Mobile Games Pathfinding

Jarmo Kauko[*]
[*]Nokia Research Center
Visiokatu 1, 33720, Tampere, Finland
jarmo.kauko@nokia.com

Ville-Veikko Mattila[†]
[†]Nokia Research Center
Visiokatu 1, 33720, Tampere, Finland
ville-veikko.mattila@nokia.com

Abstract

Pathfinding is a well-studied problem in computer games. The increasing complexity of games creates constantly new requirements for pathfinding systems. Meeting these requirements is especially hard for mobile devices due to their limited memory and processing power. This paper describes a system for fast and lightweight pathfinding in complex three-dimensional and dynamic environments, designed for mobile games. The system is based on visibility graphs and uses incremental heuristic searching. This paper also describes a novel way of combining these techniques. The experimental results on mobile devices show that the system is efficient and applicable for mobile games.

1 Introduction

In the recent years, the increased sophistication of graphics in video games has been yielding diminishing return in players' game experience. Stunning graphics is simply no longer the main driver for game sales, as the focus has been turning more towards gameplay and artificial intelligence (AI). AI is used in games to drive game characters, such as enemies and own troops that need to give an illusion of intelligent behavior; otherwise the game experience is ruined. Here, the objective of game AI is not to find an optimal solution in any given situation, but to make the game fun to play.

Games on mobile devices are currently using very primitive AI due to the devices' limited computation power. Thus, mobile games are often limited to simple shooting and racing games. Availability of AI could expand the range of games and genres that are feasible on mobile devices, considering, e.g., strategy games. In the long term, advanced AI could increase the popularity and acceptance of gaming among people, particularly adults, who are currently not playing games.

The development budgets and time of mobile games are typically strongly limited. Thus, game AI developers rarely have a chance to work on higher-level AI as much of the time is spent in struggling with low-level functionalities, e.g., in pathfinding. In the same time, retailers and network operators have raised the overall quality requirements of Java games, focusing on more complex 3D games. A clear qualitative improvement for AI techniques in mobile games will be dependent on appropriate interfaces to AI middleware to allow game AI developers to focus on higher-level creative AI tasks instead of worrying about low-level procedures. Naturally, such an interface should be generic enough to be applicable to a wide scope of mobile games. In Java games, an interface to an AI engine, running in the platform of mobile devices, could allow the adoption of computationally complex AI methods. However, this would require the standardization of an AI API in Java Community Process (JCP) to guarantee compatibility. Standard interfaces could also provide a foundation for the design of a dedicated hardware for game AI.

In this paper, we describe fundamental parts of our pathfinding system designed for mobile games. We start by introducing the underlying search space representation. We then continue with a short introduction to incremental heuristic search algorithms. After that, we describe the algorithm used in our pathfinding system. Finally, some experimental results evaluating the performance of the system are shown.

2 Visibility graphs

Visibility graphs are well studied and applied to various problems, such as urban planning and mobile robotics. For pathfinding, they can provide compact and accurate representations of two-dimensional surfaces containing polygonal obstacles. In order to use visibility graphs for our purposes, they had to be extended to operate on three-

dimensional surfaces. Additionally, we wanted to be able to assign arbitrary cost factors to different parts of surfaces, instead of having only traversable and non-traversable parts.

Visibility graphs contain nodes at each convex corner of obstacles. Here, obstacles are not required to be non-traversable, but to have a higher cost factor than surrounding areas. The nodes that are visible to each other are connected. Due to the additional requirements, existing construction methods based on visibility tests cannot be applied to our visibility graphs. The traversability cost needs to be determined between every pair of nodes, resulting in $O(n^2)$ construction time in a graph containing n nodes. The traversability cost is computed by projecting the line between two nodes onto the surface and integrating the cost factors of the surface over the distance. The resulting graph can be used to find paths using shortest path algorithms such as the A* algorithm (Russel and Norvig 1995). However, since the start and end locations of a search can be anywhere on the surface, new start and end nodes needs to be generated. This requires computing edge costs between the new nodes and the existing nodes of the visibility graph. Since computing the edge costs is an expensive operation, the size of a graph dramatically decreases the performance of graph construction and searching. To overcome this problem, we allowed combining several smaller graphs to represent large environments. Hierarchical pathfinding uses similar approach to increase the searching performance at the expense of optimality (Botea, Müller and Schaeffer 2004), (Sturtevant and Buro 2005). Sub-graphs in hierarchical pathfinding are usually required to be connected, i.e., to have a valid path between each pair of vertices. In our system, sub-graphs are only used to determine sets of nodes that are potentially visible to each other. Edges between nodes in different graphs have to be defined explicitly.

2.1 Properties of an agent

One of the requirements for the pathfinding system was the ability to support various types of agents using a single graph. Using a separate graph for each type requires more memory, limits the variety of different agents, and requires more processing to adapt to dynamic changes in the game world. This was achieved by using cost vectors instead of scalar cost values. The components of the cost vectors present different types or sources of costs. Each agent has a property vector defining the influences of the components. Similarly, a movement constraint vector was used to describe strict limits, such as the maximum radius or a security access level.

A fundamental property in pathfinding is the size of an agent. Smaller agents should be able to use narrow passages and move closer to walls than larger agents. A traditional solution to handle the size of an agent is to use a free configuration space, i.e., expand the walls of the obstacles to match the shape of an agent (Lozano-Pérez and Wesley 1979). However, the structure of visibility graphs also provides interesting properties for supporting various sizes of agents using a single graph. The following example explains the intuition behind this (see Figure 1 a): Let E_{AB} be an edge between nodes N_A and N_B. Let O be a convex corner of an obstacle next to C_{AB}. Due to the presence of O, wide agents may not be able to use the edge E_{AB}. However, a node located to O, N_O, provides an alternative route through edges E_{AO} and E_{OB}. This way the visibility graph remains valid for different sizes of agents.

As the nodes are next to the obstacles, wide agents need to be displaced away from the walls. This was achieved by computing an offset vector for each corner. The offset vectors were also used to expand the geometry to compute the maximum radius of an agent for each edge. To estimate the maximum radius, only the corners that are visible from the end points of the edge need to be considered – if a corner is not visible, the corner that is blocking the visibility is closer to the edge. If an agent moves alongside an obstacle, at each point it needs to be displaced away from the obstacle by the radius of the agent. Thus, the path around a corner of an obstacle should be a smooth curve. In order to approximate this behavior, two nodes were used for sharp corners of obstacles. The both nodes have the same location but different offset vectors, as illustrated in Figure 1 b.

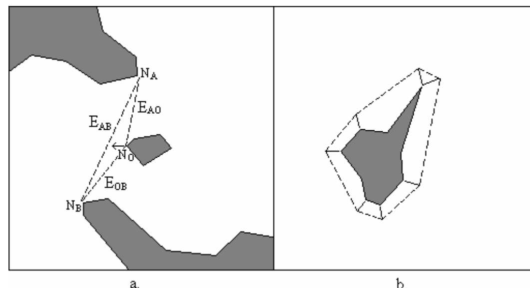

Figure 1: Examples demonstrating the construction of visibility graphs.

3 Incremental heuristic searching

Game worlds are often dynamic in a sense that they may change during the game play. These changes also occur, if an agent has incomplete or inaccurate knowledge of the game world and detects new information of it. Pathfinding needs to consider these changes and replan paths when necessary. Incremental search methods can provide fast replanning

by using data from previous searches. They are successfully used, e.g., in robotics for navigation in partially known environments. However, they are not commonly used in games.

Games usually contain multiple moving agents and other highly dynamic factors. Replanning to adapt to these changes is infeasible, and usually not even reasonable, since the dynamic objects are likely to keep changing. Thus, games generally use local planning or steering algorithms to avoid these obstacles. Local obstacle avoidance requires an agent to deviate from the original path, which again is prone to the local minimum problem (Russell and Norvig 1995). Since returning to the original path may be troublesome, games often have to replan the whole path from scratch. However, incremental searching methods can be used to repair the original path whenever the agent drifts too far away from the path. This can reduce the frequent need for pathfinding substantially. Therefore, incremental searching is particularly important for pathfinding in games.

In order to use incremental searching, search data, including the priority queue and the cost values, has to be maintained for each search. Particularly, the required amount of memory in large grid-based graphs may be unacceptable for games. Since visibility graphs contain nodes only where a path turns, they are usually small and thus well suitable for incremental searching. However, using an incremental searching algorithm on a visibility graph is not straightforward, since the start and end nodes are generated. When agent moves along the path, the edge costs from the start node become invalid and need to be re-computed. As determining the edge costs is an expensive operation, replanning becomes very inefficient. The same problem applies to time-sliced pathfinding, where the searching process is divided over multiple frames, and the agent may move while searching (Higgins 2002).

A number of incremental algorithms exists for path replanning. The Focussed Dynamic A* (D*) (Stentz 1995) and D*Lite (Koenig & Likhachev 2002) algorithms are currently the most popular, because they efficiently use heuristics to determine, which parts of the path needs to be repaired. The method described in this paper is built on the D*Lite algorithm, but is applicable to any incremental heuristic search algorithm.

3.1 Lifelong planning A* and D*Lite

The D*Lite algorithm is based on Lifelong planning A* (LPA*) (Koenig, Likhachev, and Furcy 2004) that is an incremental version of the A* algorithm. LPA* reuses information from previous searches to adapt to dynamic changes in a graph. It also uses heuristics to focus the search and to determine which nodes need to be examined. If no information from previous searches is available, LPA* is similar to A* that breaks ties among nodes with the same f-value in favor of smaller g-values. Furthermore, it shares many interesting properties of the A* algorithm. It is relatively easy to understand, implement and extend.

Like A*, LPA* uses *g(s)* to indicate the cost from the start node and *h(s)* to approximate the cost to the end node for each node s. Since graph is allowed to change, *g(s)* is only an estimate of the start cost. To update the changed start costs, LPA* maintains *rhs(s)*, which is always one step ahead of *g(s)*. Formally,

$$rhs(s) = \begin{cases} 0 & \text{if s is the start node} \\ \min_{s' \in Pred(s)}(g(s') + c(s',s)) & \text{otherwise} \end{cases}$$

Here, Pred(s) is the set of predecessors of s and *c*(s',s) is the edge cost from s' to s. A node is said to be locally consistent, if its rhs-value equals to its g-value. Otherwise, the node is locally inconsistent. A node, s, is locally overconsistent if g(s) > rhs(s), and locally underconsistent if g(s) < rhs(s). LPA* maintains a priority queue U, containing all inconsistent nodes. U is sorted by a key, which consists of primary and secondary components:

$$k(s) = \begin{bmatrix} \min(g(s), rhs(s)) + h(s) \\ \min(g(s), rhs(s)) \end{bmatrix}$$

The LPA* algorithm starts by initializing the g-value of the start node to zero. All other g-values and rhs-values are initialized to infinity. Therefore, at the beginning, U contains only the start node. LPA* processes the nodes from U very similarly to how A* operates with the OPEN set. The algorithm makes the nodes locally consistent by removing the node with the smallest key from U and updating it and its successor nodes. Underconsistent nodes are processed slightly differently from overconsistent nodes. LPA* continues to process vertices until the goal node is locally consistent and has a smaller key than or the same key as the best node in U. If U becomes empty and the cost to the goal is infinity, there is no path available. The detailed description and analysis of the LPA* algorithm can be found in (Koenig, Likhachev, and Furcy 2004).

3.2 Our algorithm

The D*Lite algorithm uses heuristics to determine the searching order, i.e., the order, in which the nodes are processed. Our algorithm uses the same heuristics to determine when to compute the edge costs from the start node. As the D*Lite algorithm searches towards the start node, the edge costs from the start node are only needed to determine the total cost of a path. Before expanding a node, edge costs from the start node to all nodes with lower priority keys need to be determined. Otherwise, the nodes are not guaranteed to be processed in the right order,

```
procedure CalculateKey(s)
   01    return [min(g(s), rhs(s)) + h(start, s); min(g(s), rhs(s))];
procedure Initialize()
   02    U = V = CLOSED = Ø;
   03    for (all s ∈ S) rhs(s) = g(s) = ∞;
   04    rhs(goal) = 0;
   05    U.Insert(goal, [h(start, goal); 0]);
procedure UpdateVertex(u)
   06    if (g(u) ≠ rhs(u) AND u ∈ U) U.Update(u, CalculateKey(u));
   07    else if (g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, CalculateKey(u));
   08    else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);
procedure ComputeShortestPath()
   09    while (U.TopKey() < CalculateKey(start) OR rhs(start) > g(start))
   10       if (V.TopKey() < U.TopKey())
   11          u = V.Top();
   12          Compute the edge cost c(start, u)
   13       else
   14          u = U.Top();
   15          k_old = U.TopKey();
   16          k_new = CalculateKey(u);
   17          if (k_old < k_new)
   18             U.Update(u, k_new);
   19          else if (g(u) > rhs(u))
   20             g(u) = rhs(u);
   21             U.Remove(u);
   22             Compute the edge cost c(start, u)
   23             for (all s ∈ Pred(u) ∪ {start})
   24                if (s ≠ goal) rhs(s) = min(rhs(s), c(s, u) + g(u));
   25                UpdateVertex(s);
   26             CLOSED.Insert(u);
   27          else
   28             g_old = g(u);
   29             g(u) = ∞;
   30             for (all s ∈ Pred(u) ∪ {u})
   31                if (rhs(s) = c(s, u) + g_old)
   32                   if (s ≠ goal) rhs(s) = min s' ∈ Succ(s) (c(s, s') + g(s'));
   33                UpdateVertex(s);
procedure UpdatePosition()
   34    if (start ∈ U) U.Remove(start);
   35    else if (start ∈ CLOSED) CLOSED.Remove(start);
   36    rhs(start) = g(start) = ∞;
   37    for (all s ∈ U) U.Update(s, CalculateKey(s));
   38    for (all s ∈ V) V.Remove(s);
   39    for (all s ∈ CLOSED)
   40       c(start, s) = ∞;
   41       V.Insert(s, CalculateKey(s));
procedure Main()
   42    Initialize();
   43    ComputeShortestPath();
   44    next = arg min s' ∈ Succ(start) (c(start, s') + g(s'));
   45    while (not reached goal)
   46       if (g(start) = ∞) then there is no known path
   47       Move towards next
   48       if (reached next) next = arg min s' ∈ Succ(next) (c(start, s') + g(s'));
   49       Scan graph for changed edge costs
   50       if (any edge costs changed)
   51          UpdatePosition();
   52          for (all directed edges (u, v) with changed edge costs)
   53             c_old = c(u, v);
   54             Update the edge cost c(u, v);
   55             if (c_old > c(u, v))
   56                if (u ≠ goal) rhs(u) = min(rhs(u), c(u, v) + g(v));
   57             else if (rhs(u) = c_old + g(v))
   58                if (u ≠ goal) rhs(u) = min s' ∈ Succ(u) (c(u, s') + g(s'));
   59             UpdateVertex(u);
   60          ComputeShortestPath();
   61          next = arg min s' ∈ Succ(start) (c(start, s') + g(s'));
```

Figure 2: The algorithm for incremental searching in visibility graphs.

and the resulting path may not be optimal. If the agent's location has changed and the path needs to be replanned, edge costs from the new start node need to be computed to satisfy this condition. The first attempt was to compute edge costs to all processed nodes. However, if the agent has traveled a long way, this method is likely to compute more edge costs than what is necessary. If a valid path has been found, nodes with higher priority than the start node do not need to be considered. Therefore, the priority of the nodes can be directly used to determine, when to compute missing edge costs from the start node to the processed nodes.

The resulting algorithm is presented in Figure 2. The algorithm uses additional priority queue, V, to store all the nodes that potentially need to be updated for the current start node. It also uses CLOSED list to store all processed nodes, which is required for initializing V. Unlike D*Lite, our algorithm updates all h-values each time the agent moves in order to minimize the computation of the edge costs. This is acceptable since the number of nodes in visibility graphs is generally very low. This is done in UpdatePosition procedure (lines 34-41). Edge costs from the start node to nodes in V are computed in lines 10-12. The Main procedures of D*Lite and our algorithm are different, since our algorithm is applied to continuous environments. The algorithm was implemented to support time-sliced pathfinding, which was very straightforward as the algorithm supports agent movement during searching.

4 Experimental results

Two versions of the pathfinding system were implemented using Java and C. The implementations were algorithmically identical and no low-level optimizations were implemented. Performance of the implementations was compared to evaluate the benefits of having a pathfinding system for mobile games integrated to the platform. Two experiments were arranged to evaluate the system. The purpose of the first experiment was to measure the general performance of the system, compare the different implementations, and evaluate the scalability of the system to large graphs. The second experiment focused on algorithmic comparison of path replanning in visibility graphs.

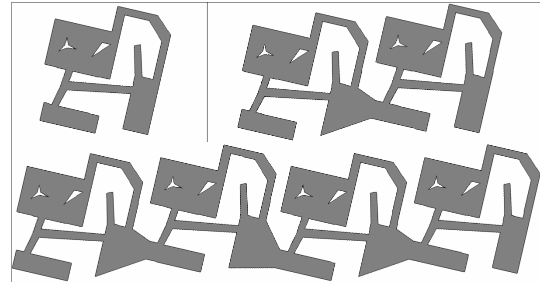

Figure 3: The test maps for general performance evaluation

4.1 General performance

The feasibility of the pathfinding system was evaluated by testing the performance of all critical operations. These operations include construction of a graph, dynamic modification of a graph and searching. The dynamic modification of a graph was allowed by adding arbitrary polygonal obstacles on

the top of the movement surface. This requires generating new nodes to the convex corners of the obstacle, increasing the edge costs and updating the maximum agent size of nearby edges. Searching of a path was divided into two tests; generating the goal node and actual searching, which also determines edge costs from the start node. All of the operations were tested on three game maps, illustrated in Figure 3. Each map is represented using a single visibility graph to test the scalability without the usage of sub-graphs. The first map is relatively small, containing only 33 nodes. The second map is approximately two times as large as the first map, containing 69 nodes. Finally, the third map is approximately two times as large as the second map and contains 141 nodes.

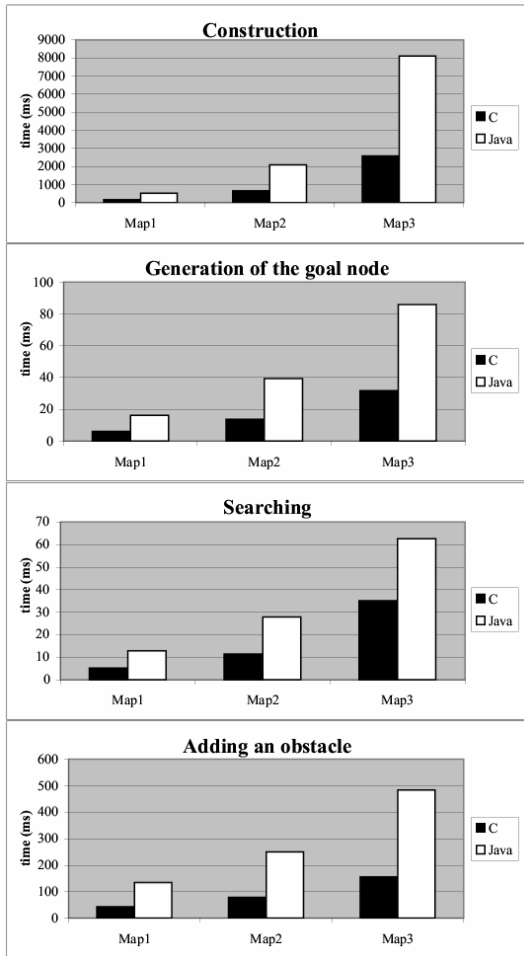

Figure 4: The results of the general performance evaluation.

The dynamic modification of a graph was tested by uniformly selecting ten locations from each map. A rectangular obstacle was added to each location. The searching was tested similarly by uniformly selecting ten locations from each map, and then finding a path between each pair of locations. This way the test contained more middle length paths, and less very short or long paths. The tests were performed on Nokia 6230i. The results of the tests are presented in Figure 4.

4.2 Path replanning test

Path replanning was tested to evaluate the benefits of incremental replanning in realistic in-game situations. For analytic and experimental results of the performance of the D*Lite algorithm, see (Koenig & Likhachev 02). As discussed before, the usage of local planning and steering algorithms often causes an agent to deviate from the path, therefore, requiring the whole path to be replanned from scratch. This experiment tests how different versions of our algorithm perform in these situations. The test map consists of three continents that are connected by two elevators as shown in Figure 5. The map is modeled using four visibility graphs; the main continent in the middle is divided into two graphs, and both side continents are modeled with one graph. The total number of nodes in the map is 68.

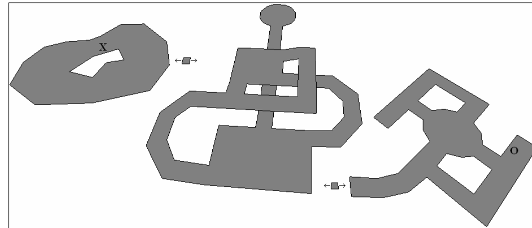

Figure 5: The test map for incremental searching tests. The map consists of three continents, connected by elevators.

Firstly, a path across the whole map was generated. In Figure 5, the start node of the path is marked with **o** at the right end of the map and the end node with **x** at the left end. Fifteen locations were selected along the path from the first three graphs. Each location was displaced away from the path and the path was replanned using the location as the new start node. The first algorithm replanned the whole path from scratch and computed the edge costs from the new start node to all nodes in that graph. The second algorithm replanned the new path from scratch as well, but used heuristics to determine the edge costs. The third algorithm was a basic incremental algorithm that computed edge costs from the new start node to all expanded nodes in the initial search. The final algorithm was the improved incremental algorithm as presented in Figure 2. The generation of the end node was excluded from the test, since the same end node can be used for recurrent searches. The test was performed on Nokia

6680 using the Java implementation. The results of the test are illustrated in Figure 6.

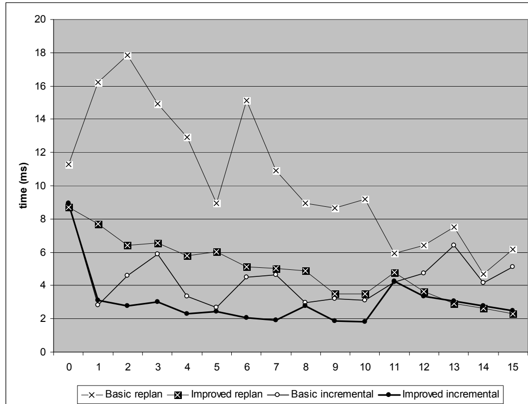

Figure 6: The results of path replanning test. The first column contains results of the initial search. The following columns (1-15) present the results of replanning the path from selected locations along the initial path.

5 Discussion

Visibility graphs allow representing complex three-dimensional environments using small number of nodes. However, determining edge costs between the nodes is computationally very expensive. As the results of the general performance evaluation indicate, the presented pathfinding system can handle small graphs relatively well in real time. On the other hand, very large graphs require much more processing time. However, a careful division into sub-graphs allows the system to scale well to large and complex environments. The system was also implemented to support time-sliced pathfinding, thus allowing multiple paths to be searched simultaneously without compromising the frame rate. The C implementation of the system proved to be remarkably more efficient than the Java implementation.

The incremental searching enables an efficient path replanning, also allowing global pathfinding and local steering algorithms to be combined more efficiently. As shown by the incremental searching test, the final algorithm replanned efficiently even for long paths. For very short paths, replanning from scratch can actually be slightly faster than incremental searching, since the incremental searching requires that the h-values are updated. In complex multi-agent environments, paths are often long and have to be replanned repetitively. Therefore, the usage of the incremental replanning algorithm can reduce the total searching load substantially.

One of the major bottlenecks was the math required for generating nodes and determining edge costs. Floating-point math was used to perform these calculations. Since the utilized mobile devices did not have a floating-point processing unit (FPU), the performance of floating-point math was a critical issue.

6 Conclusions

This paper described a pathfinding system designed particularly for mobile games. The main objective was to study the benefits of having such a solution integrated to a mobile platform to improve the performance of Java games. The system uses visibility graph-based approach to represent movement surfaces in two- or three-dimensional game worlds. It was designed to support various types of agents using a single graph, thus keeping the memory consumption low. This would also allow a fast adaptation to the dynamic changes in a game world, not limiting the variety of different agents. Large graphs can be divided into sub-graphs to guarantee scalability. As the experimental results show, the system was able to perform complex pathfinding tasks in real-time.

The pathfinding system uses incremental heuristic searching, which allows a fast path replanning. This is particularly important to combine global pathfinding and local obstacle avoidance behaviors. The paper presented a novel technique that allows incremental and time-sliced pathfinding on visibility graphs. The resulting algorithm was compared to less advanced replanning methods. The experimental results show that the presented algorithm allows fast path replanning regardless of the length of the path.

The C implementation of the system proved significantly more efficient than the corresponding Java implementation, indicating that Java games would remarkably benefit from AI solutions integrated to the platform. Obviously, this would also cut the development time and costs of Java games, as well as allow developers to concentrate on more advanced AI.

Acknowledgements

We would like to thank our colleagues in Nokia Research Center Multimedia Technologies laboratory for constructive discussion on the subject.

References

Botea, A., Müller,M., and Schaeffer, J. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):7–28, 2004

Higgins, D. Pathfinding Design Architecture. *AI Game Programming Wisdom*, pp. 122-132, 2002.

Koenig, S., and Likhachev, M. D* Lite. *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 476-483, 2002.

Koenig, S., Likhachev, M., and Furcy, D. Lifelong Planning A*. *Artificial Intelligence* 155:93-146, 2004.

Lozano-Pérez, T., and Wesley, M. An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Communications of the ACM* 22:560-570, 1979.

Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 1995.

Stentz, A. The Focussed D* Algorithm for Real-Time Replanning. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1652-1659, 1995.

Sturtevant, N., and Buro, M. Partial Pathfining Using Map Abstraction and Refinement. *Proceedings of AAAI*, pp. 47–52, 2005.