Time-sliced pathfinding on arbitrary polygon surfaces

Arvid Norberg <c99ang@cs.umu.se>

supervisor:
Michael Minock <mjm@cs.umu.se>

Abstract

Real-time games need to maintain a smooth frame rate and cannot execute classical pathfinding algorithms in a single frame. Such algorithms have to be sliced up to span multiple frames, where each frame has an execution time limit. This thesis will explore and implement a technique for time-sliced pathfinding that avoids frame drops and has straight paths, as a result from having a continuous spacial representation.

This is done by using a polygon mesh to represent the environment. The requirements of the pathfinder is that the entity, whose path to be is found, can have different sizes and thus can be forced to take different routes depending on the size. Another requirement will be that the pathfinder will make guesses about the best way to go before the whole search is complete, to make the entity react immediately when given a goal to go to. If the guesses turns out to be the wrong way, it will have to find its way back to the correct path.

Table of contents

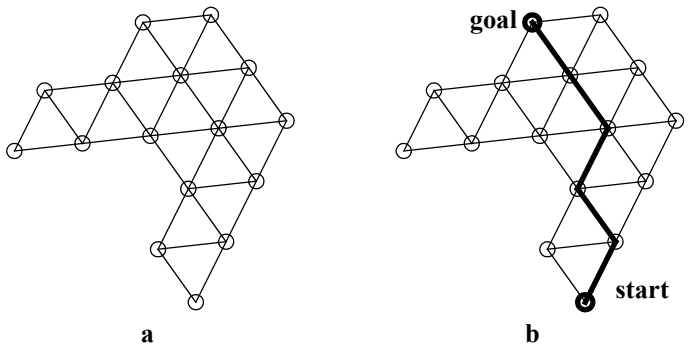
1 Introduction 3
2 Background 5
2.1 A* algorithm 5
3 Resource-bounded search 7
3.1 Guessing the path 7
3.2 Joining the paths 7
3.3 Time slicing 8
4 Navigation mesh 10
4.1 Points of visibility 10
4.2 Supporting wide entities 12
4.3 Performing the search on the mesh 13
4.4 Movement on a mesh 14
4.5 Packaging the pathfinder 14
5 Results 15
6 Conclusions 19
6.1 Floating point issues 19
6.2 Movability limitations 20
6.3 Suitable environments 20
6.4 Path optimality 20
7 References 22

1.0 Introduction

The traditional world representation for pathfinding is a finite graph, where agents (entities), can stand on the vertices and walk along the edges. The graph is placed in the world in such a way that no edge intersects an obstacle, which means that if an entity walks only on the edges it will avoid collisions. In this thesis, this kind of graph will be referred to as a finite movement graph, since it is a discrete approximation of a continuous environment.

A finite movement graph, as shown in Figure 1 a, describes the exact vertices where an entity can be located, and between which vertices it can move. This makes it straightforward to do graph-based path finds on this representation.

FIGURE 1. A finite movement graph



The good thing about finite movement graphs is that it is very easy to introduce dynamic obstacles, since you can just map the obstacle onto the spacial graph and remove the obstructed vertices. If the graph is dense enough, the pathfinder will find its ways around the obstacle.

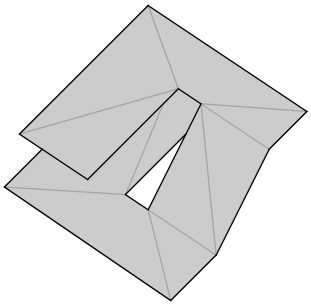
While it may work very well in a maze, it is harder to find good paths with the finite movement graph approach in a more general, indoor, setting. Since the path between two points may pass several vertices, when it in fact could walk in a straight line, the path may be crooked, as shown in Figure 1 b. This could partially be solved by having a very dense graph, but that would use a lot of memory.

A finite movement graph usually do not contain any information on how big the entity that walks on the graph may be. And since different sized entities may have to take different routes, something has to be done. This can be solved by having one graph per entity-size. This will use unnecessary amounts of memory, and computer games often live in an environment greatly limited by both memory and CPU resources. You could argue that you just have to encode width information in the edges. That would not suffice, since the vertices may also have to be moved farther away from walls and obstacles in order to avoid collisions.

Many of these problems can be tackled by using a navigation mesh [4] (also known as ground mesh [6]). A navigation mesh is a polygon mesh describing the surface where entities can walk. That means that the entities are free to move on a surface rather than being stuck on one-dimensional edges. It models the world as a continuous space. An example of a navigation mesh is illustrated in Figure 2. It is a two level mesh with a ramp connecting the levels.

The entities can be located anywhere on any polygon in the mesh. The polygon surfaces form a constraint on the entities' positions that cannot be broken. The pathfinder will of course have to respect this constraint and only find paths along the surface.

FIGURE 2. A navigation mesh



With a navigation mesh, the representation will use less memory, since it can cover a large continuous area with a single polygon, instead of having to fill that area with a dense graph. Entities will also be free to move anywhere in this polygon mesh, which leads to straight paths along the surface. By using navigation meshes, the number of collision tests may be reduced in the game. Since the positions of the entities are already maintained within the mesh boundary, they usually do not have to be collision tested against the actual world. The navigation mesh is a good enough approximation of all the obstacles.

In many computer and video games the size of the AI controlled entities can vary so much that the same collision geometry cannot be used for all of them. If all entities collides as if they all had the size of the largest entity, the small bugs will not get through paths that are too small for the giant robots. And the reverse, if all entities collide as if they were as small as the smallest, the giant robots would fit into the small hole where the bug crawls.

One popular approach to this is simply to duplicate the collision geometry once for each entity size. This is for example used by Quake [5]. Quake has three different sizes of their collision geometry for three different sizes of their entities. This technique suffers from scalability problems, since the memory footprint will increase for each entity size that is added. Another limitation is that the size of an entity must be known beforehand, and cannot change during run-time.

When having the navigation mesh available, all necessary information for having entities with a variable radius is available. All the walls are known and searching the path is a matter of offsetting the way points from their corners. More details on how this is done will be discussed later.

Since searching an arbitrary sized graph may take arbitrary time to solve, it is not suitable as a blocking operation for most games. The search has to be sliced into smaller parts that can be executed once every frame, or game step. This is a concept called time-slicing and it has to be implemented in a pathfinder in order to make it suitable for games.

In computer games, there is usually no point of simulating a robot that has to learn and only see the world from its own point of view (unless that is a part of the game itself).

The agents in computer games usually have access to the entire world representation in some way. The reason for this is that it is much easier to make seemingly intelligent decisions at a relatively low cpu and memory cost that way.

The pathfinding problem is divided into two major parts. The first is the pure graph problem, where the entity should find its way and make best guesses until the actual path has been found. The second part is how this graph can be applied to a generic 3D mesh, and how a graph can be calculated from the mesh.

The mesh pathfinder must be able to take into account the width of the entity. Preferably the entity width could be changed during run-time and not be significant to the memory footprint. It must also allow the operation to be sliced into a number of smaller operations, to make it possible to maintain a high frame rate.

2.0 Background

The pure problem of searching a graph is pretty straight forward. Since we know that the graph represents actual space, we know that there cannot be a path from one point to another that is shorter than the straight line (euclidian distance). This means we can use A* to search the graph instead of a uniform-cost search.

2.1 A* algorithm

The A* search algorithm [3] [8] is a special case uniform search with a specific heuristic function and a specific meaning of the weight of an edge.

The weight of an edge $d(E)$ must be the length of the edge E.

The heuristic for a vertex $h(V)$ must be the euclidian distance from V to the goal vertex.

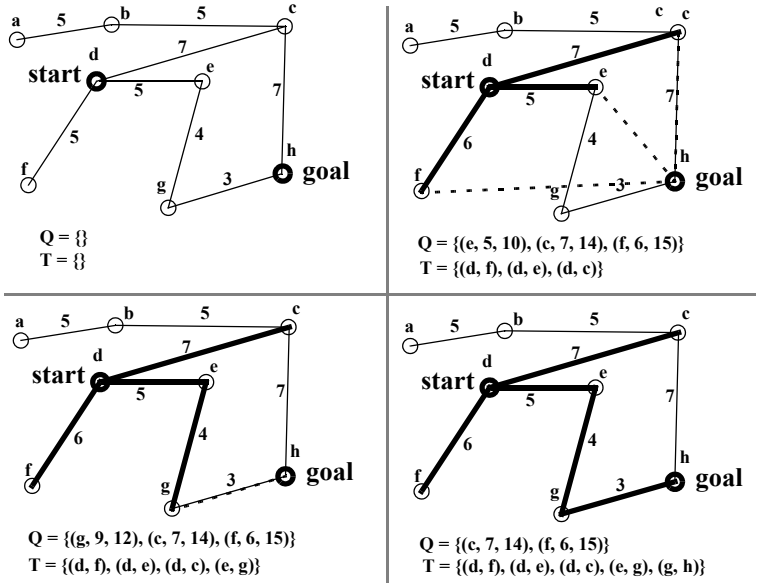
1. let Q be a priority queue of vertex, distance and weight tuples, sorted after weight
2. let T be a vector of vertex pairs, representing a path segment that has been inspected in the search
3. let $d(E)$ be the edge weight function, calculating the distance of the edge E
4. let $h(V)$ be the heuristic function, calculating the euclidian distance from the vertex V to the goal vertex.
5. let S be the start vertex and G be the goal vertex
6. $D = 0$
7. for every non-visited adjacent vertex V of S, where E is the edge connecting them
   7.1. push $(V, D + d(E), D + d(E) + h(V))$ onto the priority queue Q
   7.2. append (S, V) to T
8. pop the top element (V, D, W) off of the priority queue Q, which minimizes W

9. if $V \neq G$, set S to V and go to 6

10. we have found the shortest path from start to goal, pass T to the backtracker

An example of this algorithm is illustrated in Figure 3. The dashed lines are the distances measured by the heuristic function. The numbers associated with each edge is its distance. The tuples in the priority queue Q corresponds to those in the algorithm: (vertex, distance, distance + heuristic).

FIGURE 3. Example of an A* search. Dashed lines are the heuristics straight line distance



The vector T now contains the path segments that will be used to build the final path from start to goal. This process is called backtracking.

1. let $P = \{\}$

2. pop the last element (A, B) from the vector T

3. prepend (A, B) to P

4. if $B = G$ then we are finished, quit

5. pop the last element (C, D) from the vector T

   5.1. if $A \neq D$ then go to 5

6. go to 2

Table 1 shows an example of this algorithm applied to the resulting vector T from the previous example. It will produce the final path P.

TABLE 1. Example of backtracking

Step	T vector	Final path P
1	{(d, f), (d, e), (d, c), (e, g), (g, h)}	{}
2	{(d, f), (d, e), (d, c), (e, g)}	{(g, h)}
3	{(d, f), (d, e), (d, c)}	{(e, g), (g, h)}

TABLE 1. Example of backtracking

Step	T vector	Final path P
4	{(d, f), (d, e)}	{(e, g), (g, h)}
5	{(d, f)}	{(d, e), (e, g), (g, h)}

3.0 Resource-bounded search

The pathfinder will be limited to a certain amount of time at each step, and as long as it has not found a solution, it will have to guess. In the end it will have to produce a correct path that takes the previous guesses into account, to make the agent find its way.

The implementation of the A*-algorithm allows for changing the edge weight and heuristic functions ($d(E)$ and $h(V)$) from the outside. This makes it possible to implement agents that avoid certain points or tries to remain invisible from certain points by simply having the weight depend on those things.

3.1 Guessing the path

While the pathfinder is searching there are two implemented options. Either guess which way to go and correct it later, when the actual path is found. The other option is to continue walking along the current path, and correct it when the actual path is found. The latter is interesting if the destination is likely to be very close to the previous destination, and the previous destination was never reached.

The path guessing is done by selecting the edge whose direction vector points most in the direction to the goal. i.e. the edge that is selected is the one with the greatest dot product against the goal vector. Vertices that already have been visited during the guessed walk are ignored to avoid having the entity walk in circles.

This method gives a fairly good estimation of the general direction in which the entity should start moving. And even if the guessed edge is wrong, it may still look good and be efficient if the general direction is correct, since the final path will be based on where the entity is located by the time the A* search is completed.

To make the entity move smoothly even when the entity is not allowed to guess its path, the possible previous path will be copied to the guessed path to make the entity follow it instead of guessing. This approach may be very efficient since there usually are temporal coherencies in consecutive searches. By considering the old path as the same thing as the guessed path the path-joining will be made easier and no special case has to be made.

3.2 Joining the paths

When the search is finished the path the entity has moved and the real path has to be connected in a way so the entity will move from its current location to the goal. The entity's current location may not be the start position, since it has moved in guessed directions.

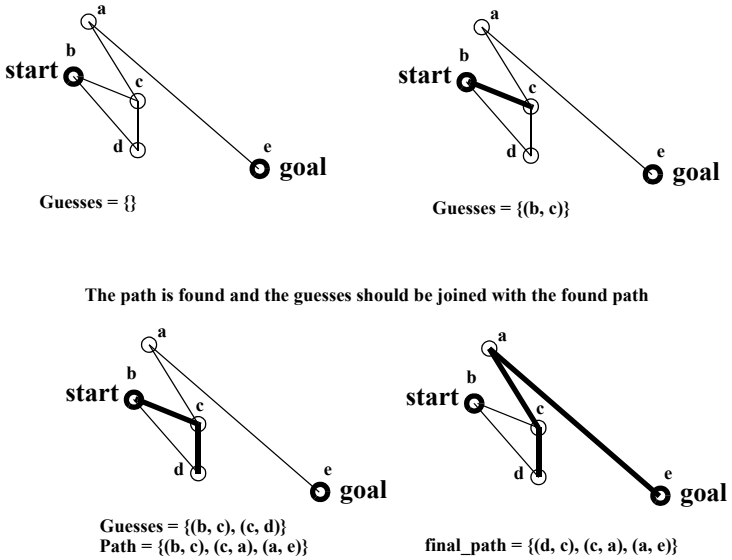
The naïve solution would be to simply backtrack the guessed path to walk to the start vertex and then continue to walk along the real path. It would be guaranteed to work, but it would defeat the purpose of doing the guesses in the first place, since the entity would always have to go back to the start position when the path has been found.

Instead, the following method is used to join them and make the resulting path as short as possible. The two paths are represented as two lists of vertices. Both starting with the start vertex (as the start of the search).

1. let G be a list of vertices in the goal path, starting with the start vertex
2. let P be a list of vertices in the found path, starting with the start vertex
3. let Q be an empty list which will be built to the final path
4. for each vertex V in G, in reverse order
   - 4.1. append V to Q
   - 4.2. for each vertex W in P, in reverse order
     - 4.2.1. if W is the same vertex as V then append all vertices we have passed in P to Q and go to 7
     - 4.2.2. if there exist an edge that connects V with W then append W and all vertices we have passed in P to Q and go to 7
5. remove the first vertex from P (the start vertex)
6. append reverse(P) to Q
7. Q is now the path from the entity to the goal

Figure 4 shows an example of this algorithm in progress.

FIGURE 4. Example of a guessed path



3.3 Time slicing

The time slicing mechanism is a fairly simple framework which allows threads to be registered. Here, a thread is simply a function object with a virtual member function that executes one slice of the thread's work.

The requirement is that the time-slicer does not run for more than a specified time, or rather, that it stops executing and returns as soon as possible when the maximum allowed time has been consumed. Therefore the time slicing framework has to check the time after each slice has executed to see if it should stop.

This means that each call to a thread's work function cannot take too much time. The maximum time that call takes will define the granularity of the slicing and sets the error bounds of the time slicer. If the slices are big (i.e. take long time), the granularity will increase and the amount of time spent will not be as controllable. As a result the time-slicers limit has to be lowered. If the slices are small (i.e. is quick to execute), the granularity will decrease and the time slicer will have more control over the time it spends. Another result is that more time is spent querying the clock for the current time.

The algorithm puts all running threads in a priority queue, where they are sorted according to the time they want to be executed next time.

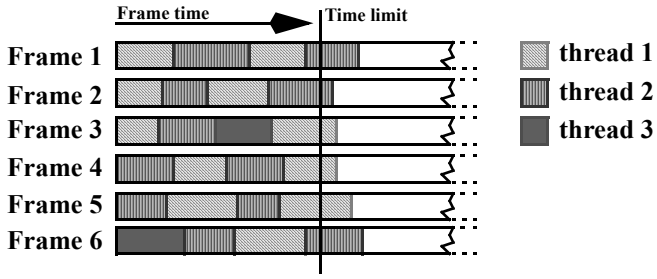
1. let Q be a priority queue of threads
2. let S be the current time
3. if Q is empty, there is nothing to do, quit
4. pop the top element T from Q
5. execute the slice for T
6. update the time when T want to execute next time
7. if T was not removed from the slicer, insert T into Q
8. if the (current time - S) is less than the allowed time, go to 3

Some care has to be taken if the thread was removed from the time slicer's priority queue while executing, but that is a minor detail.

The pathfinder has a thread that is started when a new search is initiated. This thread will do a fixed number of A* iterations per slice. It is then called each frame (or more seldom depending on other activity). When the search is finished the thread is removed and the pathfinder has its path.

The example in Figure 5 shows three threads that are active during 6 frames. Thread 1 and thread 2 tries to execute as often as possible, while thread 3 has a periodicity of 3 frames. Note that the time limit is not a hard constraint, but a bit looser defined as a stop condition. On average, it is overrun by half the granularity of the threads.

FIGURE 5. Example of three time sliced threads



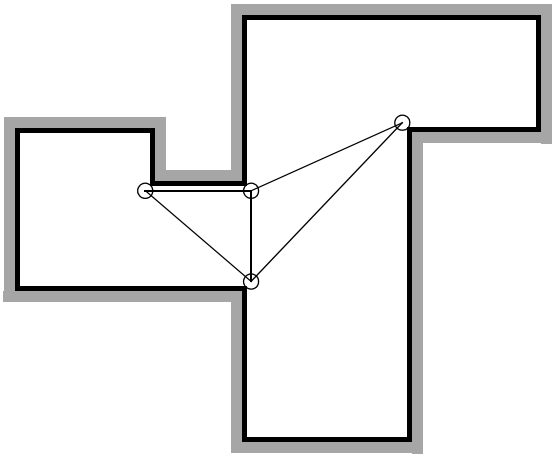
4.0 Navigation mesh

When moving on to the problem on a 3D mesh, there are a number of issues one have to deal with. To be able to apply a normal graph search on a 3D mesh, a graph has to be generated from it. This graph should contain all points where a path could naturally make a turn.

4.1 Points of visibility

To build the smallest necessary graph, a “points of visibility”-graph [1] (or “visibility graph” [8]) is used. The points of visibility graph is created by first determining the convex corners of the obstacles. Which will be the same as the concave corners of the navigation mesh. These points are then connected with all other visible corners. In this context, visible means that an entity can walk in a straight line without hitting a wall. The resulting graph is illustrated by Figure 6.

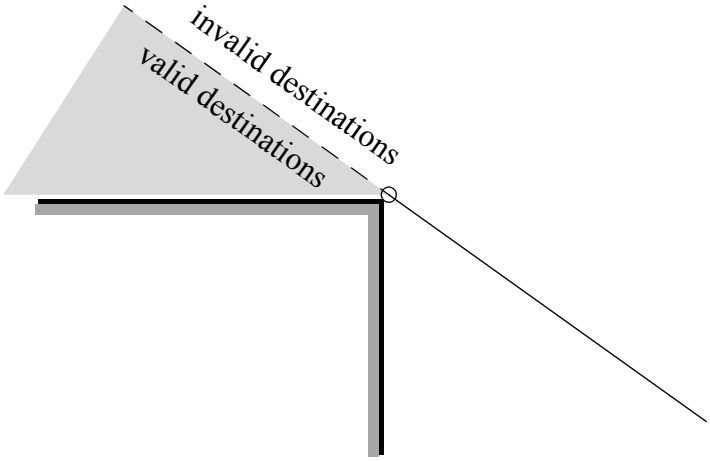
FIGURE 6. A points of visibility graph with all vertices that sees each other connected



The idea behind the visibility graph is that it creates a vertex in every spot where the shortest path will have to be able to bend. If the straight path is obstructed by something, we can be sure of that we have to go around a convex corner, that is why all the graph vertices are placed in convex corners. With this assumption we can apply some heavy optimizations to the number of edges in the graph. We can remove all those edges that do not represent a valid “go around the corner” of a convex corner.

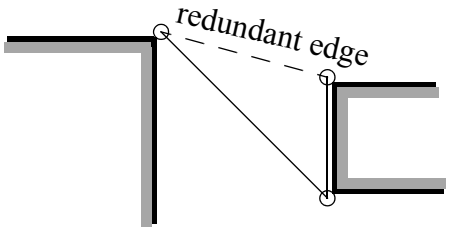
As Figure 7 shows, the path has to be bent in the same direction as the corner is, otherwise there will always be a shorter path to the goal. i.e. If the path want to turn right when it reaches the corner in Figure 7, it can always take a straight line, or bend at a possible other corner that is in its way. This is a run-time optimization, while searching you do not have to follow edges that do not meet this requirement.

FIGURE 7. Any vertices outside the marked region can be ignored while searching



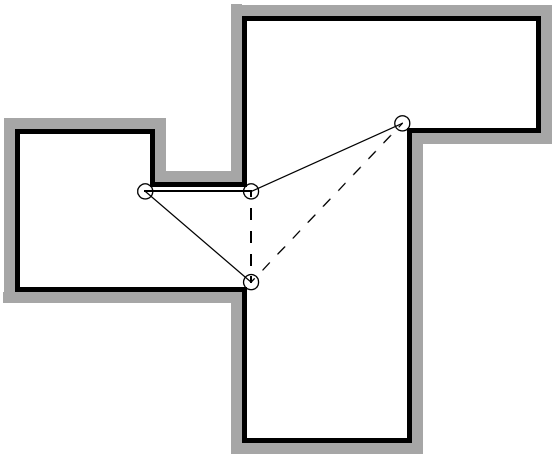
The next optimization is a static one. It will remove redundant edges that will never be valid, given the first requirement. The example in Figure 8 shows an edge that can never meet the requirement, no matter from which direction the path comes. The case is easily identified by seeing if the other end corner of the edge is on the outside of both the wall-planes involved.

FIGURE 8. Any edge that is connected to a vertex that cannot be valid can be removed



The resulting graph after this optimization has been applied will be the one in Figure 9. Two edges could be removed, illustrated by dashed lines. This will not only make the search quicker, it will also give the graph a smaller memory footprint.

FIGURE 9. The same graph with unnecessary edges removed

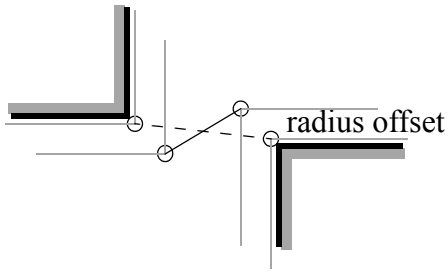


4.2 Supporting wide entities

When entities can have a radius, the corners have to be offsett from the walls [2]. This will have the effect that some edges may become valid or invalid, depending on their offsets, which in some cases invalidates the previous optimization. Figure 10 shows an example of two corners with two different offsets. The invalid edge is illustrated by a dashed line, and the valid edge by a solid line.

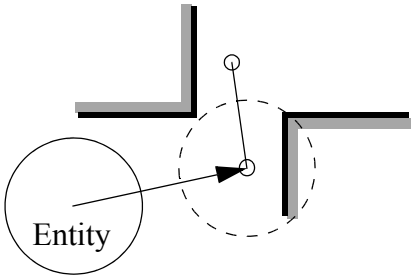
This problem could be helped by having a minimum entity width for every edge. Then the edge would still be considered invalid if the entity radius was small enough. It would make the search smaller by culling such edges at run-time, it would not help make the graph smaller though.

FIGURE 10. Edges may be valid for certain widths while invalid for other



When introducing wide entities, the graph edges will also need maximum widths. Consider the case in Figure 11. If an entity that wide would be allowed to walk along that edge it would just get stuck hitting the wall. It would never reach the corner. Each edge therefore has a maximum width limit, which may exclude the edge from the search. These maximum widths are precalculated when the graph is built.

FIGURE 11. Edges will not be able to carry entities that are too big

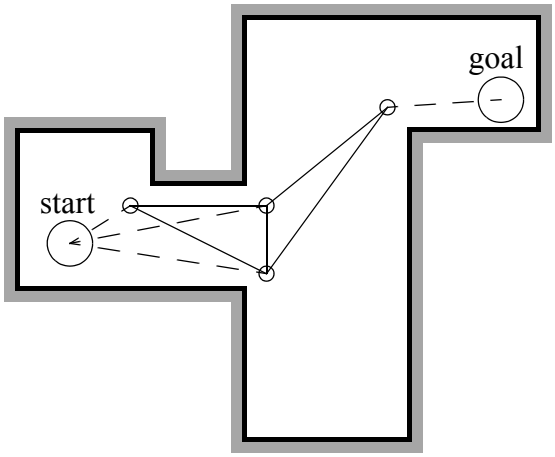


4.3 Performing the search on the mesh

Since the world is represented by a mesh, the entity for which a path should be found, can be located anywhere on it. It does not have to be snapped to a vertex in the search graph. To be able to apply a graph-search in such a case, the start and goal positions has to be linked dynamically into the graph. This is done by creating temporary edges from the start position to all vertices that are visible from the start position. The same thing is done for the goal position, as illustrated by Figure 12.

The dashed lines are temporary edges that exist only during the search.

FIGURE 12. The start and goal positions are linked into the static graph dynamically



An efficient way of finding out which vertices are visible from a certain position on the mesh is to have a PVS (Potentially Visible Set) precalculated. In this case the PVS is a list of vertices that can be visible from each polygon in the navigation mesh. It does not mean that all vertices in the PVS is visible from any position on the polygon it is associated with, but that they all may be visible. This will cut down the number of vertices to test dramatically. By using a PVS, the number of tests will not depend on the size of the graph, but only on the graph's local complexity, which will make sure the algorithm scales well. The PVS is best generated at the same time the graph itself is generated.

Visibility (or walkability) has to be tested from the start and goal positions to all the vertices in their respective PVS. All edges may not be able to represent a path wide enough for the entity, so these temporary edges also need a maximum width limit. In this case the edges are here for this entity only, so if the maximum width for an edge is

too small, it does not have to be added at all. The width of each edge is measured as they are tested for walkability.

Once the temporary edges are in place, the problem can be considered a pure graph problem and the A* search is applied to the resulting graph.

4.4 Movement on a mesh

The entity is represented as a single point with a radius, it could be considered as a pivot point for it. This point is the one that is moved along the mesh surface. Each such point knows which polygon it stands on.

When the point should be moved it is given a movement vector. The movement vector is projected down on the plane on which the point is currently located. The new movement vector will be aligned with the polygon and represent a valid movement direction. But it may not yet represent a valid complete movement, it has to be clipped against the polygon edges. If it does not hit any of edges, it is OK to move the point with the vector.

If it hits an edge, the vector is clipped against the edge, the point is moved using the clipped vector. If the edge is a wall edge (i.e. it has no polygon on the other side), the remaining movement vector (from current position of the point to the goal position) is projected onto the wall and passed to a recursive call of move.

If the edge that was hit has another polygon on the other side, the point's polygon reference is changed to that one. Then a recursive call is made to move with the remaining movement vector. That vector will then be projected onto the new polygon and make sure the point always is on the mesh surface.

5.0 Results

This section will show test runs of the implementation and describe some of the design decisions taken. The figures in this section are actual screenshots from the implementation.

5.1 Packaging the pathfinder

To make the pathfinder reusable and extendable, it needs a good interface. This pathfinder is packaged as a small state machine. This state machine is considered an agent.

The interface to an agent consists of a dynamic data structure that get passed in and out. This dynamic structure is called action. The name implies that it is only used as a command returned by an agent, but it has all the requirements for the data structure that needs be passed into the agent as well, so it is reused for that purpose.

The input action object contains information about the entity that is using the agent in its decision making. The pathfinder only asks for the entity's position in the world, but since the structure is dynamic, it is easy to ask for other information in other agents without breaking the interface.

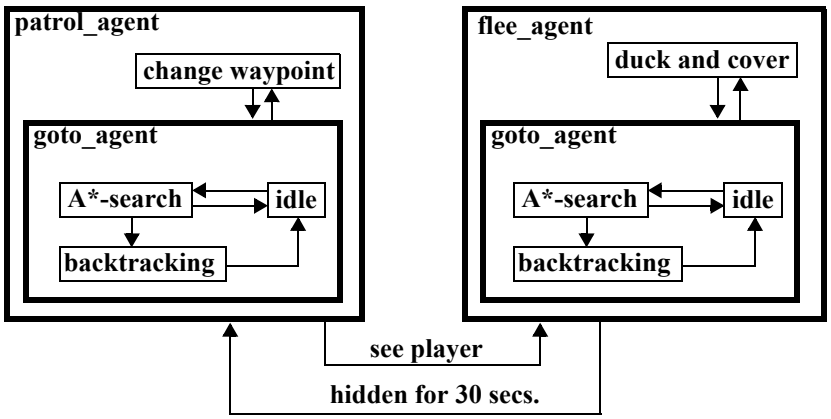
The output action object contains commands in the form of “stop”, “walk towards this point”, “goal cannot be reached“. The same goes here, new agents can invent new commands since the output action object is dynamic too.

The fact that the passed object is a dynamic structure makes it possible to implement new agents which contains the pathfinder (goto_agent) and acts as a layer, abstracting it with new high level functionality. Example of higher level agents that can be based on the goto_agent is a flee_agent, patrol_agent, follower_agent etc. All these can then be combined in an easy way to form more and more complex state machines. This concept is known as hierarchical state machines.

Figure 13 shows an example state diagram for a hierarchical state machine [9]. At top level is a state machine that can either be fleeing or patrolling along some path. If it sees the player it will change state to flee, if it has been hidden for 30 seconds it will change back to the patrol state. These high-level states, flee and patrol, are in turn implemented using a goto_agent and another state between which they control the switching. At the bottom the goto_agent takes responsibility for doing the searches and backtracking.

This way of splitting responsibility makes it easy to create higher level AI components.

FIGURE 13. Example of a hierarchical state machine



5.2 implementation

Figure 14 shows a typical path through this maze. Note that the navigation mesh is not flat, it has levels on top of each other. The path is offseted with the entity’s radius from all corners to reduce the wall collisions.

FIGURE 14. A typical path through a 3D maze

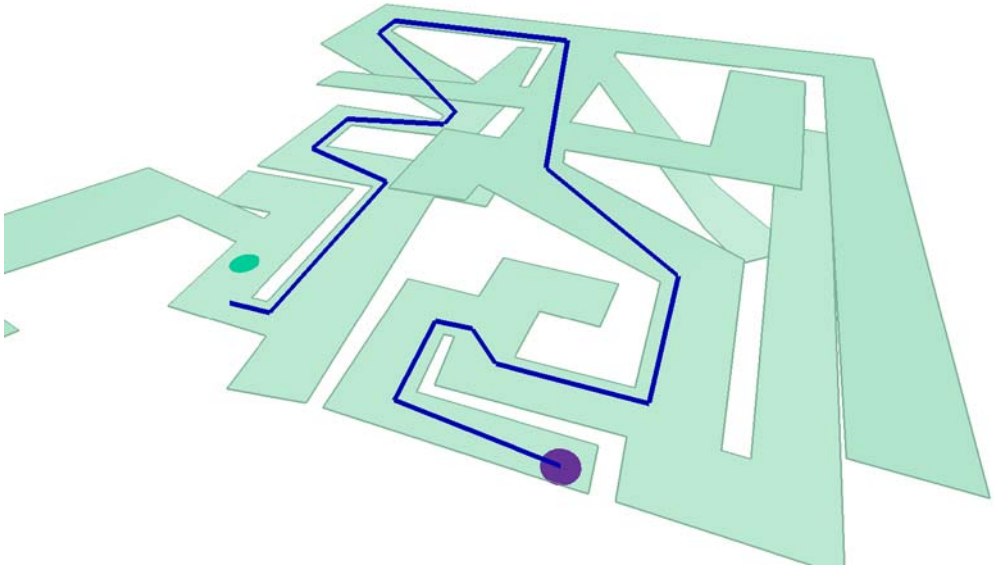
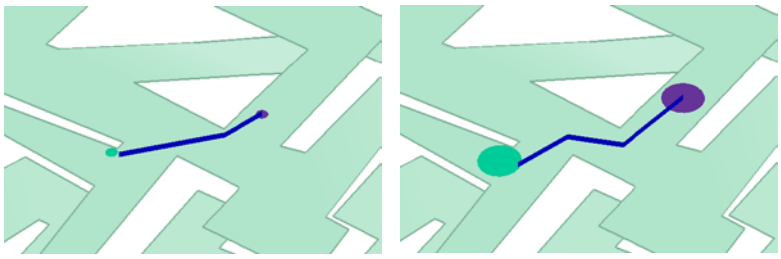


Figure 15 illustrates that the edge selection, depending on entity width, works. In Figure 15 b the entity is too wide to be allowed to walk on the edge that is a straight line. Instead it has to look for other edges and finds the shortest path available. It will give a nice effect of having the entity actually be aware of its width and bend around the corner, as shown in Figure 15 b. While in Figure 15 a, the entity is small enough to fit onto the straight line-edge, and it will take the shortest path.

FIGURE 15. A small and a large entity walking the same path



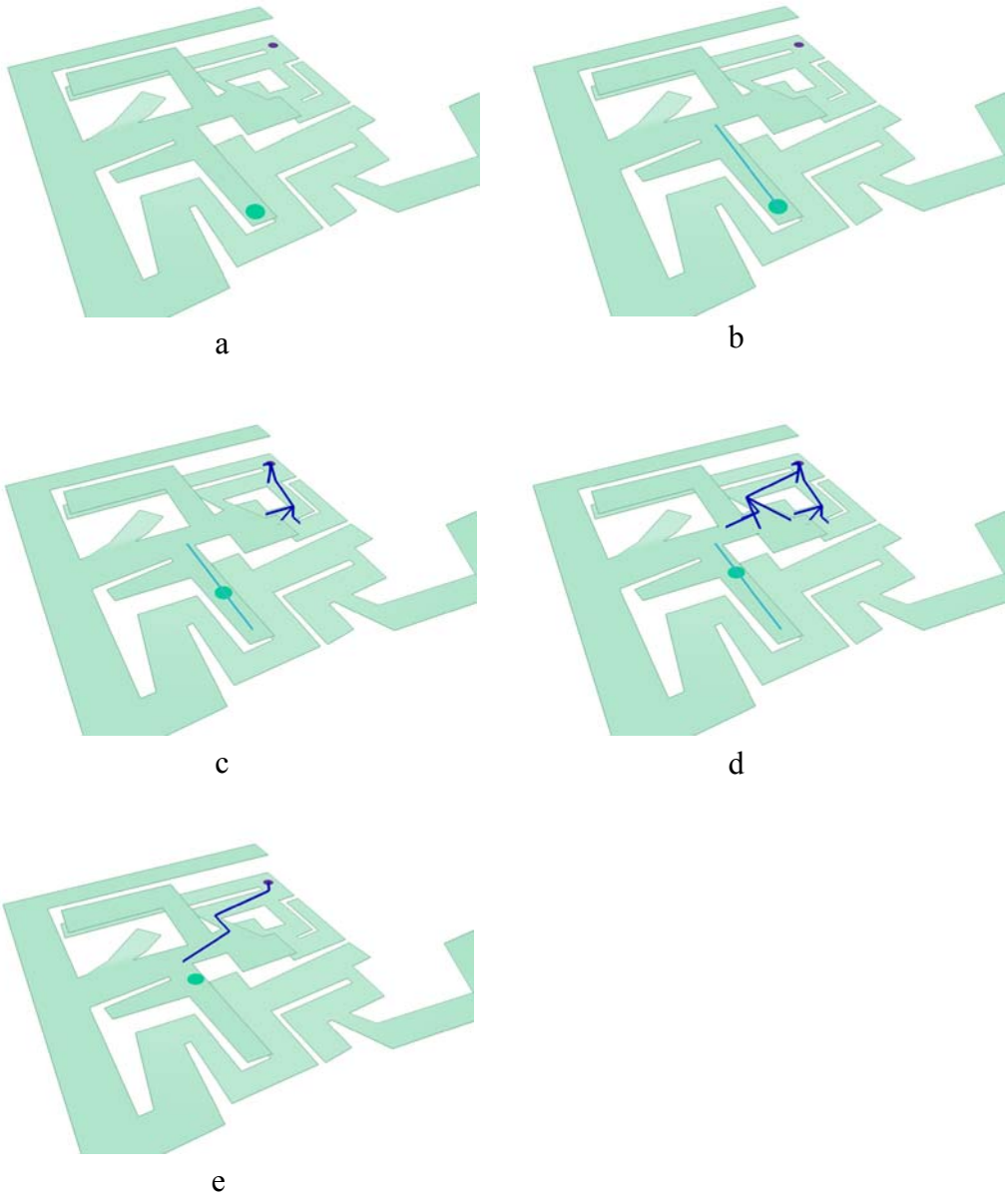
a **b**

Figure 16 shows an entity (at the bottom) trying to find its way to the marker at the top of each image. Figure 16 a is the initial setting. Figure 16 b shows the next frame, the first frame of the search. The entity has started to move in a guessed direction. While moving along the shown vector, the A*-algorithm searches for a path.

Figure 16 c shows the state a number of frames later, when the pathfinder has started to scan the adjacent graph segments. The A* heuristic makes it search in the direction of the entity. In Figure 16 d it is getting close to the entity's start position.

When the path is found, the backtracking and the joining of the guessed path will produce the final path shown in Figure 16 e.

FIGURE 16. A pathfinder run with guessing while searching

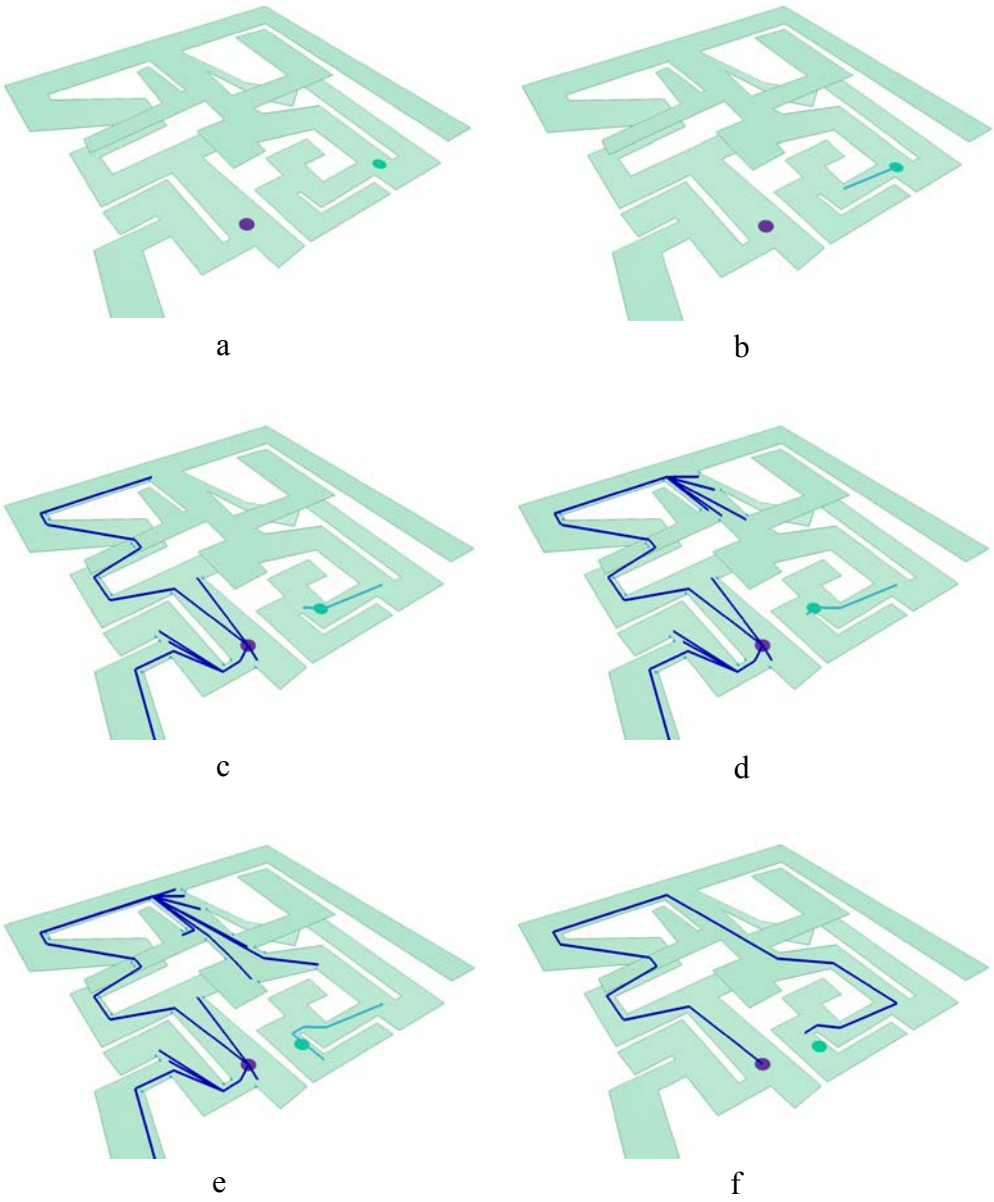


a b c d e

This can be compared with having no guessing at all, then the entity would not have started to move until the last frame in this strip. The time it would have had to wait (and the distance it moved in the guessed direction) depends of course on how much of the search tree can be visited per frame, and the movement speed of the entity. In the test above, only small portions of the search is performed per frame, resulting in a better performance increase when using guesses. It also results in worse performance when the guesses are wrong. As we can se from Figure 17, this is the behavior if the guesses are incorrect.

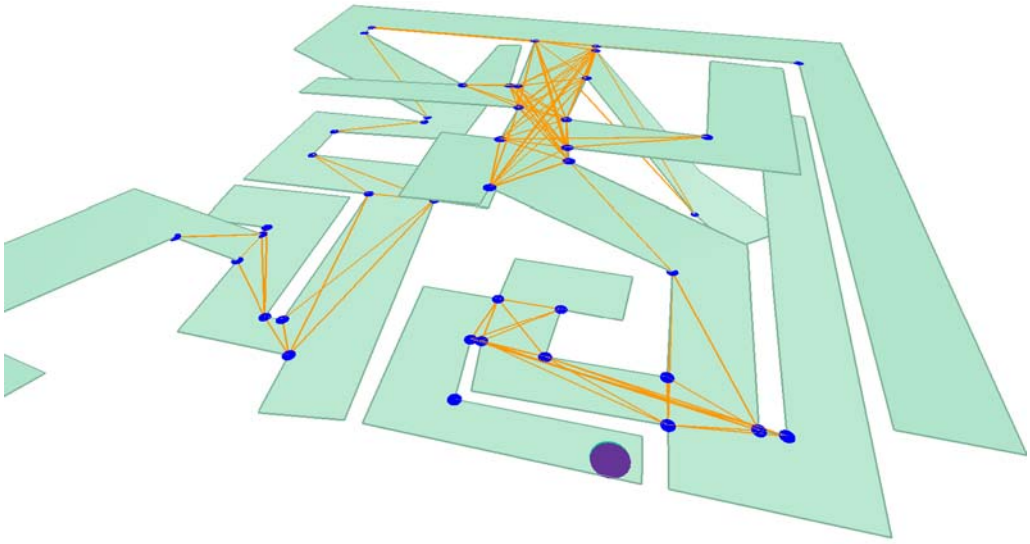
As Figure 17 b shows, the entity guesses a path in the direction of the goal, but it is not the right way. Since there is no way for the entity to know this, it continues on this path until the search is complete, in Figure 17 f. The guessed path is then joined with the found path, which makes the entity go back the way it came from.

FIGURE 17. A pathfinder run with incorrect guesses



The search graph, that is generated by applying points-of-visibility, is shown in Figure 18.

FIGURE 18. The search graph generated from the navigation mesh



Many of these edges could be optimized away if a minimum and maximum width of the entities was used. The graph is still space efficient, since most parts of the navigation mesh area do not have dense edges over it.

6.0 Conclusions

This part will discuss problems with the implementation and alternatives. It will also discuss some limitations in the algorithms and methods used, that are not necessarily implementation issues.

6.1 Floating point issues

The pervasive use of floating point arithmetic [7] have a few problems. Since the precision of a floating point value varies depending on the actual value, the epsilon (maximum allowed precision error) may only work within some boundaries of the value space. The best way to solve this would probably have been to implement the navigation mesh using fixed point, with fixed precision.

It also made it a lot harder to actually implement the navigation mesh operations, and make them stable.

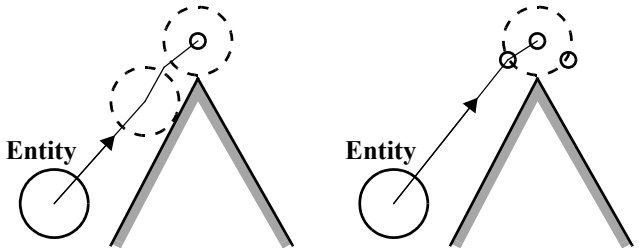
The floating point imprecision had, among other things, the implication of having to project the moving point down to the plane every now and then. Otherwise one would run the risk of having the point deviate from the mesh surface more and more due to the imprecision.

6.2 Movability limitations

One thing that may make the found path look a bit weird is the fact that at any place where two walls forms a convex corner, there will only be one graph vertex. This is fine as long as the angle between the walls is greater than or equal to $90°$. If the walls form a sharp corner, there should be two more vertices to make the path smoother. As Figure 19 shows to the left, the entity will hit the wall and slide along it until it passes it and can take a straight line to the its way point.

This problem become more and more obvious the sharper the corners get, one easy way to avoid it is simply to bevel the corner to get two vertices. The best way, though, would be to detect these cases and insert two more vertices, as showed to the right in Figure 19. Then the entity would be guaranteed not to hit the wall as long as it followed its path.

FIGURE 19. To the left the current implementation, to the right the ideal implementation



6.3 Suitable environments

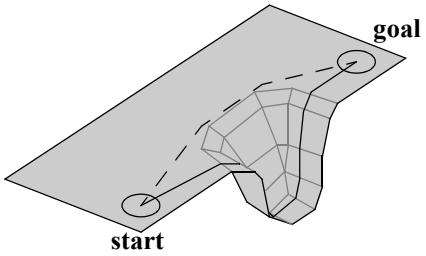
The pathfinder is not very suitable for large open areas with lots of small obstacles, like a forest. Since every obstacle will need to be surrounded with search graph vertices, and they will have to be connected to a large number of other vertices. The PVS for each polygon in the mesh will also see a lot of vertices. These effects will have a bad influence on memory footprint as well as CPU usage during searches.

This pathfinder works best with indoor environments, which usually consists of rooms and corridors. Outdoor environments could be suitable too, if they are relatively simple, and do not contain a lot of small obstacles.

6.4 Path optimality

The implementation cannot guarantee that the shortest path is taken when it walks on arbitrary meshes. It can guarantee that the shortest path is found in the search graph, but the search graph is built from corners and not from all the geometry of a navigation mesh. Figure 20 illustrates an example where this pathfinder will take the longer path instead of the shortest (dashed).

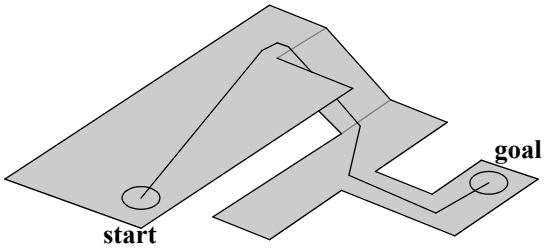
FIGURE 20. The shortest path (dashed) compared with the path found by this pathfinder



The reason why this happens is because in the case above, there is no search graph, because an entity can use a projected straight line to go from any point to any other point on the mesh. Therefore there is no graph, since there are no concave corners. In the case illustrated by Figure 20, it will just try to walk in a straight line, and project onto the actual mesh. That straight line will succeed, and the entity will not know anything about the pit, since there is no graph to search.

This is usually not a problem though. In the usual case, in which this pathfinder is supposed to work, it will find the shortest path. Navigation meshes that works well should have relatively flat areas between walls. There are no problems with ramps and obstacles, it is just that pits and bumps will not be detected by the search graph. Figure 21 shows a typical kind of navigation mesh.

FIGURE 21. The shortest path is found in the common case



7.0 References

[1] Thomas Young. *Optimizing Points-of-Visibility Pathfinding*
Game programming gems 2, Charles river media, 2001, ISBN 1-58450-054-9

[2] Thomas Young. *Expanded Geometry for Points-of-Visibility Pathfinding*
Game programming gems 2, Charles river media, 2001, ISBN 1-58450-054-9

[3] Steve Rabin. *A* Speed Optimizations*
Game programming gems, Charles river media, 2000, ISBN 1-58450-049-2

[4] Greg Snook. *Simplified 3D Movement and Pathfinding Using Navigation Meshes*
Game programming gems, Charles river media, 2000, ISBN 1-58450-049-2

[5] ID software: *Quake*, http://www.idsoftware.com/games/quake/quake/
(2004-01-07)

[6] Pathengine: *Pathfinding middleware*, http://www.pathengine.com/
(2004-01-07)

[7] IEEE standards organization:
IEEE 754: Standard for Binary Floating-Point Arithmetic,
http://grouper.ieee.org/groups/754/ (2004-01-07)

[8] Russel Norvig. *Artificial Intelligence A Modern Approach,*
Prentice Hall, 1995, ISBN 0-13-103805-2

[9] EventHelix.com: *Hierarchical State Machine*
http://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm
(2004-02-15)