDr.edrich Pondelicek DrSc. *Matematicka Logika*. CVUT, 1999.

[6] Hart P. E., Nilsson N. J., and Raphael B. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE, 1968.

[7] Maur P. Kolingerová I. Post-optimization of delaunay tetrahedrization. http://herakles.zcu.cz/research/triangulation/index2.php.

[8] Pearl Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[9] A.Nash S.Koenig M.Likhachev. Incremental phi*: Incremental any-angle path planning on grids, 2009. http://idm-lab.org/bib/abstracts/papers/ijcai09d.pdf.

[10] Jonathan Richard Shewchuk. Tetgen a two-dimensional quality mesh generator and delaunay triangulator. http://www.cs.cmu.edu/~quake/triangle.html, year=2012, note=[cited 06 April 2012],.

[11] K.Daniel A.Nash S.Koenig and A.Felner. Theta*: Any-angle path planning on grids, 2010. http://idm-lab.org/bib/abstracts/papers/jair10b.pdf.

[12] Subhash Suri. Triangulation. http://www.cs.ucsb.edu/~suri/cs235/Triangulation.pdf.

[13] H. Choset K. Lynch S. Hutchinson G. Kan tor W. Burgard L. Kavraki and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[14] Weisstein Eric W. Triangulation, 2012. `http://mathworld.wolfram.com/Triangulation.html`.

[15] Jiří Žára Bedřich Beneš Jiří Sochor Petr Felkel. *Moderní počítačová grafika (2. vydání)*. Computer Press, 2005.

CD Content

The attached CD contains source codes for Any-angle algorithms, A* algorithm, triangulation program Triangle, tetrahedralization program TetGen, 2D and 3D maps, the thesis text in PDF format and source codes of the thesis text in LaTeXformat. In following Table is the CD structure described.

Directory	Description
`src`	source codes
`doc`	source codes of the thesis
`thesis.pdf`	thesis text in PDF format
`maps`	2D maps
`maps3D`	3D maps
`cfg`	2D grid maps

Table 1: Directory structure on CD