Generic Path Planning for Real-Time Applications

Christoph Niederberger, Dejan Radovic, Markus Gross

Computer Graphics Laboratory, ETH Zürich
{niederberger,grossm}@inf.ethz.ch, d.radovic@alumni.ethz.ch

Abstract

We present a fast and robust path planning algorithm for generic static terrains with polygonal obstacles. Our algorithm finds shorter, and therefore more intuitive paths than a traditional A approach with a similar underlying graph. The presented algorithm is derived from A* and is modified to circumvent undecidable situations and unintuitive results. Additionally, we present two post-processing steps to enhance the quality and visual appearance of the resulting paths. The first method minimizes the number of waypoints in a path while the second method takes the slope of the terrain into account in order to generate visually more pleasing paths. We show that our algorithm is fast and, therefore, well suited for real-time applications, such as games or virtual environments.*

1. Introduction

Open terrain navigation in static environments can be considered as a path planning problem where the task is to find a sequence of waypoints from a start to a goal location. Additionally, the line segments connecting the waypoints have to be collision-free with respect to obstacles. This task has been well researched and many approaches have been published.

For real-time applications such as games, the key to efficient path planning is how the environment is spatially decomposed. Therefore, the need for a level designer who defines a graph of landmarks still exists. This task is crucially to find a lean and correct graph representing the topology of the landscape, since the smaller the graph the faster will be the search on the graph. On the other hand, a smaller graph introduces an approximation error by reducing large areas to single points.

A generic path planning algorithm needs to meet several requirements:

- The resulting paths should have the lowest possible cost to prevent any indirection.
- It should be fast to not thwart the simulation process, it should be correct, i.e. no collisions occur, and it should be robust, i.e. the same request generates the same path.
- We would like an automatic approach to assure that no human interaction is necessary.
- Last but not least, the algorithm should be generic with respect to different maps, i.e. it should not be fully optimized for a specific map type.

We present a novel approach to fast path planning in generic terrains that meets the above mentioned requirements. The presented algorithm is a deviation of A* and processes static maps that contain polygonal obstacles. Our solution finds shorter paths which connect arbitrary start and goal locations than traditional approaches.

2. Related Work

Path planning problems arise in many different applications, and many publications have been written especially on robot motion planning (see [5][10] for surveys). Static environments and point-sized agents are the most simple case which we will consider here. They often occur in games and other virtual reality environments where physical correctness is not crucial. Because the game industry as the major beneficiary is not interested in publishing their internals of a game release, most solutions are given as generic descriptions in books [2][3][1][4].

For general environments, the most efficient and complete approach is the *roadmap* (or silhouette) method [5], or its variant, the probabilistic roadmap (PRM) [8][9]. Roadmaps reduce the agent's free configuration space C_{free} to a skeleton R_{free} which can be used to search a path from a given start to a goal location in C_{free}. The PRM is very useful in high-dimensional *C*-spaces. It searches randomly for configurations in C_{free} and connects them to a roadmap.

Generally, *graph search* based approaches use the vertices of a graph to represent feasible points in C_{free}, e.g. landmarks. Variations include methods based on Voronoi decomposition [11], and *cell decomposition* methods. For static, two-dimensional environments with convex polygonal obstacles and a point sized agent, these approaches afford efficient solutions, as surveyed by Schwartz et al. [13]. Exhaustive graph search algorithms, such as A*, constitute the only known optimal algorithms [7]. But the optimality is bound by the approximation error of the decomposition, since the vertices of the graph always reduce an area or line to a single graph node.

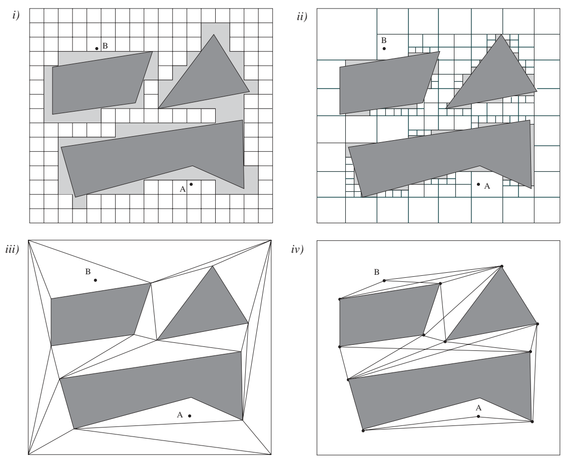


Figure 1: *Four different approaches to the discretization problem.*
i) Rectangular Grid
ii) Quadtree
iii) Convex Polygons
iv) Points of Visibility

When dealing with real-time applications, the path planning process has to be as fast as possible. Therefore, many simplifications are made. This often leads to more approximation errors. For exampe, path planning for the real-time strategy game *Star Trek®: Armada* is presented in [6]. It applies a quad-tree based decomposition of the playing field to reduce the number of cells. However, their rubber-band algorithm used to make the paths looking natural does not even achieve local optimality.

Multi-agent path planning, path planning for dynamic environments using D* [15] – a dynamic variation of A* – and dynamic replanning have been researched well, too, but are out of the scope of this publication.

3. The Problem of Path Planning

In order to let an object or character move inside a scene from one location to another, a path has to be planned that guarantees a collision-free translation from the start to the goal position. Hence, the whole task of path planning is usually broken down into four subproblems:

- First, one has to find a suitable discretization of the ground on which one can build a graph. This can be done offline in a preprocessing step. The resulting graph should be as lean as possible to allow a fast search. If the graph is too large, the search will be significantly slowed down. One the other hand, the discretization should be as fine as possible so that the areas corresponding to graph nodes are not too large. This would lead to an approximation error which ends up in suboptimal paths.
- For a specific path request, the task of point location determines the corresponding graph nodes for each the start and goal position. This depends heavily on the chosen discretization.
- Then, the graph has to be searched for a solution which connects the found nodes. For static environments as expected, the A* algorithm is commonly used.
- Afterwards, the resulting sequence of graph nodes needs to be transferred back to the original environment.

Therefore, the main problem seems to find an optimal trade-off between graph nodes representing spatially small areas (less approximation) and a small number of nodes (faster search). Additionally, we will show that unintuitive and suboptimal results can occur since the graph is fixed and cannot be adapted to a specific request.

We present a novel algorithm which partially overcomes the mentioned trade-off and finds near-optimal paths even for coarse discretizations.

4. A* Algorithm

The standard search algorithm for the shortest path problem in a graph is A*. It is a directed breadth-first search and combines the advantages of uniform-cost and greedy searches using a fitness function

$$f(n) = g(n) + h(n) \qquad n \in N,\ f{:}N \rightarrow \Re^{+} \tag{1}$$

where *g(n)* denotes the accumulated cost from the start node to node *n* and *h(n)* is a heuristic estimation of the remaining cost to get from node *n* to the goal node.

During the search, the A* algorithm maintains two lists of nodes: The *open list* contains the nodes that have to be considered next and the *closed list* which contains the nodes already visited. The algorithm itself consists of expanding the one node from the open list whose fitness function is minimal. Expanding a node means putting it into the closed list and inserting the neighbors into the open list and evaluating the fitness function. The algorithm stops, when the goal node gets expanded.

The choice of a good heuristic is necessary in order to achieve both quality and efficiency of the search. As long as the heuristic underestimates the real cost, the shortest path is guaranteed to be found. Nevertheless, underestimating can easily lead to an expansion of too many nodes. But when the heuristic is allowed to overestimate the remaining cost, faster results can be achieved because less nodes get expanded. If overestimating the distance to the goal, the A* algorithm tends to expand nodes that lie on the direct path to the goal before trying other nodes. But this can also lead to significantly slower searches if the final path contains directions that lead away from the goal.

5. Overview of the Algorithm

Our approach is a cell decomposition approach which uses a modified A* approach to find paths in a two-dimensional environment for a point sized agent.

We propose a solution which uses an automatic and coarse tesselation of the ground into trapezoids in order to obtain a generic discretization. On this tesselation, we build a graph whose nodes do not have a fixed position a priori. Rather, they are allowed to move freely on the portal between two neighboring cells. We do not specify the exact location of a node until necessary. Since we build the graph on the fly using the actual request and state we generate shorter paths than the traditional static approach.

For a specific path request, we locate the nearest portals and fix the corresponding graph nodes. Then, we use a modified A* algorithm to find a path on this graph. This algorithm uses two different strategies depending on the direction of the search in order to find optimal paths. When a node is expanded, its position gets fixed in order to calculate the heuristic function.

Additionally, one usually applies some post-processing to obtain a visually appealing form. We will present two methods that improve the quality of a given path.

6. Detailed Description of the Method

The following section discusses various aspects of our solution. First, we present the chosen discretization and improve the given algorithm to better fit our needs. Second, we discuss various approaches to build a graph on such a discretization and show that the traditional approaches fail to find optimal and intuitive solutions. Then, we present our modifications to A* and the resulting advantages.

6.1. Discretization

As explained above, the first task is to discretize the scene into obstacle-free regions. Different approaches to this problem are proposed in [2]. The first and trivial idea is to use a regular grid of some arbitrary resolution as shown in Figure 1i). This approach obviously leads to a very dense graph. Additionally, it can exhibit loss of connectivity, since some cells are only partially empty. A second approach is a quad-tree, depicted in Figure 1ii). The quad-tree displays a better approximation of the scene and a leaner graph, but still has a major drawback: At the borders of the obstacles there are many very small cells. Since many paths follow the borders of an obstacle the expected speed-up is partially lost. The problem with partially occupied cells remains, too. The third approach is to tessellate the ground into convex polygons, shown in Figure 1iii). Convex polygons have the useful property that any straight path inside the polygon can not collide with its border. Additionally, since the obstacles are polygonal, we can obtain a partition of the walkable ground without

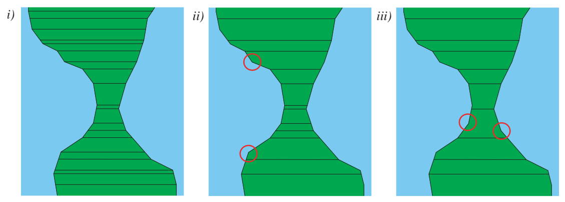


Figure 2: *Discretization of the green area into obstacle-free regions.*
i) The tesselation using Seidel's algorithm is not optimal with respect to the pathfinding problem.
ii) Merging neighboring cells into larger ones that still remain convex.
iii) Allowing for slightly concave polygons further reduces the number of polygons.

the loss of connectivity. Finally, a very lean graph can be achieved when using the points-of-visibility approach in Figure 1iv). For each obstacle corner, all other visible corners are connected to build a graph with a small number of nodes. However, the determination of the nearest graph node for an arbitrary location is rather difficult since a visibility-check is necessary for every potential node.

We have decided to use a tesselation into convex polygons due to its simplicity. To the end, we use the algorithm of Seidel [14]. This algorithm tessellates the ground into trapezoids with horizontal borders. These allow for very fast line intersection calculation which makes this approach very interesting. Additionally, the algorithm can also deal with polygons containing holes. This is very important since in our scenario the large polygon that defines the border of the map

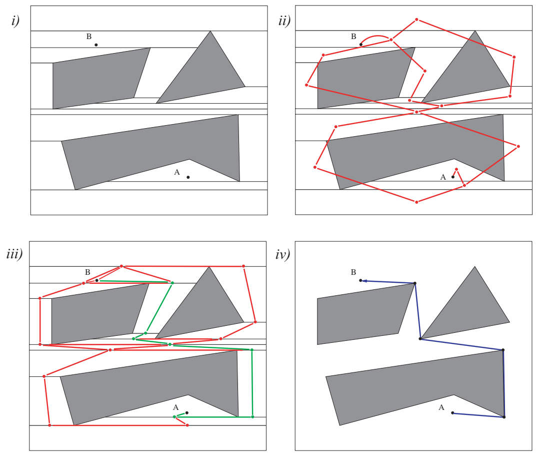


Figure 3: *Building the Graph:*
i) The tesselation using Seidel's algorithm.
ii) The graph using the trapezoid centers as nodes.
iii) Using portal centers as nodes. The resulting node sequence connecting A and B is shown in green.
iv) The resulting path.

contains other smaller polygons representing the obstacles. Additionally, the tesselation process automatically yields a query-tree that allows to efficiently handle the task of point location. Given an arbitrary point, its corresponding trapezoid can be found in O(logN) with N being the number of cells.

But the resulting tesselation of Seidel's algorithm is still suboptimal as depicted in Figure 2. We need to merge neighboring trapezoids into arbitrary convex polygons wherever possible in order to reduce the number of nodes in the graph. This algorithm is due to Hertel and Mehlhorn [12][16]. This process is not negligible. Tests on sample maps have shown that on average 50 percent of the trapezoids are eliminated. When allowing for slightly concave polygons by introducing a tolerance parameter even more trapezoids can be eliminated. The consequence is that paths can potentially intersect the border of the polygon. If this is inadmissible, one can simply expand the contours of the obstacles in a preprocessing step as described in [17].

6.2. Building the Graph

Building a graph on this tesselation leads to several possible approaches. One could use the polygon centers as graph nodes as shown in Figure 3ii). This strategy fails since we then have to find the corresponding portals[a] of each pair of adjacent polygons. Instead, we could use the portal centers directly as nodes, see Figure 3iii). While the resulting graph contains more nodes than the first approach, we get two

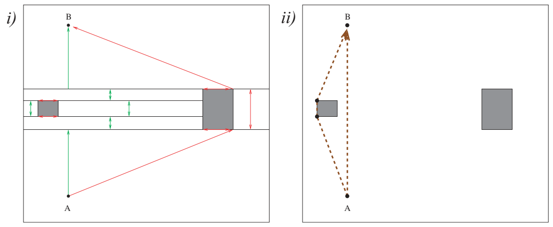


Figure 4: *Using whole portals as nodes leads to undecidable situations.*

advantages: First, the step to find portals is omitted. Second, the nodes represent more accurately the geometric locations relevant to the final path which allows for a better cost estimation between the nodes.

The presented solution works fine but has some major limitations. Consider the situation depicted in Figure 5i) where we search a path from A to B. A* will find the path leading *around* the obstacle instead of the expected straight path because the graph displays a shorter route around the obstacle. We see that choosing the portal centers as node locations is a rather arbitrary choice. Since paths dodging an obstacle follow its contour, we could try to anticipate this by placing

[a]A *portal* is defined as the connecting line segment between two adjacent polygons and acts as a doorway.

the nodes on the portal end-points. This solves the situation in Figure 5i), however, it fails in other situations as shown in Figure 5ii) where another obstacle is inserted. Additionally, the resulting graph has twice as many nodes as the previous one.

All problems encountered so far stem from the fact that long segments are reduced to just one or two points in the map and make it impossible to get a correct heuristic for A*. We could introduce a maximal portal width and split up broad polygons in order to reduce the deviation. However, this would lead to an even denser graph because of the additional interconnections between horizontally neighboring polygons. Also, note that no matter how small the portals will be, there are always counter-examples that produce unintuitive results.

When we abandon the idea of graph nodes representing precise locations, the idea to use the whole portals themselves as nodes ends up in a very lean graph again as shown in Figure 4.In order to construct the graph, we introduce a distance measure between portals – the minimal distance $d_{min}(s_1, s_2)$ between two segments s_1 and s_2 which is in most cases the vertical distance between two portals. This distance measure ensures that the total length of the final path is never overestimated and therefore the optimal path has to be contained in the set of possible solutions of the A* search.

When reconsidering the above examples, we see that although the first situation is solved correctly, no assertive answer can be given in the second situation, since both paths around the small obstacles have identical costs. As a consequence, the problem can not be decided and in fact, the outcome will depend on the implementation of the algorithm. Therefore, we have to find a modification that makes this approach robust.

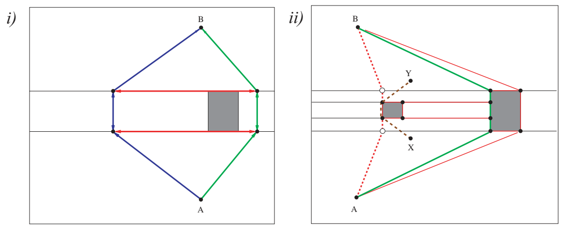


Figure 5: *A graph built with portal centers can lead to unintuitive results:*
i) The green sequence will be chosen instead of the blue one which finally leads to a longer path.
ii) When placing the nodes on the obstacle borders, the approach finds the correct solution for example i), but fails to find the correct solution when adding a second obstacle. In this case the green path will be chosen. Adding more nodes (white) can solve this (dashed red), but it will fail when connecting X and Y (dashed brown).

One possibility could be to tweak the A* heuristic function in which estimates the remaining cost to the goal node. Since we are free to chose the function it allows to prefer portals that lie on a straight line between the start and goal location. Adding a term that enforces the A* search to expand nodes that lie near to the connecting line also considerably improves the efficiency of it. This technique is vital to open terrain navigation since in most cases the paths are straight or deviate only little from the straight connection between start and goal. However, the search is significantly slowed down when the path is forced to lead away from the goal location. Also, the computational expense to calculate such a heuristic function further slows down the search. A further limitation is that the search direction is always attracted by the straight line connecting the start and goal location. When the path leads away from this line it will tend to return rather than moving ahead from its current position to the goal since the heuristic is only globally defined. Instead we would like to have a local search that depends also on the current position.

When looking for a solution that unifies the advantages of the above approaches while eliminating the incorrect solutions we have to find one with a small number of nodes and accuracy with respect to distance measures. The fundamental problem of the whole approach so far is that the graph is built offline in a preprocessing stage. It stays fixed for its lifetime with the exception of the start and goal node that are introduced for a search. Therefore, there is no way to bring in the information of a *specific* pathfinding request into the structure of the graph.

6.3. Modifying the A* Algorithm

As we have seen, the major drawback is that no specific information of a request can be included into the graph since it is built offline. Our solution to this problem is that we do not specify the exact location of a node until neccessary. The location is set according to the previous course of the path using two different strategies.

When moving away from the goal location, we do not know exactly in which direction the final path will lead. Therefore, we use a *lazy strategy* and select the closest possible location on the next portal. This results in a minimal deviation from the final path. Here, we still introduce an approximation error since the final path will most likely not traverse this exact location. This situation is depicted in Figure 6i) where the red as well as the blue path can lead to the goal. When moving towards the goal location, we use a *greedy strategy* since we know where to go. This means that we set the node as close as possible to the straight line connecting the actual location with the goal. An example where this strategy is used can be seen in Figure 6ii) and iii).

Using these two strategies, we can take advantage of several facts. First, the graph is still small and has a minimal number of nodes with respect to the tesselation. We do not have to introduce additional nodes that slow down the entire

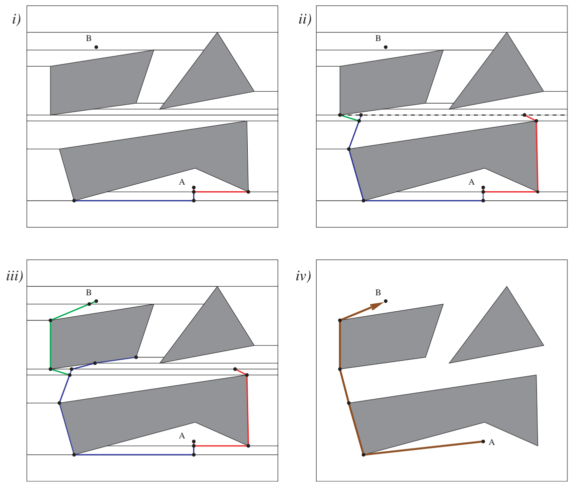


Figure 6: *Using two different strategies:*
i) The lazy strategy when moving away from the goal places the nodes as near as possible on the next portal.
ii) and iii) The greedy strategy is applied when moving towards the goal and places the node on the straight line to the goal.
iv) The final path. Note the difference between this solution and the one found in Figure 3. This one is actually shorter.

process. Second, we have less approximation errors since we have no spatial approximation when using the greedy strategy. Only the lazy strategy introduces an overestimation of the path length which is bound by the factor $\sqrt{2}$. This worst case occurs when we decide to move vertically instead of diagonally within a square. Third, we do not have to tweak the heuristic as presented in the last section since every situation can be decided entirely. Considering Figure 6ii), the red and green path both lead to the dashed line. Assuming that both of them have the same cost up to this point, A* could not decide which one to favor since both are indistinguishable. Our approach allows for an online distinction because both paths reach different locations on the portal, hence, their predictions for the remaining cost differ and make both paths distinguishable.

Last but not least and as a result of the above mentioned advantages, we can achieve shorter paths than the traditional approach. Consider for example the final path shown in Figure 6 and compare it to the solution shown in Figure 3. Actually, the path found in Figure 6 is a little shorter than the other.

Of course, our approach has also some drawbacks. It seems to be slower than the conventional approach because we have to additionally evaluate the distance between two successive nodes. Also, we have to calculate the intersection between the straight line to the goal and the next portal. But since our portals are always horizontally, this can be done very fast and is therefore negligible.

7. Postprocessing

After having found a sequence of nodes in the graph using the above presented algorithm, these nodes do not form an optimal path since we still have more nodes than necessary – one for each traversed portal. Now, we have to abandon the graph and return to the original map and apply a postprocessing of the resulting node sequence. Additionally, we adapt the path to a 3D environment by incorporating the slope of the terrain when moving on the path.

7.1. Path Optimization

The goal of this postprocessing stage is to find a sequence of waypoints that constitutes a path with a minimal number of waypoints. In order to achieve that, we propose a *cone-of-sight algorithm* which is based on visibility of points and portals.

Again, our algorithm uses the advantage of the horizontal portals by Seidel's algorithm. Without loss of generality, we assume to move vertically upwards and will present our algorithm with the example given in Figure 7i). We start at point A and have a sequence of portals to pass. We have two different fourth portals to show how the algorithm works in different cases.

Our starting point and the first portal form together a cone of sight as depicted in Figure 7ii). The algorithm keeps track of three points *L*, *R*, and the cone's starting point, in this case *A*. *L*, respectively *R*, denote the borders of the cone. When looking at the next portal in Figure 7iii), its left end lies inside the cone. Therefore, the left border is narrowed by placing *L* to this point. Since the right end of this portal is outside our

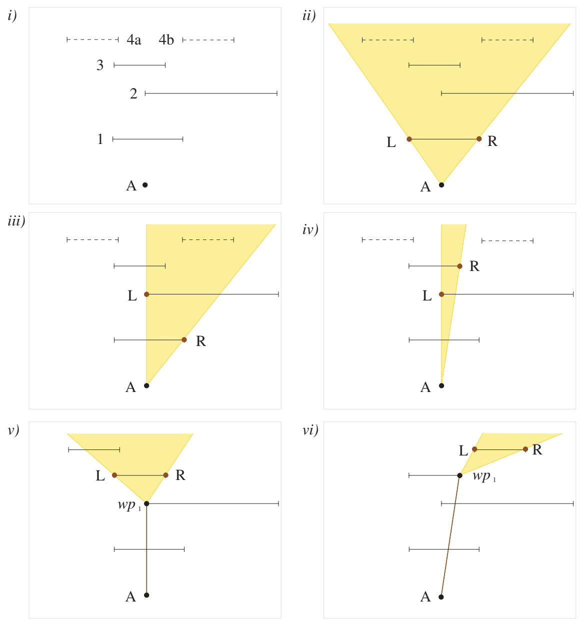


Figure 7: *The cone-of-sight algorithm is used to find the optimal sequence of waypoints.*

cone, it will not restrict our current search. Looking at the third portal in Figure 7iv) further restricts our cone, this time from the right side. This time, *R* is moved inwards and lies now on the right end of the third portal. These steps are continued until we reach a portal that completely lies outside the cone.

Considering portal 4a in Figure 7iv), we see that it is outside the cone on the left. We therefore have to place a waypoint on the location of L and restart the algorithm at this point. The new situation is shown in Figure 7v). This backtracking step explains why we have to keep track of *L* and *R*. Considering alternative 4b, we see that the portal lies on the right side of the cone. Therefore, we have to set *R* as the first waypoint and continue from the third portal as depicted in Figure 7vi).

This algorithm guarantees that the shortest path for a given portal sequence is found. Thus, the straightness criterion for paths introduced in the third section is met. For example, the Rubber Banding Algorithm presented in [6] does not achieve this. Our algorithm benefits from the fact that portals are always horizontal which allows again for fast intersection calculations.

7.2. Movement in 3D

The system is now able to efficiently compute qualitative paths but still constitutes a purely 2D navigation facility. This section presents an approach how to extend the system in order to take height information into account. This post-processing step is not necessary but results in aesthetically pleasing paths.

Instead of having agents that strictly follow the path in a straight line from one waypoint to another, we allow them to diverge to some degree. We propose an approach, where the path is adapted to the slope of the terrain. In order to avoid steep slopes, we turn the agent away from its ideal course using the current position $\boldsymbol{p}$ of the agent, the next waypoint w_i, and an arbitrary function $f: \Re \rightarrow [0, \angle max]$, where $\angle max$ is the maximal deviation angle. This results in

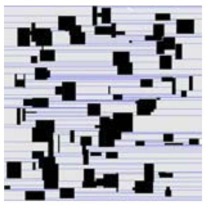
$$\delta(\boldsymbol{p}) = f(\|\boldsymbol{w}_i - \boldsymbol{p}\|)\|\nabla(\boldsymbol{p})\|\cos(\varphi), \qquad (2)$$

where

$$\varphi = \angle(\boldsymbol{w}_i - \boldsymbol{p}, \nabla(\boldsymbol{p})). \qquad (3)$$

$\delta(\boldsymbol{p})$ in Equation 1 denotes the deflection from the angle pointing to the next waypoint w_i. This formula is only applied when $\varphi > 0$, thus, when the agent is ascending. The first term $f(\|\boldsymbol{w}_i - \boldsymbol{p}\|)$ ensures, that the deviation decreases as the waypoint is approached. In order to assure that the goal point w_i is really reached, the function f should become 0 when the distance $\boldsymbol{w}_i - \boldsymbol{p}$ decreases. The second term $\|\nabla(\boldsymbol{p})\|$ accounts for the steepness – the greater the length of the gradient, the larger the deviation form the straight path will be. The last term $\cos(\varphi)$ weighs the direction of the gradient against the orientation of the movement. The deviation is

Table 1: *The characteristics of the maps used for the results. The last three columns show the number of nodes of the graphs built on the maps. The last two columns are derived from the original map by setting the maximal portal width to 330, respectively 100, units.*

	Loose Map	Dense Map
		
Size	1000x1000	1000x1000
# Obstacles	100	200
Center Nodes	281	395
Width 1/3 Nodes	323	408
Width 1/10 Nodes	567	568

maximal when the next waypoint lies in the direction of the maximal steepness. If the slope is parallel to the moving direction ($\varphi = 90°$) the deviation is zero.

Of course, such an adapted path could pass an obstacle region, because it deviates from the original path. But since every waypoint is reached exactly and waypoints usually lie at the border of an obstacle, this problem is negligible. Additionally, the user has the possibility to specify the maximal deviation angle to keep the agent near the original path. Furthermore, one could enlarge the obstacle regions in order to introduce a tolerance bound around each obstacle [17]. Figure 10 shows the adapted path in red in comparison with the planned path in white.

8. Results

In order to compare our approach with different approaches we set up a test suite that automatically generates paths. The measurements were taken by calculating paths with random start and goal locations on two different maps. The characteristics of these maps are outlined in Table 1. We have compared our approach with three implementations which have been presented in this paper:

- Center: This approach uses a static graph built on the portal centers of the tesselation.
- Width 1/3: The maximal portal width of the above approach has been set to a third of the map width and the portals are connected with the maximal fanout.
- Width 1/10: The maximal portal width has been set to a tenth of the map width with an according fanout.

All these implementations use the same underlying A* mechanism with the same performance optimizations as our

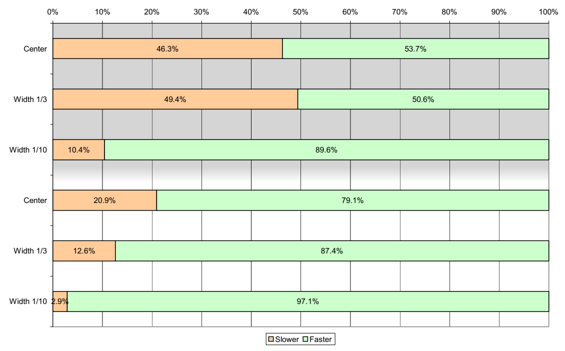


Figure 8: *Comparing the run time of traditional approaches with our approach. The order of the rows is the same as in Figure 9. Green denotes cases where our approach was faster.*

approach. These approaches also use the path optimization procedure presented in Section 7.1.

In more than 90% of the cases our approach finds paths of equal or shorter length as shown in Figure 9. The paths are absolutely shorter than the other approaches in over 50% of the tests. This result is even more distinct on the loose map with less obstacles (97%, respectively 60%). Using a maximal portal width only slightly improves the quality of the result as can be seen. On the loose map this technique is more effective since the portal width is more likely to be large.

Comparing the path generation time of the different approaches, we see in Figure 8 that our approach is competi-

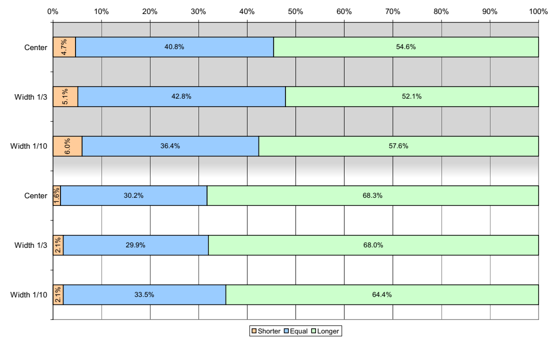


Figure 9: *Comparing the path length of traditional approaches with our approach. The top three rows (gray) were generated on the dense map, while the bottom three rows (white) used the loose map described in Table 1. On each map we compared our approach with three different implementations that use static graphs. Red denotes cases where traditional approaches found a shorter path, blue for equal length, and green when our approach found a shorter path.*

tive. On the dense map (top rows), the algorithms with approximately the same number of nodes perform similar while the difference grows with less obstacles. The approach with the maximal portal width set to a tenth of the map width always performs slower since its graph has more nodes as shown in Table 1. For our approach, absolute time values are 0.19 ms on average on the loose map and 0.28 ms on the dense map on a 1 GHz Pentium III computer with 512 MB RAM.

9. Conclusion

We have developed a fast and robust path planning algorithm for generic static terrains with polygonal obstacles. The presented algorithm is based on the A* algorithm and results in paths that are on average shorter than traditional approaches

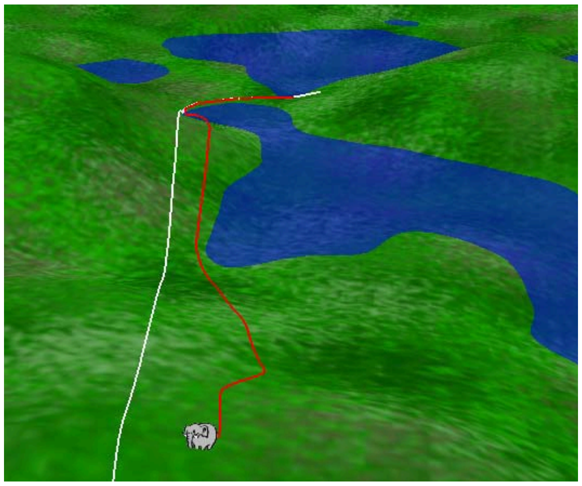


Figure 10: *The resulting path (white) and the movement adapted to the slope of the terrain (red).*

and the computation takes approximately the same time than other approaches. It does work on a graph with moveable nodes and uses two different strategies, the lazy strategy when moving away from the goal location, and the greedy strategy, when moving towards the goal location. This allows for a better cost estimation and, therefore, shorter paths. Additionally, our algorithm circumvents unintuitive results.

Due to a postprocessing step, the generated paths are made up of a minimal number of waypoints by removing unneccessary ones. Additionally, we presented a method to visually enhance the impression of natural movement by taking the slope of the terrain into account during a post-processing step. An example of a resulting path (white) and the actual movement (red) is depicted in Figure 10.

Our algorithm is well suited for real-time applications such as games or other virtual environments with artificial entities. We have successfully implemented the algorithm in a real-time simulation environment where multiple autonomous agents live in an artificial world and use the algorithm to find paths around lakes and other obstacles.

References

[1] E. by Dante Treglia. *Game Programming Gems 3*. Charles River Media, Hingham, MA, 2002.

[2] E. by Mark DeLoura. *Game Programming Gems*. Charles River Media, Hingham, MA, 2000.

[3] E. by Mark DeLoura. *Game Programming Gems 2*. Charles River Media, Hingham, MA, 2001.

[4] E. by Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Hingham, MA, 2002.

[5] J. Canny. *The Complexity of Robot Motion Planning*. MIT Press, 1988.

[6] I. L. Davis. “Warp speed: Path planning for star trek: Armada.” In *AAAI Spring Symposium Technical Report (2000 AAAI Spring Symposium)*, 2000.

[7] T. Hu, A. Kahng, and G. Robins. “Optimal robust path planning in general environments.” *IEEE Transactions on Robotics and Automation*, 9:775–784, 1993.

[8] L. E. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan. “Randomized query processing in robot path planning.” In *ACM Symposium on Theory of Computing*, pages 353–362, 1995.

[9] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. “Probabilistic roadmaps for path planning in high dimensional configuration spaces.” *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[10] J. C. Latombe. *Robot Motion Planning*. MIT Press, 1969.

[11] C. O’Dunlaing and C. Yap. “A retraction method for planning the motion of a disk.” In *J. Algorithms*, pages 187–192, 1985.

[12] J. O’Rourke. *Computational geometry in C*. Cambridge University Press, 1994.

[13] J. T. Schwartz, M. Sharir, and J. E. Hopcroft. *Planning, Geometry and Complexity of Robot Motion*. Ablex Publishing Corp., 1987.

[14] R. Seidel. “A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons.” Technical report, "Institute for Computer Science", Department of Mathematics, Freie Universität Berlin, 1990.

[15] A. Stentz. “The focussed d* algorithm for real-time replanning.” In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[16] P. Tozour and I. S. Austin. “Building a near-optimal navigation mesh.” *AI Game Programming Wisdom*, pages 171–185, 2002.

[17] T. Young. “Expanded geometry for points-of-visibility pathfinding.” *Game Programming Gems 2*, pages 317–323, 2001.