Efficient Multi-Agent Path Planning

Okan Arikan
University of California at Berkeley

Stephen Chenney
University of Wisconsin at Madison

D.A. Forsyth
University of California at Berkeley

Abstract. Animating goal-driven agents in an environment with obstacles is a time consuming process, particularly when the number of agents is large. In this paper, we introduce an efficient algorithm that creates path plans for objects that move between user defined goal points and avoids collisions. In addition, the system allows "culling" of some of the computation for invisible agents: agents are accurately simulated only if they are visible to the user while the invisible objects are approximated probabilistically. The approximations ensure that the agent's behaviors match those that would occur had they been fully simulated, and result in significant speedups over running the accurate simulation for all agents.

Keywords: path planning, virtual agents, proxy simulations, simulation level of detail

1 Introduction

Simulations of thousands of interacting agents are in demand for entertainment applications (gaming [7] and film [16]), training simulations, and scientific visualization [2]. The *path planning* problem is to determine where each agent should move on each frame of the animation. Path planning algorithms are responsible for maintaining many essential aspects of plausible agent behavior, including collision avoidance and goal satisfaction. Path planning also consumes a significant part of the computation time for many simulations, particularly in highly dynamic environments where most of the agents are moving at the same time with varying goals. This work cannot easily be avoided, because of its high impact on visual quality and the outcome of the simulation (for instance, agents must show evidence of delays due to obstacle avoidance).

Path planning is typically performed on one agent at a time, and broken into at least two tasks. The first is concerned with global path planning, and identifies the ideal path from the agent's current position to its target location. Global path planning typically ignores local transient obstacles, such as other moving objects. The second, local task is concerned with moving the agent along the planned path at a reasonable speed and taking into account the obstacles ignored by the global path plan.

A key observation is that the local task is of less importance if the agent concerned is not in view. For instance, if the local simulation is primarily concerned with avoiding agent-agent collisions, and the viewer cannot see such collisions, then there is little point in incurring the computational cost to avoid them. The situation is analogous to geometry culling in the rendering sense, where invisible geometry is not considered by

the renderer.

Local out-of-view interactions do, however, influence the overall simulation even if they occur out of view. For instance, an agent moving through a crowd would take longer to get somewhere than an agent moving alone. The difference in speed is due entirely to local interactions between agents. For this reason it is not sufficient to just ignore out-of-view motion. We can, however, approximate it. The viewer does not actually see the interactions, so the simulation is acceptable as long as their cumulative effect is still perceived by the viewer (for instance, agents still take longer to get places in crowds).

In this paper we present an efficient path planning methodology designed for large numbers of agents moving simultaneously around fixed obstacles and around each other. Our approach consists of a simulator that is designed for agents that the viewer can see, and a modified simulator, the *proxy simulator*, that operates on out-of-view agents. The proxy simulator achieves quantitatively similar simulation outcomes at a tiny fraction of the cost of complete, accurate path planning without significantly impacting the viewer's experience of the environment.

Section 2 looks briefly at related work on both path planning and efficient simulation. We then describe the accurate path planner (section 4) and the proxy planner (section 5).

2 Related Work

Early work on global path planning for a single agent phrased the problem as constrained optimization by defining a cost function favoring shorter paths and adding constraints to avoid the obstacles. Gradient descent methods may be applied to find the minimum cost path [5], but are subject to local minima problems and do not necessarily reach the goal state. Barraquand, Latombe, Overmars and Kavraki [1, 13, 9, 15] proposed temporary random navigation as a way to recover from the local minima. Unfortunately, most randomized or optimization driven path planning algorithms can be expensive in particular environments, and may even fail to reach the goal state.

Previous algorithms for multi-agent path planning have also been framed as optimization problems. Constraint forces are used to prevent the objects from running into each other [12]. Clearly these approaches are at least as expensive and prone to local minima as the single agent case. See [10] for an overview of probabilistic path planning.

The idea of landmarks survives in many computer games, where A^* search is almost universally used to find paths among the landmarks. The performance and quality of path planning on landmark based systems depends on the placement of the landmarks which is usually done by the level designer. Although A^* and its variants guarantee to give the optimal path, they can be too slow for large environments, particularly those of the size we address in this paper.

2D computational geometry algorithms for path planning use a polygonal description of the environment and operate on the visibility graph of the vertices of the obstacles (see [11] for a review). For our work, we adopt this type of planning as it does not involve discretizing the environment with landmarks and does not suffer from local minima issues.

Finally, the authors [4] have described *simulation proxies*: simulations that are intended to operate on out-of-view objects to significantly reduce the overall cost of large simulations. This paper focuses on a specific aspect of that work: the design of a proxy simulator for large scale path planning. An overview of cheap simulation methods is provided in [4].

3 System Overview

Our system is targeted to the path planning problems found in real time computer strategy games or similar applications. Typically, a virtual battle is modeled by simulating the behaviors of individual objects that move on a 2D terrain or attack other objects on the user's orders. In such games path planning and collision avoidance between objects constitutes the bulk of the computational workload of the simulator, due to both the large numbers of objects and the typically high complexity of path planning and collision detection algorithms. In most battle games the user is prevented from seeing the entire battlefield at any one time, so an approximate, or *proxy*, simulator can replace accurate path planning and collision avoidance for invisible objects. We describe such a proxy simulation that allows us to simulate very large numbers of objects moving on a plane.

The path planning problem in this situation has the following properties:

- The world is 2D and consists of free space and *fixed* obstacles defined by closed simple polygons.
- The agents in the world have a fixed size and are ordered by a user to move at constant speed to a target destination position. Agents without orders stay in the same place, and are said to be *stopped*. For our experiments, the user is replaced by a process that selects objects and gives them orders at random.
- Objects should move along the shortest path to their target. They may deviate from this path to avoid collisions with fixed, stopped and other dynamic obstacles.
- The viewer can only see a limited region of the world.

A simulator in this world moves the agents from frame to frame according to the above rules. The aim of our work is to perform this simulation as cheaply as possible, subject to the constraint that *the viewer experiences reasonable behavior at all times*. In this case reasonable behavior means that objects take times to reach their destinations that are consistent with the motion of all the other objects in the world, the restrictions on their motion, and their orders. We are not concerned that the moment-to-moment motion is correct *outside* the visible region of the world. This idea is explored further in section 5.

4 Full Path Planning

The path planning and collision avoidance problem is solved for an individual agent when it receives orders to move from its current location to a new destination point. The path is constructed in three stages — one stage for each type of obstacle to avoid.

1. When the order is first received, a path plan is constructed around all the fixed obstacles in the world. This stage makes extensive use of precomputed data structures, as described below. Precomputation is appropriate here because the fixed obstacles do not change through the simulation.
2. The agent next plans around agents that are stopped in its projected path. This step is also performed when the order is given, but does not rely on precomputation because it is not known ahead of time where objects will be stopped. After this phase the object is free to begin moving.
3. On a frame-by-frame basis the agent checks for collisions with other moving agents, or agents that have stopped since the order was given. If a collision is detected, the path plan is revised according to the guidelines below.

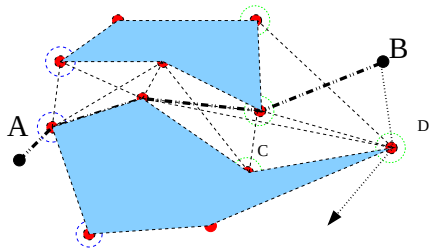

Fig. 1. Shaded polygons represent obstacles on a 2D terrain. The filled circles are the obstacle vertices and form nodes in the visibility graph. Dashed lines are the edges of the visibility graph for these obstacles. The horizon vertices for points A and B are circled in dashes and dots respectively. Note that a shortest path between any two points must first reach a vertex of the horizon of the starting point, traverse zero or more edges in the visibility graph and go from a vertex in the horizon of the destination point to the final destination point. The shortest path between A and B is shown in the figure. Note that horizon vertex C can not be on a shortest path to or from B as B can see both edges incident on these vertices. But since B can not see the both edges of D, it can be on a shortest path and one such path is also drawn.

When an object stops, i.e. reaches the end of its path plan, it is removed from the simulation until it is given another motion order so that stopped objects do not consume any dynamics time.

Objects can also construct partial path plans that are only valid up to an intermediate point along the path to their final destinations. This gives the path planner the option to terminate early before constructing the whole path. If an object reaches the end of its partial plan, another path is generated from the latest position of the current plan toward the final destination. The length of the sub-plan to be constructed is controlled by a parameter — the *plan ahead time*.

4.1 Fixed obstacles

In our simulator, fixed obstacles such as water and rocks are represented as simple polygons on the plane. We precompute a visibility graph over obstacle vertices: there is an edge between two obstacle vertices if a object can go between them without intersecting any fixed obstacle. Note that this assumes that all objects are the same size, but this is not a hard limitation as it can be circumvented by local revisions to the path plans.

The shortest paths between any two pairs of vertices in this graph are precomputed by running Dijkstra's algorithm for every vertex until shortest paths are found to every other vertex (figure 1). The shortest paths and their lengths between obstacle vertices can be stored compactly in a symmetric table where the entry (i,j) points to the next obstacle vertex to go on the shortest path from vertex i to vertex j. The algorithm for finding the edges of the visibility graph has complexity $O(n)$ [14] where n is the number of edges in the graph. The shortest paths between every pair of vertices on this graph can be computed in $O(n^3)$ and stored in a table of size $O(n^2)$. Recall that this is a precomputation, so the complexity does not affect the run-time performance. Our unoptimized implementation performs this computation in 5-10 minutes on a mid-performance PC for maps of typical game complexities.

Having computed the visibility graph and shortest paths, the plane is partitioned into *horizon regions*. Each horizon region contains points with the same *horizon*, or

set of visible obstacle vertices (figure 1). In this context, two points are visible if an object can go between them on a straight line without intersecting a fixed obstacle. Asymptotically, the number of vertices forming the horizon of a horizon region can be on the order of the total number of obstacle vertices in the environment. The number of horizon regions in the environment can grow exponentially in the number of obstacle vertices making the storage $O(n^n)$.

Despite the worst case complexity of each horizon region, and the potentially exponential number of regions, in our test maps the number of horizon vertices was on the order of 4-8 and the typical number of horizons was on the order of hundreds. These maps were typical of the complexity one would expect from a computer game environment. Moreover, if an obstacle vertex has its two incident edges visible from within the horizon region, then that vertex does not have to be included in the horizon as it can never be the part of a shortest path beginning or ending at that horizon, decreasing the size of horizons (figure 1). Horizons exhibit a great deal of spatial coherence, i.e. nearby regions usually have the same horizon, and can be efficiently stored with spatial data structures.

The resulting precomputed visibility graph, shortest paths on the visibility graph and the horizon decomposition of the environment are then used at runtime to compute the shortest paths. Each order consists of the start point, p_{start} (the agent's current location) and a destination point, p_{end}. To generate the path, we first obtain the set of horizon vertices for p_{start}, H_{start} and the set of horizon vertices for p_{end}, H_{end}. These sets are available from the precomputed horizon regions. We then find the pair $(v_{start}, v_{end}), v_{start} \in H_{start}, v_{end} \in H_{end}$ such that the path $p_{start} \rightarrow v_{start} \rightarrow v_{end} \rightarrow p_{end}$ has the shortest distance. The shortest path between v_{start} and v_{end} is available from the pre-computation for any v_{start}, v_{end}.

The optimal path can be represented as the pair (v_{start}, v_{end}). The actual path is a line segment joining the p_{start} to v_{start}, then the stored shortest path from v_{start} to v_{end}, followed by the line segment joining v_{end} to p_{end} (figure 1). Note that v_{start} and v_{end} may be the same vertex. Moreover, if p_{end} is visible from p_{start} then the shortest path is simply the line segment joining the two vertices (a case detected by raycasting).

The worst case complexity for finding the closest pair of horizon vertices is $O(h^2)$ where h is the number of obstacle vertices. However, as indicated above, horizons tend to be of a small size, so the enumeration is very fast. In our path planning system, planning paths around fixed obstacles was cheap in terms of real time computation even on the most complicated maps. In terms of asymptotic cost, there exist $O(n)$ space and $O(log(n))$ query algorithms to compute the shortest path between two points in a polygonal environment where n is the number of obstacle vertices [6, 8]. However, the constant factors for the asymptotically optimal algorithms are large and such algorithms perform worse that our system.

4.2 Stopped objects

The cost of inserting objects into the horizon data structure when they stop moving (and removing them when they restart) is prohibitive. Instead, we plan paths around stopped objects on the fly. Once the global path plan that avoids fixed obstacles is available, it is checked against other objects for any collision. If a stopped object is on the intended path, then the plan has to be modified to go around the obstruction. Such stopped objects can be partitioned into object groups where the distance between any two objects in distinct groups is large enough to let an object through. Thus, if an object group obstructs the path, the only alternative is to traverse all the objects in the group.

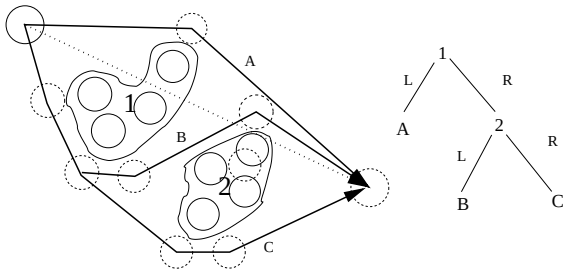

Fig. 2. There are static objects, groups 1 and 2, on a moving object's intended path (shown as a dashed line). The possible paths are enumerated by considering traversing the left and right sides of the obstructing group (1) and continuing the process recursively. Note that traversing the one side of the obstructing group (1) may reveal other obstructions (2). The plans discovered this way can be represented as a binary tree whose leaves are the candidate paths.

All the paths around the obstruction can be enumerated recursively by considering the left and right shoulders of the obstructing group (figure 2). Note that this enumeration procedure must still take the fixed obstacles into account. The enumeration procedure can stop when all the possible evasive patterns are discovered and evaluated for optimality or a long enough path segment is discovered (controlled by the plan ahead time parameter). In the first case, while we are guaranteed an optimal path, the cost of obtaining the path is considerable. In the second case, the partial path may not be optimal, but the cost of planning the partial path is much lower, allowing us to generate partial plans for more objects on any single frame.

During this enumeration step, the paths must be checked against fixed obstacles in order to avoid paths that go through them. If no path that doesn't go through fixed obstacles is found, then we must abandon the original path plan as it can not take us through the obstruction. This can happen if a bridge is blocked by other objects so the objects that need to pass the bridge must use an alternative route, found by searching the visibility graph for a path that doesn't contain the obstructed original path. As this step is usually time consuming, we can also precompute and store the second and third shortest paths along with the best paths between every two horizon vertices and only resort to a search if all three paths are blocked.

Note that a stopped object can start moving after an evasive path is computed around it. Thus, the computed path may not be optimal under the new situation. In our simulator we have achieved acceptable results without re-planning for such a case.

4.3 Moving objects

The intended path for a given object can be further obstructed by other moving objects. The simulator discovers such collisions by checking the new position of a moving object at every time step. If two dynamic objects will collide, we stop one of the parties, chosen at random, and wait for the other one to move and clear the path. If waiting for either one of the interacting parties does not solve the problem, we temporarily stop one of the objects and force the other one to plan around it using the previous section's algorithm for planning around stopped objects. If one of the objects is already stopped (as can happen in an agent stops after the path was initially planned) then we must also replan for the agent that is still moving.

In an environment with lots of dynamic objects, we are likely to encounter many potential agent collisions at run time, so this step causes lots of path replanning in order to find paths around other objects. This is the primary contributor to the computation time. However, if the agents are not visible, we are not concerned which precisely which local path they take to avoid each other. We care only that they appear to have taken some detour and hence been delayed by the presence of other agents, but have no interest in precisely which detour. In the next section we describe a modified algorithm that avoids replanning for out-of-view collisions but still captures the effects that such replanning would have on the agent's behavior.

5 Proxy Simulation

All that ultimately matters in a virtual environment is that the viewer have a reasonable experience in the environment. The viewer can typically only experience what they can see, so we are free to do anything with the out-of-view motion in the environment provided that it does not impact significantly on what happens in view. Ultimately, anything that impacts the viewer's experience must enter the view, so it is these view entry events that we are primarily concerned with capturing with our out-of-view simulation. We might also need to capture other out of view events to ensure that we get a reasonable stream of view-entry events.

We refer to a simulator that generates view-entry events as a *proxy simulator*. For highly efficient simulation, we will allow the proxy simulator to approximate the out-of-view motion, but in such a way that the viewer does not experience the difference. In a computer game environment, this means that we must be sure that a player who knows about the proxy should not have an advantage over a player who does not. Thus, the approximate simulation should meet the following requirements for every individually identifiable object:

- Objects in any part of the battlefield should not pass through fixed obstacles (otherwise the informed player could "teleport" objects across walls and rivers).
- Inter-object interactions must be accounted for. For example, an object should not be able to pass through a bridge whose entrance has been blocked by other static objects.
- Objects should have speeds consistent with their environments. For instance, the average speed of an object should not be very big if it has interacted with lots of other dynamic objects recently. This means that objects should (a) arrive at their destinations at times consistent with a full simulation and (b) have encounters along their paths that are also consistent. For example, objects should encounter ambushes at consistent times.

5.1 Tracking Agent Locations

In order to determine when an agent should re-enter the view, we will keep track of its approximate location. This allows us to determine which objects should be visible if the viewer moves, and also allows us to detect and account for the interactions between out-of-view agents and their environment. We subdivide the battlefield to smaller rectangles that act as our cells. The viewer has a rectangular overhead view of the battlefield, so can only see the cells that intersect with this rectangular region. Only the objects that are in these cells may be visible. Furthermore, only agents in the same cell at the same time can interact with each other.

The proxy model for path planning and collision avoidance is a discrete event simulator that is used to manage the membership information of every object for every cell. More specifically, the proxy is used to ensure that each cell contains a list of objects that are stamped with the times they will enter and exit the cell. At every frame, dynamics state for objects entering any of the visible cells is obtained from the proxy simulator and the proxy simulation is turned off for those objects. Similarly, objects leaving the visible cells are switched to the proxy simulation.

The behavior of a discrete event simulator is defined by the events it uses and the processing it performs on them. We begin by describing three events, and introduce another later. We also defer the discussion of how the times for these events are predicted.

Stop: An object in the proxy simulator has reached its destination. When this event happens, the proxy simulator switches the object from dynamic to static and removes it from all the cells that it has been associated with. Note that the stop event may have formed a blockade that prevents other dynamic events from moving. Thus, when a stop event is processed, we look for dynamic objects that collide with the stopping object and recompute their events if necessary.

Replan: Since objects may also have partial path plans, the validity of a path plan may expire before the object reaches its final destination. In this case the proxy must generate a new plan from the last point in the current plan to the final target of the object.

Entry: This event is scheduled for agents in the proxy simulator that will enter one of the visible cells. When this event happens, the proxy simulator deletes the agent from any cells it is in. A complete dynamic state must also be assigned to the agent. This is done by randomly assigning state based on the time the agent entered the cell and its expected behavior within the cell, as described below.

5.2 Setting Event Times

The proxy simulator needs to be able to predict when an agent will reach the end of its plan, so that stop or replan events can be scheduled, or when the agent reaches a visible cell, so that enter events can be scheduled. This is done through the use of hierarchical data structures that determine which objects might interact with each other, and a probabilistic model to estimate the impact of the interactions on the event times.

The process of giving orders and planning around fixed and stopped objects is not changed by the proxy simulator. These operations are already fast, and they provide valuable information about where the object will go and how long it will take to get there. In particular, combined with the constant velocity of the object, we can use the basic plan to compute the soonest that an event can occur. The proxy avoids replanning around the collisions between dynamic agents, which are the dominant simulation cost for the accurate model. These interactions can only add delays to an agent's travel time, and it is these delays that we model probabilistically.

The proxy simulator gathers the path plans for the objects that it needs to simulate and inserts each object into the cells that its planned trajectory intersects. The proxy simulator also marks each object with the corresponding entry and exit time stamps for every cell that it is scheduled to visit. The time stamps are computed by considering the object's speed and the other objects with which it is expected to interact along the path.

These interactions are discovered using a quad tree decomposition of every cell that is used to eliminate objects whose paths don't cross. Whenever an object is inserted into a cell, the root of the quad tree for that cell is investigated for other objects. If another object also occupies the same quad tree node, then both objects are refined by

inserting them into the children node. Note that the object already in the node can be directly pushed to a lower level in the tree as it can not collide with another object for if it did, it would be in a lower node.

The process is continued by traversing the nodes in the order until no other object exists in the node, meaning object will not collide, or the maximum depth of the tree is reached, meaning a possible collision. Note that since the path plans are piecewise linear, the insertion can be done by traversing each linear segment and inserting it into the nodes that it intersects in order. If we descend all the way to the leaf, we then check the objects in the leaf node for a collision. The maximum depth of the tree is a user specified parameter that can be tuned with cell and object sizes.

Detected interactions are approximated by delays to the original path plans of both interacting parties. This is reasonable since a dynamic object will either move and clear the path or one of the interacting parties will plan around the other one, usually resolving the collision without getting stuck.

The delays are sampled from a probability mixture distribution obtained by sampling the delays that result from object interactions in the full simulation and fitting the mixture with a EM algorithm [3]. The resulting distribution is usually good enough to capture accurate speed relationships for the dynamic objects.

Introducing a delay in an object's intended plan may invalidate some of the interactions that have been computed previously, in particular all those objects that previously expected to interact with the delayed object. Thus, all the future interactions must be rechecked after the introduction of a delay. Since objects can have very long path plans into the future, this step may involve re-computation of interactions for lots of other objects already in the proxy simulator. We overcome this problem by computing interactions only for a particular amount of time in the future and rechecking. The amount of time that we will compute an object's interactions is controlled by a *latest interaction time* parameter. This introduces one additional event:

Reinsert: This event is generated for every object when their latest interaction time is hit and causes the proxy to look for additional interactions along the agent's path.

The above procedure provides sufficient information to estimate event times for the simulator. It is run after an order is given and the basic path plan is computed. The interactions along the path are noted, delays for each interaction are sampled, and then added to the minimum travel time to obtain estimated times for the events.

The use of a distribution on delay times makes some assumptions about the nature of the delays encountered by an object. The primary assumption is that the delay depends linearly on the number of agent's encountered along a path. This seems to be a reasonable assumption in our particular situations. If it was not valid, the distribution could be parameterized by the number of interactions, although at some cost to the learning process and storage. Another implicit assumption is that delays do not depend on how long the simulation has been running or other time-dependent parameters, which is certainly reasonable in this case and is likely to be a valid for most environments.

In this framework, the viewer motion can be accommodated by sampling states for the objects in the cells that are becoming visible. Since we have the entry times and the delays that objects accumulate for every object in the cells, we can efficiently estimate their positions. Objects becoming invisible due to the viewer motion don't require special treatment and can be handled by computing necessary events and inserting them into the cells they will visit.

5.3 Discussion

The main source of error in the proxy simulator is the approximation of dynamic object interactions as delays. However, these delays are estimated based on data from the full model, so on average they cancel out. One infrequent case where the proxy is seriously awry is when two objects deadlock in trying to pass. This happens if two objects try to pass a narrow bridge in opposite directions and is due to the lack of a "back up" logic in our path planner. If the bridge is only one object wide, then each object will wait for the other object to clear the path as active replanning fails to plan around the other object. Such a situation could be detected by checking if the interacting parties succeed to find an evasive path plan. The deadlock could then be resolved with a smarter path planner that can back up objects to clear the path for others. However, in our current implementation we ignore the deadlock because it has little impact on the behavior of the simulation as a whole. Narrow gaps are typically rare and the probability of units trying pass in opposite directions at the same time is low.

Note that proxy simulator does not have to give exactly the same results with the accurate simulation. Individual objects can have different states and follow different paths with the accurate simulator as long as they do not contradict a user's expectations. For example, an object should not appear or disappear from the user's screen or get to its destination too soon or late. Our proxy simulator satisfies this requirement most of the time, however, it is possible to create difficult scenarios where the proxy simulator, as it is, will generate inconsistent states with the full simulator. The true success of the proxy simulator must be measured with a large set of human experiments, which we have only performed within our research group.

6 Results

We have tested our proxy simulator on different terrains with different number of uniquely identifiable objects. Figure 4 shows the amount of dynamics computation time required to simulate 1600 objects for a simulator using a proxy compared to the full simulation. Figure 3 shows the amount of dynamics time required for simulating 1 second of dynamics as a function of the number of objects with constant density on a Pentium III 800MHz computer. For all simulations, objects are distributed randomly on the terrain and pick random places to go. The object density on the terrain is kept constant by increasing the size of the terrain as the number of objects is increased.

Increasing the number of objects increases the number of possible interactions quadratically. Since the full simulator needs to re-plan in order to avoid collisions, it may have to deal with additional collisions that the revised plan may cause, suffering additional cost for every interaction that needs plan revision. On the other hand, the proxy simulation approximates these interactions with probabilistic delays and thus does not re-plan. Even though the proxy simulator still detects all the interactions between the dynamical objects, the cost of handling these interactions is considerably cheaper than the full simulation.

7 Conclusion

We have demonstrated a novel path planning algorithm for large numbers of agents operating in a virtual environment. Our algorithm is particularly efficient because we take advantage of out-of-view motion to reduce the cost of the simulation.

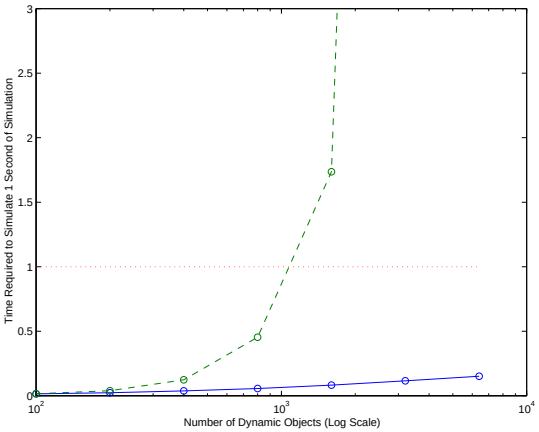

Fig. 3. The real time required to simulate 1 second of simulation time as a function of the log of the number of objects on a constant density terrain. The upper curve (dashes) represents the simulation without proxy and the lower curve (full) represents the simulation with proxy. The dotted horizontal line is the real-time cutoff. Since the object density is constant for all simulations, the user sees approximately the same number of objects in the view. Since the proxy simulator does very little work for the invisible objects and their interactions are handled using a discrete event simulator, the simulation cost increases much slower than the full simulation. The very fast increase in the full simulation cost comes from the fact that the path plan revisions needed to avoid collisions between dynamic objects tend to snowball, while the proxy approximates dynamic-dynamic object interactions with probabilistic models, thus avoiding local path plan revisions.

We can not guarantee the correctness of our proxy model all the time as it can be made to contradict with the accurate simulation with elaborate setups (e.g. objects passing through narrow bridge in opposite directions). However, proxy simulations may be the only way to simulate environments that are simply too big for the accurate simulation, or cut the state that has to be propagated in distributed virtual environments. Note asymptotic gains in the scalability of simulations are only possible through using proxies. Thus guaranteeing good behavior of proxy simulations is an important research topic.

Ultimately our aim is to provide tools that aid in the construction of proxy simulations. Such tools would accept an accurate simulation in some form and, possibly with user intervention, construct a proxy that saved computation but retained the important properties of the original simulation model.

References

1. J. Barraquand and J. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. In IEEE Int. Conf. Robot. & Autom., pages 1712–1717, 1990. 257, 1990.
2. Michael Batty, Bin Jiang, and Mark Thurstain-Goodwin. Working paper 4: Local movement: Agent-based models of pedestrian flows. Working Paper from the Center for Advanced Spatial Analysis, University College London, 1998.
3. Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

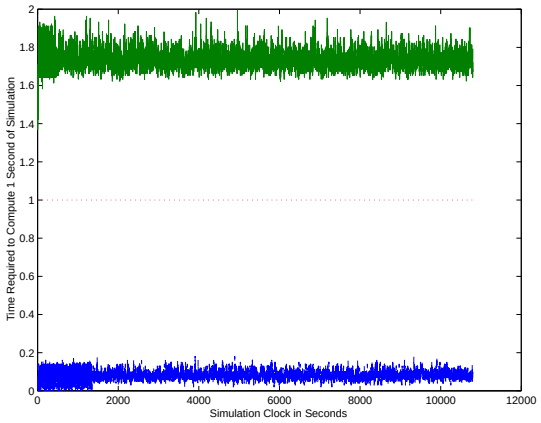

Fig. 4. The dynamics computation time required for one second of actual simulation on the vertical axis for 3 hours of simulation for the full simulation (at the top) versus the proxy simulation (at the bottom). The simulation is run with 1600 objects picking random places to go on a moderately dense map.

4. Stephen Chenney, Okan Arikan, and D.A.Forsyth. Proxy simulations for efficient dynamics. To appear in Eurographics 2001, Short Presentations.
5. B. Faverjon and P. Tournassoud. A local based approach for path planning of manipulators with a high number of degrees of freedom, int. conf. robotics & automation, 1987.
6. Guibas and Hershberger. Optimal shortest path queries in a simple polygon. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 1987.
7. Demis Hassabis. Level-of-detail ai. Lecture at the 2001 Game Developers Conference.
8. Joseph O'Rouke Jacob E. Goodman. *Discrete and Computational Geometry*. The CRC Press, Boca Raton, New York, 1997.
9. Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 1996.
10. Jean-Paul Laumond. *Robot Motion Planning and Control*. Lectures Notes in Control and Information Sciences. Springer Verlag, 1998.
11. J. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, CRC Press LLC, Boca Raton, FL, 1997.
12. L. Overgaard, H. Petersen, and J. Perram. Reactive motion planning: a multi-agent approach. Applied Artificial Intelligence, 10(1), 1996.
13. M. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Department of Computer Science, Utrecht University, The Netherlands, 1992.
14. S. N. Maheshwari Sanjiv Kapoor. Efficiently constructing the visibility graph of a simple polygon with obstacles. In *SIAM Journal on Computing*, volume 30(3), pages 847–871, August 2000.
15. Dimitris Metaxas Siome Goldenstein, Edward Large. Special issue on real-time virtual worlds: Non-linear dynamical system approach to behavior modeling. In *The Visual Computer*, volume 15, pages 341–348, 1999.
16. Marjolaine Tremblay and Hiromi Ono. Multiple creatures choreograhy on Star Wars: Episode I "The Phantom Menace". SIGGRAPH 99 Animation Sketch. In Conference Abstracts and Applications, page 205, August 1999.