Layered Intelligence for Agent-based Crowd Simulation

Bikramjit Banerjee, Ahmed Abukmail and Landon Kraemer

SIMULATION 2009; 85; 621
DOI: 10.1177/0037549709340659

Layered Intelligence for Agent-based Crowd Simulation

Bikramjit Banerjee
Ahmed Abukmail
Landon Kraemer
School of Computing
The University of Southern Mississippi
118 College Drive #5106
Hattiesburg, MS 39406-0001, USA
{bikramjit.banerjee, ahmed.abukmail, landon.kraemer}@usm.edu

We adapt a scalable layered intelligence technique from the game industry, for agent-based crowd simulation. We extend this approach for planned movements, pursuance of assignable goals, and avoidance of dynamically introduced obstacles/threats as well as congestions, while keeping the system scalable with the number of agents. We demonstrate the various behaviors in hall-evacuation scenarios, and experimentally establish the scalability of the frame rates with increasing numbers of agents.

Keywords: agent-based simulation, crowd behavior simulation

1. Introduction

Crowd behavior simulation has been an active field of research [1–5] because of its utility in several applications such as emergency planning and evacuations, designing and planning pedestrian areas, subway or rail-road stations, as well as in education, training and entertainment. In agent-based crowd simulations, where each pedestrian is modeled as an autonomous agent, a tradeoff is commonly made between the complexity of each agent and the size of the crowd. This is because, by common wisdom 'simple characters are more efficient to evaluate, but complex characters can capture more realistic crowd behaviors' [6]. The assumption underlying the above quote is that realistic crowd behaviors are hard to achieve with simple agent models. Although we only focus on navigational behaviors in this article, we show that it is possible to model complex behaviors realistically (such as static obstacle avoidance, separation, collision avoidance, approaching assignable goals, and avoidance of dynamically introduced obstacles/threat) with an extremely simple agent model, leading to a scalable simulation system.

SIMULATION, Vol. 85, Issue 10, October 2009 621–633
© 2009 The Society for Modeling and Simulation International
DOI: 10.1177/0037549709340659
Figures 1–11 appear in color online: http://sim.sagepub.com

The main idea is to distribute the intelligence in the terrain [7] rather than accumulating it into a complex/bulky model that each agent must follow. Although this idea of *smart terrain* is not new, to the best of our knowledge, this is the first application of this idea to crowd simulation. More importantly, we advance this approach to incorporate new behaviors that are specific to crowd simulation.

In this article, we focus on crowd movement on a two-dimensional surface. We use the layered artificial intelligence (AI) framework [7] to create an efficient platform for agent movement, that is also easily expandable to incorporate increasingly complex behaviors at will, by simply adding more layers. We first create a flow-field for basic agent movement, avoiding static obstacles in the world, using the Markov decision process (MDP) [8] framework. We show that realistic behavior in this context needs a refinement that *semi-Markov* decision processes (SMDPs) offer. We also show how the combination of SMDPs and layered AI allows us to easily handle the assignment of different goals to different agents. This means that an agent is not limited to approaching the *nearest* goal, but an *assigned* goal, unlike what the SMDP framework alone offers. We also extend the layered AI framework to handle the dynamic introduction of new obstacles/threats, as well as congestions. One limitation of our approach is the pre-processing time needed to create the initial flow-field. We provide a discussion that shows that this step can be parallelized to reduce the pre-computation time. Finally,

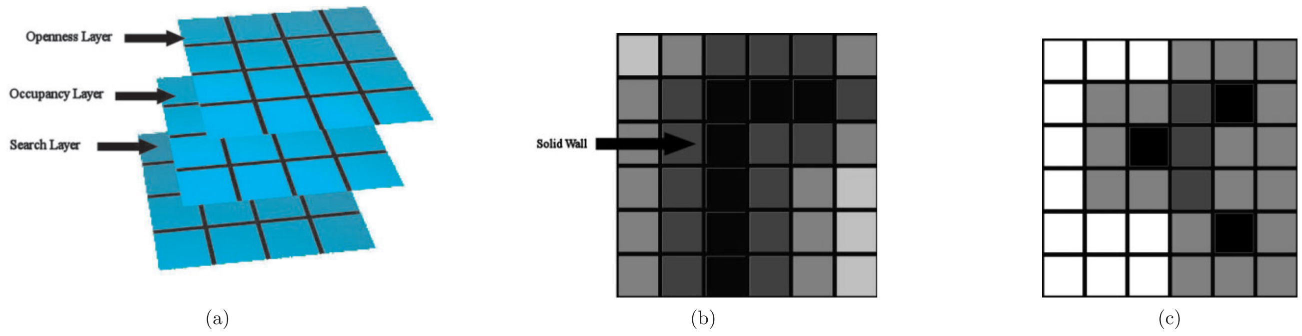


Figure 1. (a) Several informational layers overlay the underlying physical grid. (b) The openness/obstacle layer. (c) The occupancy layer. Black cells are occupied, progressively lighter cells are more easily walkable. These figures are adapted from [7] to illustrate the layered AI approach.

we show the frame rates resulting from our implementation, which clearly establishes the efficacy of our scalable approach in modern crowd simulation.

It should be noted that goal-directed movements that form a major component of our approach, may not be entirely suitable for emergency evacuation situations. However, several other kinds of applications require goal-directed movements, as noted by several authors in the past [1–3, 9, 10], and this aspect remains an important component of crowd simulation. Other components of our framework, such as handling dynamic obstacles and congestion avoidance are relevant to simulating emergency evacuations.

2. Layered Intelligence

We consider crowd behavior in an environment created on a two-dimensional surface. We divide the surface into square grids, where each cell has a sufficient area to hold no more than one person of average size. We have used the concept of layered AI from the game industry [7] for crowd simulation in this environment. The basic idea is to distribute terrain and other navigation-related information into several layers and have an agent make simple navigation decisions based on a combination of these layers. For instance, there could be a single layer called *occupancy layer* where each agent enters its current position. When an agent makes a decision of which cell to move to next, it will need to consult this layer and omit any neighboring cell that is already occupied by other agents. Once its decision is made, it will need to update its position on this layer, to avoid other agents colliding with this agent. Similarly there could be an *obstacle layer*, which contains information about all static obstacles in the environment. When deciding which neighboring cell to move to next, an agent must also consult this layer to omit cells that are blocked by obstacles. Rather than binary (blocked or available) values, the layers usually contain values from a continuous range of [0, 1] to indicate proximity to agents/obstacles. The approach is illustrated in Figure 1. Here, the layers contain values closer to zero in darker cells (zero for black cells) indicating locations that are harder to occupy, and values closer to one in lighter cells (one for white cells) which can be occupied more easily.

Essentially, each type of information that is relevant to navigation is captured in a separate layer. In games, information such as which cells are easily visible and hence open to enemy fire, which cells have the enemies just searched, and are not likely to search again anytime soon, etc., are captured in separate layers, called the *openness layer*, *search layer*, etc. [7]. Normalized values (i.e. in the range [0, 1]) are stored for each cell in each layer, reflecting its value from that layer's perspective. Let $layer_i(x, y)$ be the value of cell (x, y) in layer i, with a total of L layers, $i = 1 \dots L$. An agent at location (x, y) needs to simply look up the values of all cells in the neighborhood of (x, y), i.e. $N(x, y) = \{(p, q) | (p, q) = Neighbor(x, y)\}$, *from all layers* and pick the next-best cell as

$$(p, q)_{\text{best}} = \arg \max_{(p,q) \in N(x,y)} \prod_{i=1}^{L} layer_i(p, q) \qquad (1)$$

In this article, we use simple formulae to compute openness/obstacle and occupancy layers (Figures 1(b) and (c)). The obstacle layer is simply binary (zeros occupied by walls and ones open) in contrast to Figure 1(b) which shows a larger range of values (gray levels), while the occupancy layer is computed as (conforming to Figure 1(c))

$$layer_{\text{occupancy}}(x, y) = \begin{cases} 0 & \text{if agent at } (x, y) \\ 0.5^k & k = \text{number of} \\ & \text{agents in } N(x, y) \end{cases} \qquad (2)$$

The above formula for the occupancy layer is actually implemented as a constant-time process per agent, and encourages slight (just one cell deep) separation among agents (as shown in Figure 1(c)), unless they are pressed in a congestion. In every new frame, an agent only needs to update the neighborhood of its new location, and re-adjust the neighborhood of its previous location, in addition to the two successive locations.

The complexity of the decision-making process (Equation (1)) is $O(|N|L)$. Hence, with a fixed sized neighborhood (say the nine cells surrounding and including (x, y)), the decision would be constant time. With n agents in the environment, the total time complexity for generating the next frame of agent positions would be $O(n)$, which is the best possible speed we can hope for in a distributed simulation (since it is $\Omega(n)$ in sequential machines). The layered approach to intelligent decision-making abides by the following principle that game AI developers value for scalability and efficiency: *put the intelligence in the data, not in the code*.

3. Path Planning

In this article, we extend the layered approach to include path planning to allow planned movements as opposed to reactive movements. We also allow different destinations for different agents, and build a *path-plan layer* for *each* destination. An agent will thus consider only the path-plan layer for its desired destination and ignore the other path-plan layers. As an illustration consider an agent egressing a building to approach one of four surrounding parking lots, where they have parked. We can have a path-plan layer for each of the four parking lots, and have many agents approach their respective parking lots concurrently by consulting the appropriate layers. The success of this approach obviously depends on the number of destinations being small. The advantage is *open design*, i.e. we can extend and complexify the scenarios arbitrarily, simply by building extra layers.

In games, path planning is usually performed offline (i.e. pre-computed) with a Floyd–Warshall technique, or online with A* for dynamic path-finding. In the layered approach, we are primarily concerned with minimizing run-time processing; hence, we completely eliminate the need for A* with the intent of handling dynamic changes to the optimal path entirely within the layered framework. Although the Floyd–Warshall technique would give us path plans for all start–end cell pairs, the format of the output is not quite conducive to the layered approach. We need a path-planning technique that will produce a flow-field that tells an agent which neighboring cell it should move to, given its current cell location. Furthermore, these decisions should be based on real numbers that can be meaningfully combined (using Equation (1)) with other layers to formulate a more informed movement decision that is also based on occupancy, static obstacles and possibly dynamic obstacles. Another major limitation of the Floyd–Warshall algorithm is that it only gives the optimal decision from any cell, but fails to offer an alternative

Algorithm 1 The value iteration algorithm from [8]

```
1: Input real $\epsilon$
2: Initialize $V(s) \leftarrow 0, \forall s \in S$
3: repeat
4:   $\Delta \leftarrow 0$
5:   for each cell, $s \in S$ do
6:     $v \leftarrow V(s)$
7:     $V(s) \leftarrow \max_a R(s, a, T(s, a)) + \gamma V(T(s, a))$
8:     $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
9:   end for
10: until $\Delta < \epsilon$
```

if that optimal move is impossible to make (e.g. because another agent is currently occupying that cell). These limitations force us to look beyond the Floyd–Warshall technique.

3.1 MDPs

We view movement on the two-dimensional grid as a MDP. Formally, a MDP is given by the four-tuple $\langle S, A, T, R\rangle$, where S is the set of environmental states that an agent can be in at any given time, A is the set of actions it can choose from at any state, $R : S \times A \times S \mapsto \Re$ is the reward function, i.e. $R(s, a, s')$ specifies the reward from the environment that the agent receives for executing action $a \in A$ in state $s \in S$ leading to state s'; $T : S \times A \times S \mapsto [0, 1]$ is the state transition probability function specifying the probability of the next state (s') in the Markov chain consequential to the agent's selection of an action (a) in a state (s). A MDP solver's goal is to learn a policy (action decision function) $\pi : S \mapsto A$ that maximizes the expected sum of discounted future rewards from any state s,

$$V^{\pi}(s) = E_T\left[R(s, \pi(s), s') + \gamma R(s', \pi(s'), s'') + \gamma^2 R(s'', \pi(s''), s''') + \cdots\right] \quad (3)$$

where $s, s', s'', s''', \ldots$ are samplings from the distribution T following the Markov chain with policy π, and $\gamma \in [0, 1)$ is the discount factor. In the above equation that specifies this expected sum of discounted rewards starting from state s, $R(s_k, \pi(s_k), s'_k)$ is the reward for taking action $\pi(s_k)$ in state s_k and transitioning to state s'_k, where the next action is to be chosen. The discount factor γ^k associated with the kth step reward reduces the value of future rewards relative to current rewards so that an agent has the incentive to achieve higher rewards as early as possible.

In the context of the two-dimensional grid, S is the set of all grid cells. The action set A is the set of nine cells that an agent at location (x, y) can move to (including the action of staying put in cell (x, y)). Some of these neighboring cells may be blocked by static obstacles, in which case the corresponding actions are unavailable, and excluded from set A (or, equivalently, the corresponding transition probabilities $T(.,.,.)$ are set to zero). We use the information from the static obstacle layer to compute the MDP solution, so that the resulting path plans avoid them in a realistic manner[1]. We use value iteration [8] algorithm to solve the MDP induced by the two-dimensional grid with the reward function defined simply as

$$R(s,a,s') = \begin{cases} 1 & \text{if } s' \text{ is a goal cell} \\ 0 & \text{otherwise} \end{cases}$$

and transition function (T) is deterministic, meaning that an action a in state s leads to a unique next-state (s') every time. Therefore, for our purpose, we can redefine T as

$$T : S \times A \mapsto S$$

i.e. $T(s,a) = s'$ in accordance with the above. The value iteration algorithm is shown in Algorithm 1. This is a dynamic programming approach to calculating $V(s)$ for all s, where step 7 is the main one-step backup operator that estimates the new value (V) of a state, given the best combination of current reward and the discounted old value function (V from the previous iteration) of its neighbors. This iterative refinement continues until the largest value modification (Δ) is below a certain threshold (ϵ). The result (i.e. $V(s)$, for all $s \in S$) of this process can be readily used by an agent to select an appropriate action in any state by

$$\pi(s) = \arg\max_a V(T(s,a))$$

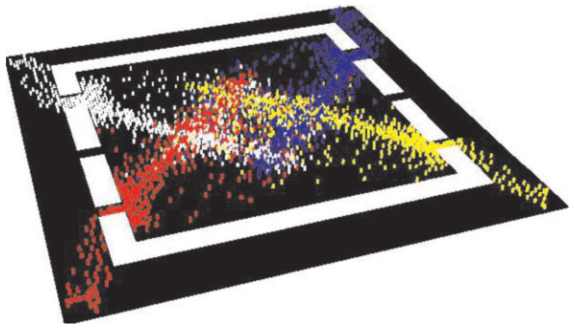
If several actions have the same maximum value, then the agent can pick one of these at random.

3.2 SMDPs

Different actions often take different amounts of time to complete. For instance, when an agent wants to move from cell (x, y) to cell $(x + 1, y)$ (or any of the four non-diagonal neighboring cells), the distance covered (say centroid to centroid) is less than if the agent wanted to move to $(x+1, y+1)$ (or any of the diagonal neighboring cells). In particular, if the former distance is one unit, the latter is $\sqrt{2} = 1.414$ units, assuming square grid cells. Hence, the agent would take longer to execute a diagonal step than a Manhattan step. This discrepancy in action times can be neatly incorporated in the MDP formalism to produce SMDPs, where the step 7 in the algorithm in Algorithm 1 needs to be modified to

$$V(s) \leftarrow \max_a \left[R(s,a,T(s,a)) + \gamma^{t(a)} V(T(s,a))\right]$$

1. This means that an agent starts avoiding an obstacle before they actually encounter it in their immediate neighborhood, simulating vision-based avoidance, much like A*.


(a)

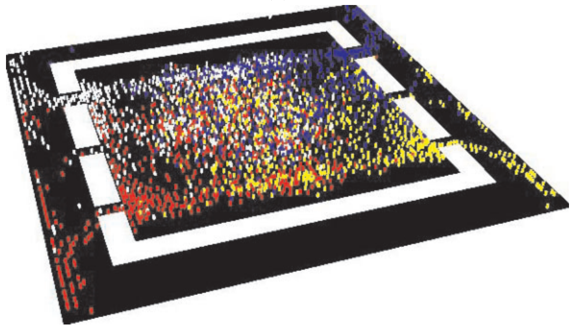
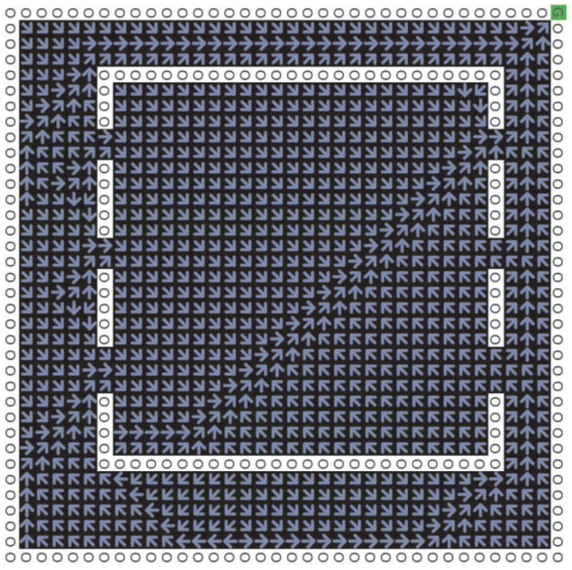

(b)

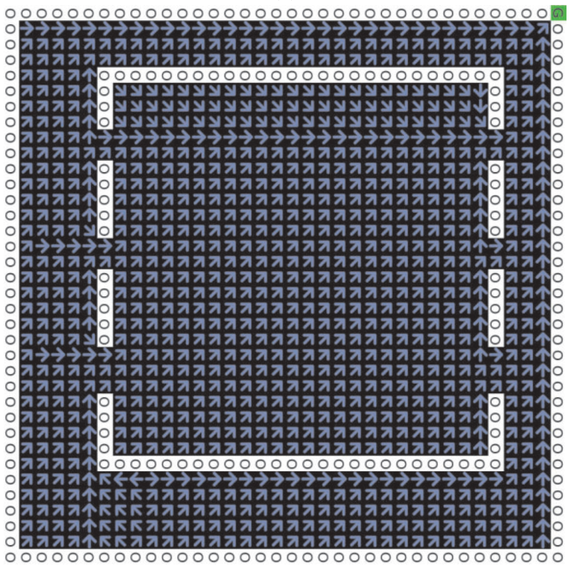
Figure 2. Crowd behavior (a) without and (b) with differing diagonal cost taken into account

where $t(a)$ is the time taken to execute action a. This adjusts the backup operator to reflect that the value of the neighboring state ($T(s,a)$) resulting from action a in state s is to be discounted (relative to the immediate reward, $R(s,a,T(s,a))$) to the extent of the duration of action a. Figure 2(a) shows the result of ignoring the difference in action execution times (i.e. $t(a) = 1$, for all a), while Figure 2(b) shows the result of taking these differences into account. In this simulation, 500 agents are placed in a hall with six exits, in a grid world of size 137×137. The four corners of the grid world are four different goals that an agent could move to. Each agent is colored by the goal that is randomly assigned to it. This simulation uses four path-plan layers (one for each corner goal) and an occupancy layer, but since we have used a binary static obstacle layer (in contrast to Figure 1(b)), this layer can be eliminated after the information needed to handle static obstacles is captured in the path-plan layers (e.g. note that the path-plan layer in Figure 3 contains binary static obstacle information). In Figure 2(a) the agents line-up in the direction of their respective goals (which we believe is an unrealistic artifact), and also fail to use the middle exits. In Figure 2(b), both of these problems have been eliminated by using the SMDP formulation. Figure 3 illustrates the difference between the flow fields produced by the two methods.

In this figure, the map has a lower resolution (36×36) for the sake of clarity, and the static obstacles are all



(a)



(b)

Figure 3. Navigation flow-field (a) produced by the algorithm in Algorithm 1 and (b) produced with the SMDP adjustment (b), for the path-plan layer with the goal at top-right corner

one cell deep (unlike in Figure 2), marked with zeros. The band of zeros around the periphery serves to establish the world's boundary. The goal cell is shown (in green) at the top right. It is worth noting from Figure 3 that while the raw MDP method pushes the agents away from the right-hand exits when in their vicinity, the SMDP adjustment nudges them toward those exits. This is the main reason for the agents being able to use the right-middle exit when in its vicinity.

One important limitation in this regard is that the agents may be unable to use the middle exits if they are far from them, while the other exits are temporarily congested with other agents. While the occupancy layer (Equation (2)) enforces some separation among agents and as a by-product may be sufficient (in some cases) to push the agent toward such unused (or scarcely used, hence less crowd producing, less separation force) exits, we would like to solve this problem without relying on separation alone. As a first-cut approach, this scenario calls for replanning which is expensive with the layered technique. One possibility is to treat congestion as a dynamically placed obstacle (see Section 5 for how we handle such obstacles), but the key difference is that while we may be told where a dynamic obstacle has been placed, the location of a congestion must be determined autonomously. We discuss how we use the concept of dynamically placed obstacles to simulate congestion avoidance behavior, in Section 6.

4. Complexity Analysis and Improvement

The pre-processing step to produce the flow-field for each goal in a separate layer (i.e. Algorithm 1 applied once for each goal), is computationally intensive and therefore it can benefit from improvement. The pre-processing time can be improved quite easily via parallelization. The parallel algorithm can be very simple and would result in significant improvements, especially as the size of the map grows. The idea would be to allow for the path-plan for each goal layer to be processed separately on a different processor. Since we have used four goal layers, a quad-core machine would suffice for computing the path plan for the goal layers concurrently. The path computation for each of the layers is independent of the others, because the path plan for one goal is independent of the other goals. The load can be distributed uniformly among all CPUs, with each writing the results to its own memory segment and would not require any synchronization. However, in this paper we only present the results from sequential implementation.

Figure 4 shows the number of seconds taken to generate the four path-plan layers, using a plain sequential method, on varying map sizes. The maps were of the same form as the previous figures, but the sizes of the obstacles grew proportionately with the map size. First, Figure 4 verifies a roughly cubic trend that is expected, because the loop 5 in Algorithm 1 is $O(m^2)$ (m being the number of cells along each dimension of the map), while the loop 3 has an average complexity of[2] $O(m)$. To put the computation time into perspective, a $1{,}200 \times 1{,}200$ map

2. Consequently, the value iteration algorithm is no more expensive than the Floyd–Warshall algorithm on average maps, i.e. unless the maps are unusual involving $O(m^2)$ (or worse) path lengths among various locations.

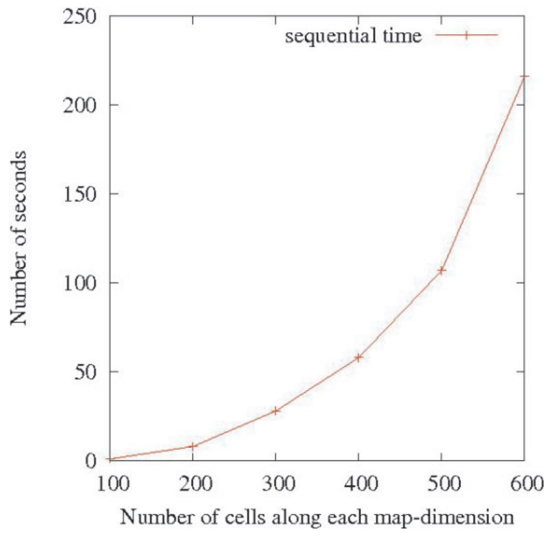


Figure 4. Plot of run-times for the path-plan layers for all four corner goals (Figure 2), run sequentially for various map sizes

covers approximately a quarter of a square mile according to the proportions used in our system, which roughly corresponds to the size of a football stadium.

As Figure 4 shows, an area one-quarter of that size (i.e. 600×600, or one-quarter of a stadium) takes 3.5 minutes pre-processing time which is a compelling factor in favor of the layered approach. Sucar [11] discusses how a MDP can be broken into subtask MDPs that can be solved in parallel with limited communication across processors for synchronization. We believe that this technique will significantly improve the speed-up of a parallel version of the algorithm, since it will be parallelizing not only the path plans for different goals, but also the path-plan layer for individual goals.

5. Dynamically Placed Obstacles

A second extension to the layered approach that we propose is to handle obstacles that are added dynamically. We use the word ‘obstacle’ in a broader sense, to refer to anything that an agent would want to avoid during navigation. For instance, rubble created by an explosion, agents that died in a stampede, a raging fire, etc., are all considered obstacles that a user adds after a simulation starts. As mentioned in the previous section, we would also like to treat a temporary congestion at a desired exit to be an obstacle that agents (with that exit on their path) must avoid, but a user will not be expected to locate it. All such obstacles necessitate dynamic changes to the optimal path plans. The layered framework enables a simple solution to this problem, albeit in a limited way. The idea is to create a new layer for a dynamically added obstacle, instead of modifying the path-plan layer. The new layer will create a *a trough* of values in and around the location of the obstacle, so that when combined with the other layers (Equation (1)), the agents will avoid bumping into it. In our implementation, for the purpose of simplicity an obstacle is characterized by five parameters:

- (c_x, c_y), the coordinates of the center of the obstacle;
- r_i, the inner radius, that is, the physical extent of the obstacle; for obstacles that are arbitrarily shaped, this is the radius of the bounding sphere.
- r_o, the outer radius ($> r_i$), that is, the extent of the influence of the obstacle; normally all obstacles extend their influence as far as visual distance, i.e. an agent that can see it will repath to avoid it; however, for some obstacles the influence could extend further; for instance, the effects of a fire or an explosion can be perceived from locations that are further away, and hence have larger r_o;
- a, the avoidance intensity, that specifies how strongly an agent would want to avoid stepping close to the obstacle; if $a = 0$, an agent can walk right by the obstacle (without stepping on it), whereas for higher a, an agent would want to avoid it by a larger distance, e.g. a fire.

In this *dynamic obstacle layer*, the value of a cell (p, q) is computed as

$$layer(p,q) = \begin{cases} 0 & \text{if} \quad d \le r_i \\ \left(\dfrac{d - r_i}{r_o - r_i}\right)^a & \text{if} \quad r_i < d < r_o \\ 1 & \text{if} \quad d \ge r_o \end{cases} \qquad (4)$$

where d is the distance between (p, q) and (c_x, c_y).

Figure 5 shows two successive snapshots ((a) and (b), roughly 2 seconds apart) from placing an obstacle at the center of the room. We have developed this tool to allow a user to place a circular obstacle with an adjustable r_i, at an arbitrary location with a mouse-click. All agents located within this obstacle immediately become immobilized, simulating dead agents[3] from whatever hazard (explosion, fire, etc.) originated the obstacle. In this figure, $r_o = 4r_i$ and $a = 1$. It should be noted that larger values of a enforce r_o more strongly making the agents form a more sharply defined circular pattern of avoidance. In contrast, a lower value (here $a = 1$) makes the circular outline more diffuse which we believe is more realistic. It is also noteworthy that Figure 5(a) is a snapshot roughly

3. This is implemented quite naturally in the layered framework, since unless an agent within the obstacle is in the periphery, it will be surrounded by zero-valued cells, according to Equation (4), and hence unable to move.

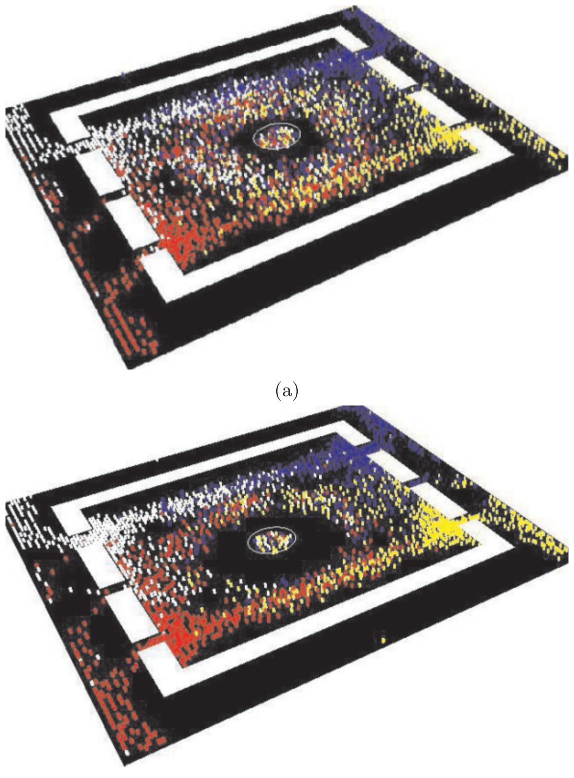


Figure 5. Two successive snapshots taken every 2 seconds after an obstacle is placed dynamically

2 seconds (run on a Lenovo Thinkpad 2.6 GHz dual core machine with 3 GB RAM) after the obstacle was placed, which includes the time taken to compute the new dynamic obstacle layer. On the said machine, the delay from this computation is nearly imperceptible on a 137×137 grid.

5.1 Discussion

A major advantage of a separate dynamic obstacle layer is the ease of handling temporary obstacles. Recall that one of our goals in this article is to use this technique to handle temporary congestions at bottlenecks. As such, when a temporary obstacle disappears (e.g. a fire dies down, rubble is removed by emergency response personnel or a bottleneck crowd dissipates), the agents must resume their original path plans. In the layered framework, it amounts to simply deleting the dynamic obstacle layer, since the original path plan is never modified. However, this also leads to a major limitation of this approach, during the period that the obstacle exists. Since the path plan is not modified by taking the existing static obstacles (such as walls) into account, it is possible that even though an agent is on the other side of a wall relative to the obstacle (and, hence, should be unable to see it), it still diverts along a curve if the avoidance field extends that far. In order to avoid this undesirable effect, it is necessary to modify Equation (4) to take the static obstacles into account. This enhancement is detailed in Section 5.2.

A minor limitation is that when an obstacle disappears, the (presumed dead) agents located within r_i will resume movement. This can be prevented simply by tagging all agents immobilized within r_i as dead agents. A dead-flagged agent will never move at any future time. Another limitation in the case of large maps is the fact that an entire layer is created for an obstacle that occupies only a small area. A simple solution is to create a sublayer for only the area covered by the influence field of an obstacle, and combine it with other layers much like applying a filter in image processing. Another shortcoming of the dynamic obstacle avoidance approach is that it is essentially a myopic/greedy modification of path plans that has the potential to trap agents in tight passes or corners. This problem is hard to avoid, although in principle, it may be feasible to determine a minimal path recomputation that completely adjusts an existing path plan layer to a dynamic obstacle, along the lines of minimal replanning by Focused Dynamic A* [12]. This is an important future direction.

5.2 Influence Clipping for Dynamically Placed Obstacles

In the research that was reported in [13], the influence of a dynamically placed obstacle was not clipped by static obstacles in the environment. Figure 7(a) shows the scenario, where two obstacles (white circles) are placed, and the influences, as computed by Equation (4), transgress the walls. We have implemented a simple and efficient technique to rectify this problem, that effectively simulates a *discrete* version of vision suitable for discrete grid domains, instead of the expensive continuous line-of-sight tests frequently employed in graphical simulations.

Given r_o, a square bounding box can be defined around the center (c_x, c_y) of an obstacle, such that no cell outside this box is affected by Equation (4). Initially all cells in this box are marked 'unprocessed'. For each unprocessed cell in this box, starting from the periphery and progressing toward the obstacle, we perform the *Bresenham line rasterization test* [14] that defines a sequence of cells that overlap with a straight line between the selected cell and (c_x, c_y). If this sequence includes a static obstacle cell, then we say that this line is blocked by a static obstacle. Bresenham's line algorithm has been popular in raster graphics, and is used to render a line on the screen pixel by pixel. In our case, the cells in the grid act as the pixels. An illustration of Bresenham's method between cells A and B (marked in red, with obstacles marked in black) is shown in Figure 6. The rasterized line is a series of cells

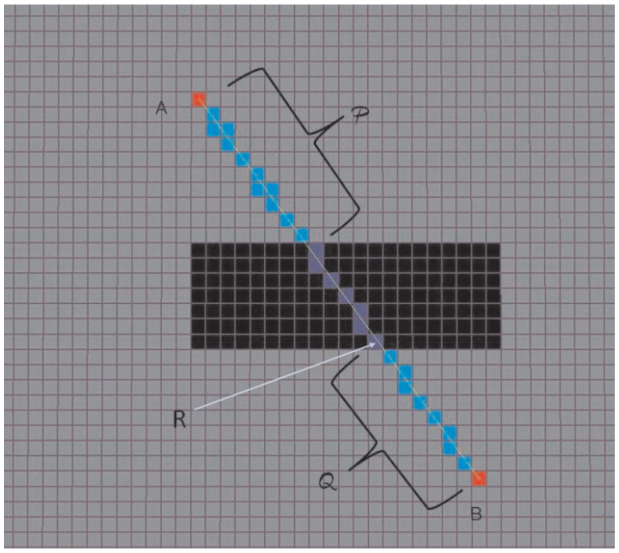


Figure 6. Illustration of Bresenham's line rasterization

marked in blue, with the dark blue cells being those overlapping with the obstacle cells. Thus, this line in Figure 6 is effectively blocked.

If a selected unprocessed cell has an open line of vision to the dynamic obstacle, then clearly all cells on the Bresenham line are also open. Equation (4) can be computed for these cells immediately, and then they are marked as 'processed'. On the flipside, if the selected cell (cell A in Figure 6) has a blocked line to the center (cell B in Figure 6, as discussed in the previous paragraph), then all cells between the blocked cell and the selected cell are blocked as well (i.e. the cells marked $\mathcal{P}$ in Figure 6). This observation allows us to immediately mark these intermediate cells as 'processed' with a value of one, while the cells (marked $\mathcal{Q}$ in Figure 6) between the blocked cell on this line that is closest to the obstacle (i.e. cell R), and the obstacle itself, are all open. Again, the values of these cells can be computed by Equation (4) before marking them as 'processed'. Since the Bresenham calls and Equation (4) do not have to be invoked for processed cells, significant savings in time are obtained by processing cells from the periphery of the bounding box. Figure 7(b) shows the result of this method on the same map as Figure 7(a).

6. Congestion as a Dynamic Obstacle

One of the major behavioral bottlenecks for intelligent agents in crowd simulation is the simulation of congestion avoidance behavior at bottlenecks. Although, this behavior is available in some existing systems [15], it is not immediately clear whether the lightweight agents in a layered approach can support such complex behavior. We break this problem into the following three subproblems, the first of which can be performed offline, similar to path planning.

- *Bottleneck identification*. Here we automatically locate the *potential* congestion points by analyzing the map offline.
- *Congestion identification*. At run-time, we check the local crowd density at each bottleneck for the occurrence of a congestion.
- *Congestion avoidance*. If congestion is identified at a bottleneck, an obstacle is dynamically placed at that bottleneck.

We now discuss our solution to each of these subproblems in further detail.

6.1 Bottleneck Identification

We identify bottlenecks by analyzing the *path-plan layers*, as created in section 3, as another pre-processing step. For any given goal, the path-plan layer for that goal provides flow information that can be exploited to identify potential congestion points, even before agents are deployed on the maps. However, depending on the relative locations of the goal and the static obstacles, the flow may entirely avoid some bottlenecks in the actual map, which means that such bottlenecks cannot be identified by those flows. For instance, in Figure 8(a), the general flow toward the orange goal will be as shown in arrows. Note that this flow will only uncover the bottleneck B, but will fail to highlight the bottleneck A, since the path plan will avoid this exit from both outside as well as inside the room. So we need to pool together the bottlenecks identified by the flows to different goals, and if a bottleneck is avoided by the flows of all goals, then it is reasonable to assume that the bottleneck will not be congested since the agents will likely avoid it.

Our algorithm to identify whether or not a state s is a bottleneck under policy π, is given by

$$bottleneck^{\pi}(s) = \begin{cases} \text{true} & \text{if } C(s) > C(T(s,\pi(s))) \\ \text{false} & \text{otherwise} \end{cases} \quad (5)$$

where $C(s) = |\{p \in S \mid T(p,\pi(p)) = s\}|$, i.e. the number of states that transition into s under policy π. Figure 8(b) illustrates this process, with the goal somewhere in the top right, and the policy shown for only the states in the vicinity of the bottleneck. Note that in this figure, $s' = T(s,\pi(s))$, $C(s) = 3$ and $C(s') = 2$. Therefore, according to Equation (5), state s qualifies as a bottleneck. Similarly, state s'' also qualifies as a bottleneck. It is straightforward to check in Figure 8(b) that this will not be true for the other states. In essence, Equation (5) detects a 'funnel shape' in the flow graph in any path-plan layer, where a state with a high in-degree transitions to a state with a lower in-degree.

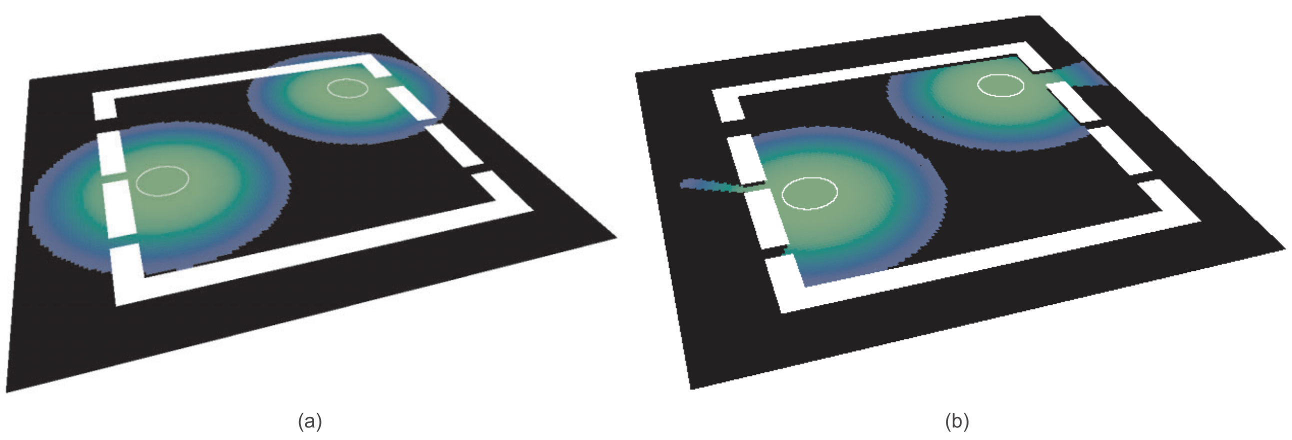


Figure 7. (a) Naive application of Equation (4), where the influence of dynamically placed obstacles transgress walls. (b) Influence clipping for dynamically placed obstacles. The influences approach zero toward the centers of the influence circles, while they approach one toward the periphery of the circles.

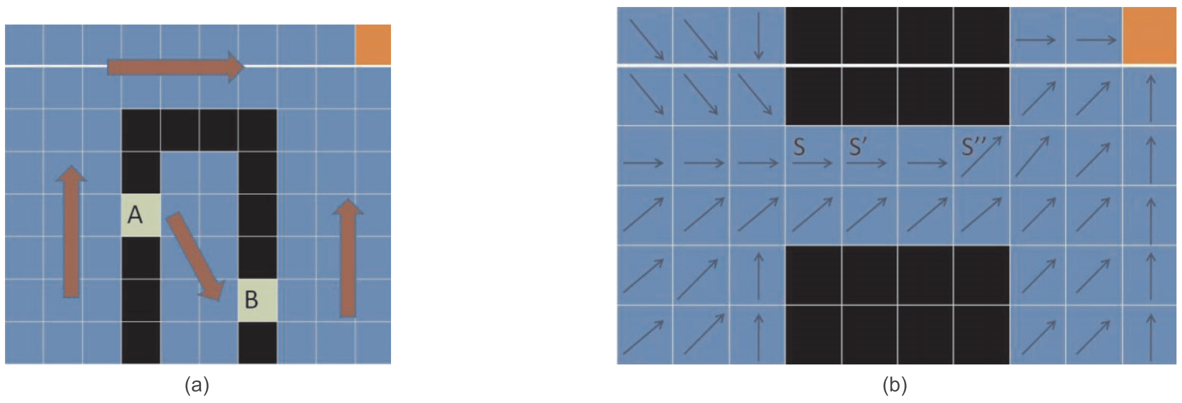


Figure 8. (a) A policy for a given goal may fail to identify some bottlenecks in a map, marked A in this case. (b) Illustration of the bottleneck-detecting algorithm. Black cells are the static obstacles and the top right cell is the goal.

Now, if B_{π_i} is the set of bottlenecks identified under the policy for the ith goal (π_i), i.e. $B_{\pi_i} = \{s \in S \mid \mathit{bottleneck}^{\pi_i}(s) = \text{true}\}$, then the overall set of bottlenecks *over all goals* $i = 1, 2, \ldots, G$, is given by $B_{1\ldots G} = \bigcup_{i=1}^{G} B_{\pi_i}$. In practice, however, the above method could create several bottlenecks that are close to each other. For instance, in Figure 8(b), both s and s'' are identified as bottlenecks, but since they are close to each other, it would be useful to have just one bottleneck somewhere between them, instead of two. We implement this as a refinement on $B_{1\ldots G}$, where all bottlenecks that are within a (small) distance δ from each other are replaced by one at their centroid.

The result of this bottleneck-finding method is shown in the two maps in Figure 9. Note that although these bottlenecks are not placed at exactly the centers of the doorways, this will not be a problem. This is because these bottlenecks will only be used to locate congestion, the influence of which is not clipped by the Bresenham method of Section 5.2, since such congestions are extensive and span around local obstacles. As a result, even though the centers of the congestions (the bottlenecks) are offset from the true locations, the influence field will roughly overlay the desired area. Figure 9 (left) also shows three excess bottlenecks around corners, but they are unlikely to be congested; hence, these extra bottlenecks do not affect the congestion avoidance behavior.

6.2 Congestion Identification

Given the list of bottlenecks, each bottleneck is checked for congestion every few frames. An area is defined around each bottleneck, and the number of agents in this area is recorded from the occupancy layer. If the density of agents exceeds a threshold (T_{HI}) then a congestion is identified at this bottleneck, and congestion avoidance behavior is invoked. The complexity of density computation

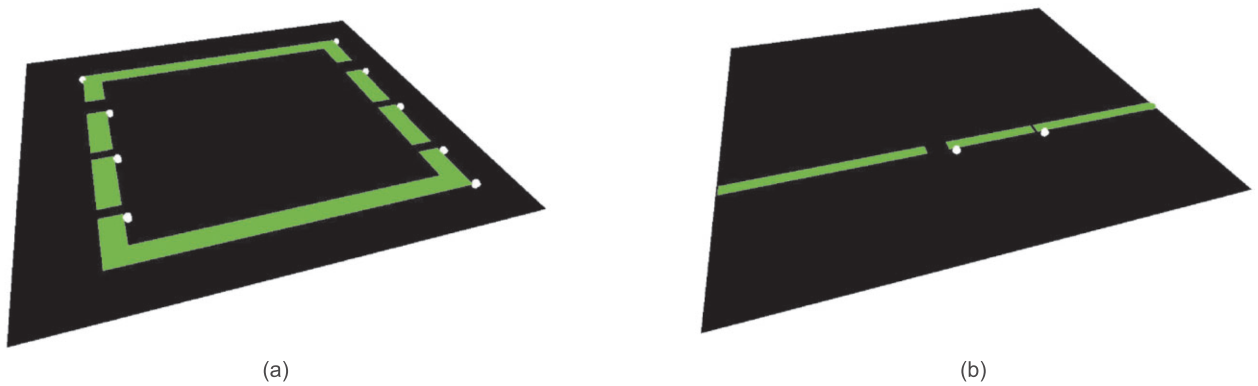


Figure 9. Bottlenecks, as computed by our method, are shown as white spheres on the two maps

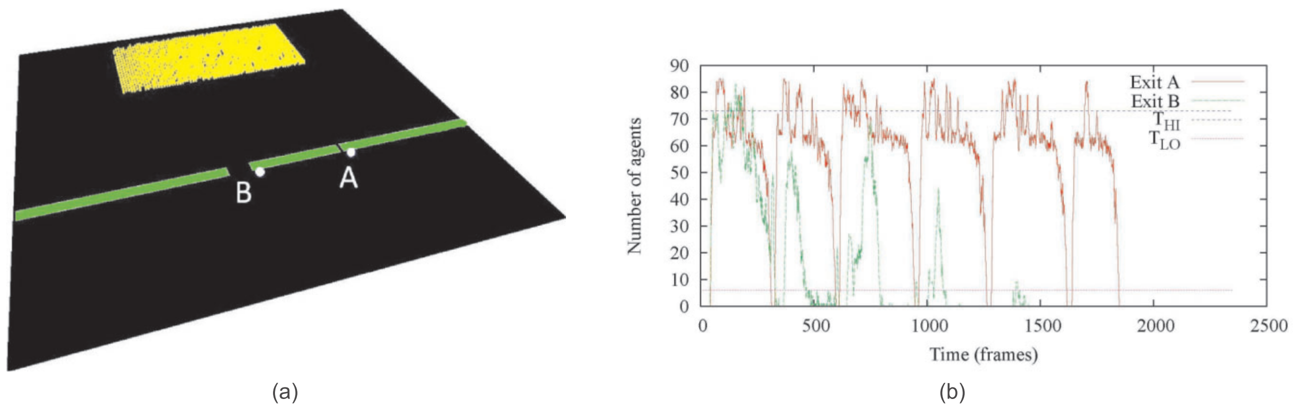


Figure 10. (a) Map to evaluate congestions; (b) plot of crowd size at the two exits, A and B on the map in (a)

is $O(|B_{1\ldots G}|k_m^2)$, where the maximal area around any bottleneck is chosen to be $k_m \times k_m$ cells. However, since the density may not change significantly from one frame to the next, it may be sensible to stagger the density computations over the set of the bottlenecks, leading to $O(k_m^2)$ cost per frame with a reasonable number of bottlenecks. Congestions are also monitored, and if the density falls below a threshold T_{LO} ($T_{\mathrm{HI}} > T_{\mathrm{LO}}$), then the congestion is removed, by cancelling the congestion avoidance behavior (i.e. by simply deleting the congestion layer created) at that bottleneck.

6.3 Congestion Avoidance

In order to prevent agents adding to an identified congestion, we dynamically place an obstacle at a congestion, with the following choices: the size of the congestion region is chosen as the value of r_{i} and r_{o} is a large multiple of r_{i}, to simulate visible distance. The value of a is chosen to be around two to achieve non-linear decay of the congestion's influence from r_{i} to r_{o}, with a curvature that lies between a linear ($a = 1$, i.e. mildest avoidance) and a step function (very large a, gives sharpest avoidance within any visible distance). We found that $a \approx 2$ achieves this effect in a visually realistic way for congestions.

Since agents *within the congestion area* are not to be tagged as dead, or immobilized, we allow these agents to continue to push through the bottleneck. Only agents who were approaching this congestion, but are *not located within the congestion area*, are repelled, thus simulating congestion avoidance behavior.

6.4 Evaluation of Congestion Avoidance

In the map shown in Figure 10(a), the agents are initially located behind the two exits, marked A and B. They all have the same goal (at the bottom right), but they have two choices on the path to their goal: the narrower exit A that is directly en-route to their goal, and a much wider exit B that requires a short detour. Since the path-plan layer does not take agent densities and potential congestions into account, the agents will initially prefer exit A.

However, when the narrowness of exit A forces a congestion to occur, the agents will eventually avoid this congestion and utilize exit B.

One aspect that is not immediately clear is what happens when the congestion areas of different bottlenecks overlap. Agents will be counted as contributing to the densities of all such bottlenecks, which could throw the congestion identification off-track, possibly identifying congestions where none exist. In Figure 10(a), the exits A and B are chosen to be close to each other, precisely to make this happen, in order to evaluate the impact of congestion area overlap on the avoidance behavior.

Figure 10(b) shows a plot of the number of agents within the congestion area of each of exits A and B, against time. The thresholds T_{HI} and T_{LO} used for this experiment are also shown. Although exit A becomes congested periodically, we see that the exit B is congested only once in the beginning, even though visually it is not congested. At other times, the green curve (crowd size at bottleneck B) is always below T_{HI}, meaning that the bottleneck is wide enough to accommodate the crowd that periodically accesses it whenever A is congested. The minor visual aberration of the crowd at B, between 120 and 330 time units in Figure 10(b), is a result of the overlap of crowd areas between A and B, and this verifies that the overlap can indeed have a (possibly brief) impact. Note that this problem can be easily fixed by making T_{HI} and T_{LO} functions of the *bottleneck size*. However, automatically assessing the size of a bottleneck is currently beyond the capability of our system.

7. Evaluation of Scalability

In order to establish the scalability of the layered approach, we have run experiments on a fixed grid of size 724 × 724, populated with varying numbers of agents. In each case, the area to be populated was chosen on the same map, and the density of the crowd was kept roughly fixed to isolate the effect of separation dynamics on the relative frame rates. The resulting number of agents are seldom round figures. Figure 11 shows the frame rates as a function of the sizes of the crowd, run on an HP Pavilion 1.6 GHz laptop with 2 GB RAM, running Windows XP.

The frame rates in Figure 11 pertain to only the simulation, *without* the graphic rendering, i.e. these figures show the number of times the position update loop completes per second, for the given number of agents, in the domain of Figure 2. Each crowd size was used in 10 independent experiments and the averages are reported. The standard errors are low in all cases. Figure 11 only shows frame rates that are at least 14 frames per second. For real-time visualization, a frame rate of 30–60 is required, which means that the said machine can handle crowd sizes of roughly 10,000 to 20,000 agents, producing real-time movements. With more sophisticated hardware, it is possible to support much larger crowd sizes in our framework.

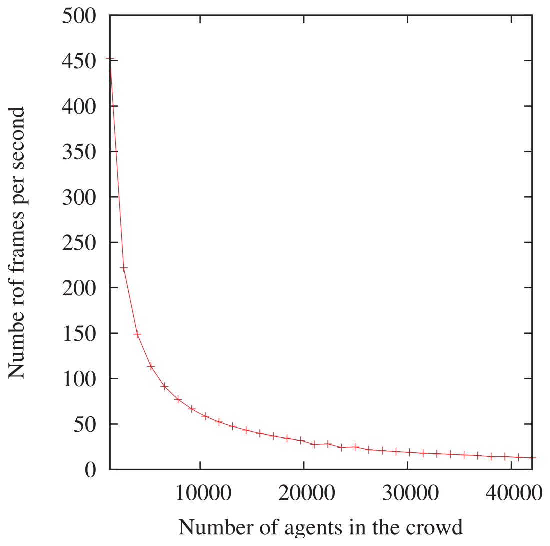


Figure 11. Plot of frame rate against the crowd size

8. Related Work

In order to face the complex challenges that crowd simulation poses, primarily three approaches[4] have been used traditionally. One approach assumes that the individuals are passive entities that drift in the presence of forces: the so-called 'social forces' model [1] and the associated variants of the gaskinetic model proposed by Helbing. This model has been extended to include further individual-level details such as familial ties and altruism [4]. The idea of social forces is similar to our use of various layers imposing 'forces' and the compound forces guiding each agent. However, our approach is distinctly different from ordinary differential equation (ODE)-based methods, is discrete, and relies on the simplicity of the process of composing forces in a distributed fashion.

The other approach is an agent-based model where individuals are modeled as intelligent agents with (limited) perception and decision-making capabilities. Some of the earliest applications of simple agent-based behaviors were seen in Reynolds' flocking model: the 'boids' [16]. In this and related works, each agent is endowed with a mix of simple steering behaviors, that produce complex macroscopic (group-level) behaviors as *emergent phenomena*. The basic idea of emergent behaviors has been extended to rule-based systems [2, 3] that offer the added advantages of efficiency and variety in behaviors. Although our approach is also agent based, we prefer a thin client where the intelligence emerges from the interaction of the agents with their environment. We do observe emergent group

4. We do not discuss user-guided or scripted crowd behaviors since we are interested in autonomous crowd movement.

behaviors that are realistic, similar to other agent-based approaches, but our approach is scalable with the number of agents, in contrast to most other agent-based systems.

Thalmann et al. [17] proposed to control agents having the same goal as a unit, for efficiency. Although we allow agents to have the same goals, we control them in a truly distributed way. However, the collision avoidance behavior emerging from our framework (when the influence field in the occupancy layer is limited, as in Equation (2)) is more rudimentary than many existing systems. For instance, Rymill and Dodgson [10] describe a system that exploits psychological studies in crowd behaviors relating to collision avoidance and overtaking, to simulate believable behaviors. Although overtaking at this level of fidelity is beyond the scope of our system, more realistic collision avoidance is possible by expanding the field of influence in the occupancy layer, albeit at the cost of decreased frame rate, owing to the greater overhead per agent. Shao and Terzopoulos [9] have integrated motor, perceptual, behavioral and cognitive components through an *artificial-life* approach, to simulate autonomous agents in a large urban setting. While our framework lacks the same level of fidelity, it is able to accommodate an unprecedented level of fidelity for the number of agents it can support. A head-to-head comparison, shows a frame rate of 12.3 for 500 agents for [9] while we can support well over 25,000 agents at the same frame rate.

With respect to integrating agents within a distributed simulation, some work has been done on multi-agent systems (MASs) using the High Level Architecture (HLA) [18]. As for layered approaches, a layering of the social interaction on environmental simulation was proposed to model interaction between humans and natural environments [19]. The problem of congestion has also been addressed with respect to amusement parks by using 'social coordination' to reduce the time wasted in congestion [20]. The issue of time management in MAS was discussed to alleviate some the problems that exist in the simulation by providing 'semantic duration models' to help developers [21].

Cellular automata [5, 22] underlie the third major approach, with recent improvements [23] for pedestrian room evacuation, similar to our evaluation domain. In contrast to this approach, our method does not require the user to identify 'exits', and involves a much simpler agent model, leading to computational efficiency.

9. Conclusion

We have presented an extension of the layered intelligence technique that is popular in the game industry, for scalable crowd simulation. We have shown how several navigation behaviors can be implemented efficiently in this framework. The chief advantage of this framework is extendability, where new behaviors can be added by adding separate layers, without affecting the existing layers. We have empirically shown the frame rates to be sufficient to handle large crowds in real-time.

We have identified several aspects where this simulation system can be improved. In particular, several dynamic group behaviors can be simulated in this framework, in addition to dynamic obstacles and the dynamically modified occupancy layer. For instance, group movements such as agents belonging to the same family, where agents tend to remain close, can be simulated by creating an additional filter[5] for each family to create a sink (or trough) of values sloping in toward the centroid of the locations of the family members. This will prevent the members of the same family from dispersing, while seeking their common goal. Similarly, leader–follower behavior can be addressed by creating an ellipsoid filter attached to the leader, and sloping in toward the major axis of the ellipsoid.

Another future direction is to extend our framework to automatically select the maximal set of behaviors that can be handled at a minimal frame rate, based on an assessment of the available computational resources.

10. Acknowledgements

We would like to thank the anonymous reviewers for useful and insightful comments. This work was supported in part by startup grants from the University of Southern Mississippi. A part of this work was also supported by a Department of Homeland Security grant (BOA:4200000225, Task Order 263894), managed by the Oak Ridge National Lab through the SERRI program. The ideas and opinions in this work are of the authors, and do not reflect those of any of the supporting organizations.

11. References

[1] Helbing, D. and P. Molnar. 1995. Social force model for pedestrian dynamics. *Physical Review E*, 51: 42–82.
[2] Ulicny, B. and D. Thalmann. 2002. Towards interactive real-time crowd behavior simulation. *Computer Graphics Forum*, 21(4): 767–775.
[3] Pan, X., C.S. Han, K. Dauber and K.H. Law. 2005. A multi-agent based framework for simulating human and social behaviors during emergency evacuations. In *Social Intelligence Design*, Stanford University, March 2005.
[4] Braun, A., S.R. Musse, L.P.L. de Oliveira and B.E.J. Bodmann. 2003. Modeling individual behaviors in crowd simulation. In *Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA)*, IEEE Computer Society, Los Alamitos, CA, pp. 143–148.
[5] Fukui, M. and Y. Ishibashi. 1999. Self-organized phase transitions in CA-models for pedestrians. *Journal of the Physical Society of Japan*, 8: 2861–2863.
[6] Sung, M., M. Gleicher and S. Chenney 2004. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3): 519–528.

5. As mentioned in section 5.1, a filter is more appropriate than an entire layer, when a behavior is limited to a small region.

[7] Tozour, P. *AI Game Programming Wisdom*, volume 2, chapter Using Spatial Database for Runtime Spatial Analysis, pages 381–390. Charles River Media, 2004.

[8] Sutton, R. and A.G. Barto. 1998. *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA.

[9] Shao, W. and D. Terzopoulos. 2005. Autonomous pedestrians. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation 2005*, ACM Press, New York.

[10] Rymill, S.J. and N.A. Dodgson. 2005. A psychologically based simulation of human behavior. In *Eurographics UK Theory and Practice of Computer Graphics*.

[11] Sucar, L.E. 2007. Parallel Markov decision processes. In *Advances in Probabilistic Graphical Models*, Vol. 214, Springer, Berlin, pp. 295–309.

[12] Stentz, A. 1995. The focused D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

[13] Banerjee, B., A. Abukmail and L. Kraemer. 2008. Advancing the layered approach to agent-based crowd simulation. In *Proceedings of the 22nd ACM/IEEE/SCS Workshop on the Principles of Advanced and Distributed Simulation (PADS)*, Rome, Italy, pp. 185–192.

[14] Bresenham, J.E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1): 25–30.

[15] Gwynne, S., E.R. Galea, P.J. Lawrence and L. Filippidis. 1999. Adaptive decision making in buildingEXODUS in response to exit congestion. In *Proceedings of the 6th International Symposium IAFSS*, Poitiers, France, pp. 1041–1052.

[16] Reynolds, C. 1987 Flocks, herds and schools: A distrtibuted behavior model. In *Proceedings of ACM SIGGRAPH 1987*.

[17] Thalmann, D., S.R. Musse and M. Kallmann. 2000. From individual human agents to crowds. In *Informatik/Informatique—Revue des organizations suisses d'informatique*.

[18] Wang, F., S.J. Turner and L. Wang. 2005. Agent communication in distributed simulations. In P. Davidsson et al. (Eds), *MABS 2004* (*Lecture Notes in Artificial Intelligence*, Vol. 3415), Springer, Berlin, pp. 11–24, 2005.

[19] Torii, D., T. Ishida, S. Bonneaud and A. Drogoul. 2005. Layering social interaction scenarios on environmental simulations. In P. Davidsson et al. (Eds), *MABS 2004* (*Lecture Notes in Artificial Intelligence*, Vol. 3415), Springer, Berlin, pp. 78–88.

[20] Miyashita, K. 2005. Asap: agent-based simulator for amusement park—toward eluding social congestions through ubiquitous scheduling. In P. Davidsson et al. (Eds), *MABS 2004* (*Lecture Notes in Artificial Intelligence*, Vol. 3415), Springer, Berlin, pp. 195–209.

[21] Helleboogh, A., T. Holvoet, D. Weyns and Y. Berbers. 2005. Extending time management support for multi-agent systems. In P. Davidsson et al. (Eds), *MABS 2004* (*Lecture Notes in Artificial Intelligence*, Vol. 3415), Springer, Berlin, pp. 37–48.

[22] Bandini, S., M.L. Federici, S. Manzoni and G. Vizzari. 2007. Pedestrian and crowd dynamics simulation: testing SCA on paradigmatic cases of emerging coordination in negative interaction conditions. In *Parallel Computing Technologies*, Springer, Berlin, pp. 360–369.

[23] Gudowski, B. and J. Was. 2007. Some criteria of making decisions in pedestrian evacuation algorithms. In *Proceedings of the 6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM'07)*, IEEE Press, Piscataway, NJ.

Bikramjit Banerjee *is an Assistant Professor in the School of Computing, the University of Southern Mississippi. He has earned a PhD in Computer Science from Tulane University in 2006, with a graduate research excellence award from the Tulane School of Engineering in 2004. His research interests are in many areas of artificial intelligence, particularly multiagent systems and machine learning, in which he has published over 30 publications. His current research projects are funded by the Department of Homeland Security and NASA. For more information, visit his webpage at http://orca.st.usm.edu/∼banerjee.*

Ahmed Abukmail *is an Assistant Professor in the School of Computing, at the University of Southern Mississippi. He has earned his PhD in Computer Engineering from the University of Florida in 2005. His research interests include parallel and distributed simulation, mobile and pervasive computing, and bioinformatics. His webpage can be found at http://orca.st.usm.edu/∼ahmed.*

Landon Kraemer *is an MSc student in the School of Computing at the University of Southern Mississippi. He earned a BSc degree in computer science in 2008. His research interests include multi-agent learning, plan recognition, and related areas of artificial intelligence.*