# CrowdMixer: Multiple Agent Types in Situation-Based Crowd Simulations

**Shannon Blyth and Howard J. Hamilton**

Department of Computer Science
University of Regina, Regina, SK, Canada S4S 0A2
{blyth1sh, hamilton}@cs.uregina.ca

**Abstract**

This paper presents a scalable approach to crowd simulation that can generate complex and varied simulations by using multiple types of individuals in a crowd. Efficiency is attained by using a situation-based approach where an individual agent adopts behaviors according to its situation, which corresponds to a subspace of the universe.

## Introduction

We propose the CrowdMixer framework for creating crowd simulations with multiple types of agents. A *crowd simulation* is a visual presentation of the behavior of so many individuals that their aggregate behavior can be perceived. Crowd simulations are often used in feature films and computer games to create scenes that include huge numbers of individuals. Also, for potentially dangerous situations, crowd simulations can show the potential reactions of people, such as how they would evacuate an area (Motta Barros 2004).

A major problem when simulating crowds is balancing the simplicity of the cognitive model of the individuals with the believability of their actions in the crowd. In real life, each member of a crowd has a unique cognitive process, causing the individual to perform certain actions in a given situation. However, when evaluating the effectiveness of a crowd simulation, more emphasis is placed on the behavior of the crowd as a whole than that of the individuals. Thus, the simulated cognitive process for each individual in a crowd can be greatly simplified.

The *single-type crowd simulation problem* is to create a crowd simulation where the procedure used to simulate the cognitive process is the same for all members of the crowd. For example, all members of the crowd may be able to perform the same actions and have the same likelihood of performing each action in a particular situation. Each member can be treated as anonymous in the crowd (Sung et al. 2004).

The *multiple-type crowd simulation problem*, which is the focus of this paper, is to create a crowd simulation where the procedures used to simulate the cognitive processes are different for several distinguished types of members of the crowd. For example, if police are trying to control a riot, the actions of a police officer are different from those of a rioter. Each type of individual should have a separate behavior pattern from other types. Each member can be treated as anonymous within its type.

In this paper, agent types are incorporated into a probabilistic situation-based framework. In the resulting crowd simulations, individual agents act differently according to their current situation, their agent type, and randomly computed probabilities. The essence of the method is to use different probability models for different types of agents and to adjust these probability models as agents enter and leave situations.

The remainder of this document is organized as follows. First, related approaches are discussed. Next, an overview of the CrowdMixer approach is given. The results of an evaluation are then discussed. Lastly, conclusions and suggestions for future work are given.



## Related Work

The key to simulating a crowd effectively is to reduce the amount of information needed for each agent in the crowd, while maintaining a realistic simulation of the crowd's actions. The *cognitive model* for an agent determines how much information it stores, how much it can learn, and how it manipulates this information.

In early work in crowd simulation, a *rule-based model* was used. The same set of simple rules is used for every member of the crowd. For example, *boids* show bird-like flocking behavior, because their rules tell them to avoid collisions with other members of the flock, to stay near the centre of the flock, and to move at a velocity similar to that of the rest of the flock Reynolds (1987). The rule-based approach is fast and scales well to crowds with many members.

With a *finite state machine* (*FSM*) *model*, a FSM is used to control the behavior of each agent. A FSM is a weak cognitive model, because it does not allow for learning or probabilistic behavior. To represent the cognitive model for all agents in a simulation, Motta Barros (2004) use a single decision tree and several finite state machines in PetroSim. Multiple agent types, in particular leader and follower, are defined in

PetroSim. To select an action, the combination of agent type and situation are used to control the traversal of the decision tree. In the tree, each leaf node is a FSM for a specific agent type in a specific situation. Using this FSM, an action is selected deterministically.

In a *situation-based model*, information relevant to particular situations is attached to the environment, rather than the agents (Farenc et al. 1999). For environments with many agents and few situations, this approach stores less information than otherwise would be necessary. Information is attached or removed from an agent when it enters or leaves a situation. In the situation-based approach of Sung et al. (2004), only one type of agent is modelled. The behavior of an agent in a situation is controlled by a probability distribution for the actions in this model. The probability distribution and computed random numbers determine which action an agent takes at any given time. As well, the behavior is affected in an unspecified manner by readings obtained from four types of sensors. The types are: *empty sensor*, which senses a lack of neighbors, *proximity sensor*, which senses a point of interest, *signal sensor*, which senses a signal received from the environment, and *agent sensor*, which senses the nearest neighbor.

A relatively complex *cognitive model with learning* has been proposed by Funge (1999). The agents are capable of learning new information from their environment. The agents use this information to make decisions when planning actions. The complexity of the model increases with that of the environment. Simulating this cognitive model consumes more resources than the situation based model, because all agents must be frequently informed of many facts. Thus, the situation-based approach is more scalable than Funge's approach.

## The CrowdMixer Framework

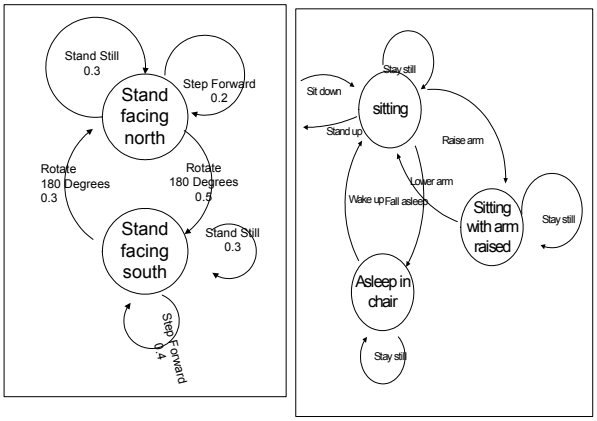
The CrowdMixer framework for creating multi-type crowd simulations is now described. Its principal features are animation transition graphs, multiple agent types, sensors, sensor effects, and inclinations.

### Animation Transition Graphs

In CrowdMixer, each agent is represented by an animation transition graph and a current state in this graph. The graph describes all possible states for the agent in the current situation as well as all actions that can be performed to move from one state to another.

A *animation transition graph* (or more simply a *transition graph*) is a labelled directed graph where each node represents a state for an agent, corresponding to a frame in the animation where a seamless transition between one action and another may be performed, and each edge represents the agent performing an action, corresponding to displaying an animated clip of the agent performing the action. Each edge is labelled with the probability of an agent in a state selecting the action represented by the edge. The sum of the probabilities on all outgoing edges from a state is 1. Whenever an agent is in a state, a transition is selected based on a random sampling of the distribution generated by the probabilities on the outgoing edges from that state.

The transition graph in Figure 1(a) has two states (*Stand facing north* and *Stand facing south*), each with three outgoing edges corresponding to the three actions that can be performed. For the *Stand facing north* state, the actions and their probabilities are *Step forward* (0.3), *Stand still* (0.2), and *Turn 180 degrees* (0.5).




**(a) Default Graph** **(b) Subgraph**

**Figure 1: Transition Graph**

Given a probability distribution, a random number between zero and one is generated to determine the next action for an agent. The possible actions for a state are mapped to distinct subintervals of [0,1]; for example, for the *Stand facing north* state the subintervals are [0,0.3) for *Step forward*, [0.2, 0.5) for *Stand still*, and [0.5, 1] for *Turn 180 degrees*. When a random number is generated, it is mapped directly to an action. For example, if 0.368 is generated, *Stand still* is selected.

### Situations

Since CrowdMixer is a situation-based approach, information is attached to the environment to control the actions of the agents. This information is in the form of situations similar to those used by Sung et al. (2004). However, in CrowdMixer, a *situation* is a subspace of the universe available for visualization, rather than an arbitrarily specified concept. When situations overlap, an area with a distinct combination of situations is called a *zone*. In the CrowdMixer implementation, a set of inclinations is attached to every object representing a situation, while Sung et al. have a set of "behaviors," which seem to serve the same purpose. When an agent enters a situation, the

inclinations for that situation are attached to the agent. An *inclination* specifies an adjustment to be made to the probability distributions of the actions that an agent can perform. In CrowdMixer, inclinations are affected by the type of the agent, while Sung et al. have only a single type of agent. When multiple inclinations affect an agent, their overall effect is calculated by composing the situations, as discussed later in this section.

Where situations overlap, they too must be composed. This composition is done by identifying all the inclinations of the situations and then composing these inclinations. Thus, all inclinations available in all situations that apply to an agent are composed.

Each situation also contains a list of sensors, which is added to any agent when it enters the situation, and a subgraph, which is added to the agent's transition graph. For example, the subgraph for the *Classroom* situation, where a student is sitting in class, is shown in Figure 1(b). An agent in this situation may be sitting, asleep in a chair, or sitting with an arm raised.

When an agent enters a situation, the relevant inclinations for the situation are added to the agent's transition graph. For example, if an agent enters the classroom situation, the graph in Figure 1(a), which represents the default transition graph for an agent, is merged with the subgraph in Figure 1(b), which represents the addition for the *Classroom* situation. To allow merging, the subgraph has extra transitions; e.g., the *Sit down* and *Stand up* transitions are provided to connect the *Stand facing north* state in the main graph with the *Sitting* state in the sub-graph. The merged graph is shown in Figure 2.

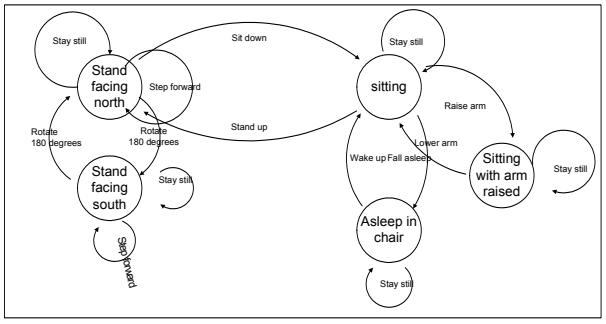



**Figure 2: Merged Transition Graphs**

## Agent Types

In CrowdMixer, each agent type is defined by a prototype, which specifies (1) all possible states and transitions that can be included in the transition graph for an agent of this type in any situation, and (2) the default transition graph for an agent of this type. When an agent is added to a scene, it is given the default transition graph for its type and a randomly chosen initial state in this transition graph.

Different inclinations can apply to different agent types. The same inclinations can have effects of varying magnitudes on different agent types. The probability of actions can also vary for different agent types.

## Sensors

A *sensor* passes environmental data to an agent. Data from sensors is used in generating the probabilities for the transition graph. Each kind of sensor can return either a floating-point value called NumericValue or a Boolean value called BooleanValue, as needed. The following six kinds of sensors are used.

***Proximity***:

NumericValue - Returns the difference between the distance to a point after a specified action has been performed and the current distance to the same point.

BooleanValue - Determines if an action will take the agent farther from a designated 2-dimensional coordinate.

***Orientation***:

NumericValue - Finds the distance between the unit vectors defined by the agent and a designated 2-dimensional coordinate, and the agent and the location the agent will be at if a given action is chosen.

BooleanValue - Determines whether the numeric value returned is less than a specified value.

***Empty***:

NumericValue - Returns 1 if there are agents in the area, 0 otherwise.

BooleanValue - Determines if there are any agents within a designated area.

***Signal***:

NumericValue - Returns 1 if the environmental signal is set, 0 otherwise.

BooleanValue – Returns a Boolean environmental signal.

***Nearest***:

NumericValue - Finds the distance between the agent and the other agent that is nearest to it.

BooleanValue - Determines whether the distance is less than a specified value.

***Nearest of type***:

NumericValue - Finds the distance between the agent and another agent of a specified type that is nearest.

BooleanValue - Determines whether the distance is less than a specified value.

The *Proximity*, *Empty*, *Signal*, and *Nearest* sensors are similar to the *Proximity*, *Empty*, *Signal*, and *Agent* sensors in Sung et al. (2004). *Orientation* sensors allow the agent to sense its orientation in relation to a point in the world. *Nearest of type* sensors are useful for dealing with multiple agent types.

## Sensor Effects

A *sensor effect* is a description of the effect a sensor reading will have on the probability distribution for determining an agent's actions. It is implemented as a data structure storing the sensor name, an action, the kind of effect, and the effect's magnitude. Sensor

effects are used when calculating the probability of an action in an inclination.

The *Center Proximity* sensor effect is as follows.

SensorName: *Center Proximity*
ActionName: *Move forward*
EffectType: *Direct Boolean*
Magnitude: 3.0

Four kinds of effect are provided: *direct numeric*, *inverse numeric*, *direct Boolean*, and *inverse Boolean*. A *numeric effect* makes sense in relation to *Proximity*, *Orientation*, *Nearest*, and *Nearest of type* sensors. These sensors return real numbers representing distances. A *direct numeric effect* makes actions that return higher values more likely to occur, whereas an *inverse numeric effect* makes such actions less likely. The value is determined by adding or subtracting to the value returned by the evaluate function, resulting in higher or lower probabilities of choosing an action. A *Boolean effect* is based on the Boolean values returned by the sensor. The strength of the sensor effect is proportional to the magnitude. Magnitude specification is helpful when it is desired that different sensor effects have different magnitudes.

## Inclinations

An *inclination* implements one high-level tendency for an agent by adjusting the probabilities of the possible actions for the agent. An inclination is implemented as a list of sensor effects that are to be applied when the inclination is in effect. For example, the *Right proximity* inclination includes a sensor effect that has a positive effect on rightward motion and a set of sensor effects that have negative effects on every other type of motion. In a given situation, multiple inclination functions may be combined to determine a probability distribution for how likely the agent is to perform an action. Inclinations are composed by combining the values returned from the inclinations.

Since each agent type has a set of available actions in a situation, the available transitions between states may differ for each agent type. When providing input for evaluation of the inclination, these actions are passed as input. Since each agent provides the set of actions that it is able to perform as input to the inclination, the inclination does not need to consider the agent type.

## Evaluating Inclinations

The algorithm that performs one time step of CrowdMixer for one agent is shown in Figure 3. The algorithm calls the EvaluateSensorEffects function, which returns a real number. A sensor effect associates a sensor with an action, such that the value stored in the sensor has an effect on the EvaluateSensorEffects function's result for that action.

Figure 4 presents the EvaluateSensorEffects function used to apply the effect of a sensor to the probability of a specific action for an agent. It generates a real number v based on input from the environment. Initially, v is set to a negative value relative to the α value used in the sigmoid function. By setting v in this way, the function effectively scales the values so that states unchanged by data from the sensors will have values that fit with those affected by the sensor data. After setting a default value for v, all of the agent's sensors are checked. If any sensor is used by a sensor effect that is applied to the input state, v is modified according to the effect type. The magnitude of the modification is determined using the magnitude specified by the user for the sensor effect. The sum of the default return value and all sensor effects that apply to the input state is accumulated.

```
Algorithm OneTimeStep
    for each possible next action n for agent a in current state s_a:
        Initialize P_a(n) based on the transition graph for agent a
        for each inclination i for agent a:
            P_a(n) = Compose(P_a(n), EvaluateSensorEffects (a, n, i, α))
    Normalize the values in P_a to sum to 1
    Build interval table T for probability distribution P_a
    N_a = MapToAction(T, random(0,1))
    DisplayAction(a, N_a)
    set current state s_a for agent a to Transition(s_a, N_a)
```

**Figure 3: OneTimeStep Algorithm**

```
Evaluate (agent a, action n, inclination i, float α)
  v = -1.0 / α
  for each sensor s in agent a
    for each sensor effect e in inclination i
      if e.SensorName = s and e.ActionName = n
        switch (e.EffectType)
          case (Direct Numeric):
              v =  v + e.Magnitude * s.NumericValue
          case (Inverse Numeric):
              v = v – e.Magnitude * s.NumericValue
          case (Direct Boolean):
              if s.BooleanValue
                  v = v + e.Magnitude
          case (Inverse Boolean):
              if s.BooleanValue
                  v = v – e.Magnitude
  return sigmoid(v, α)
```

**Figure 4: Evaluate SensorEffectsFunction**

The sigmoid function is used to apply scaling to the sum v to make the value fit in the range [0,1). Scaling is done using a sigmoid function with a parameter α to modify the steepness of the curve. Increasing α makes the curve steeper. Steep curves in the sigmoid function cause values close to zero to have a lot of variance. The result of the sigmoid function is returned by the EvaluateSensorEffects function.

For example, suppose we have an inclination with α = 2.0, and the following sensor effect.

Sensor Name: *Center proximity*
Action Name: *Move forward*
Sensor Effect Type: *Direct Boolean*
Magnitude: 3.0

To determine the probability that an agent will perform the *Move forward* action, given that the agent has a *Center proximity* sensor in its virtual memory, v is calculated: v = -1.0 / α = -1.0 / 2.0 = -0.5. Since *Center proximity* has a direct Boolean effect, and assuming that the sensor returns *true* as its Boolean value, v is adjusted by adding the magnitude of the effect: v = v + e.Magnitude = -0.5 + 3.0 = 2.5.

If the previously described sensor effect is the only effect for the inclination, and the agent has possible actions *Move forward*, *Move backward*, and *Turn 180 degrees*, the probability distribution for the agent at a specific point in time is as shown in Table 1. This table indicates that *Move forward* has a probability of being selected of 65%, whereas each other action has a probability of approximately 18%.

| | *Move forward* | *Move backward* | *Turn 180 degrees* |
|---|---|---|---|
| Value of v | 2.5 | -0.5 | -0.5 |
| Value after Sigmoid applied | 0.9933 | 0.2689 | 0.2689 |
| Probability After Normalization | 0.6487 | 0.1756 | 0.1756 |

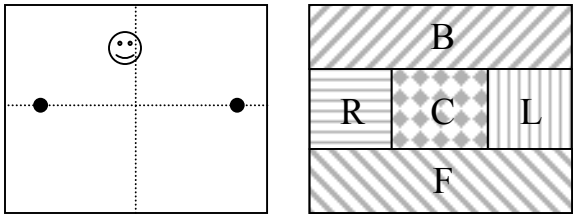
**Table 1: Results of Probability Calculations**

## Results

The CrowdMixer program generates 2-dimensional simulations of crowds consisting of multiple types of agents using a situation-based approach. We ran four tests to investigate (1) composition of inclinations, (2) composition of situations, (3) multiple agents, and (4) combinations of multiple agents, composed situations, and composed inclinations.

### Test 1 Composition of Inclinations

Test 1 checked the composition of inclinations. Two similar inclinations, *Right proximity* and *Left proximity*, are each trying to move an agent, indicated by a happy face at (-0.1, 0.3), away from point (0.5, 0) and point (-0.5, 0), respectively. The agent has four possible actions corresponding to movements of 0.1 in four directions, left, right, forward (to top), and backward.




(a) Test 1 (b) Test 2

**Figure 5: Environments for Test 1 and 2**

Figure 5(a) shows the agent's location and the points to avoid. Ignoring the inclinations, the initial probability distribution is as shown in the first line of Table 2. The distributions of individual and composed inclinations are shown in the remainder of Table 2. Each row sums to 1. The results show that when the inclinations are composed, the agent is most likely to move forward, because it is trying to avoid points to both left and right. Where only *Right proximity* is used, the agent prefers to move left or forward, and where only *Left proximity* is used, right or forward.

| | Action | | | |
|---|---|---|---|---|
| Inclination | *Backward* | *Forward* | *Left* | *Right* |
| None | 0.2500 | 0.2500 | 0.2500 | 0.2500 |
| *Right proximity* | 0.1330 | 0.3404 | 0.4570 | 0.0697 |
| *Left proximity* | 0.1094 | 0.3800 | 0.0809 | 0.4297 |
| Composed | 0.0398 | 0.6512 | 0.1797 | 0.1292 |

**Table 2: Probability Distributions for Test 1**

### Test 2 Composition of Situations

Test 2 checked the composition of situations using the same type of agent as Test 1. Four situations were used, each corresponding to two zones in Figure 5(b), namely *B* + *C*, *F* + *C*, *L* + *C*, and *R* + *C*. Situation *B* + *C* provided the agent with the *Backward* action, so in the *B* zone, the agent could only perform the *Backward* action. Similarly, in *F*, *L*, and *R* zones, the agent could only move forward, left, and right, respectively. In the *C* zone, the situations were composed, and the agent could perform all four actions. The probability distributions for every zone are shown in Table 3. In the *B*, *F*, *L*, and *R* zones, only one action is available to the agent. In the *C* zone, the agent could move in all four directions with equal probability.

| Zone \ Action | *Backward* | *Forward* | *Left* | *Right* |
|---|---|---|---|---|
| *B* | 1.0 | N/A | N/A | N/A |
| *F* | N/A | 1.0 | N/A | N/A |
| *L* | N/A | N/A | 1.0 | N/A |
| *R* | N/A | N/A | N/A | 1.0 |
| Composed (*C*) | 0.25 | 0.25 | 0.25 | 0.25 |

**Table 3: Probability Distributions for Test 2**

### Test 3 Multiple Agents in a Simulation

To test multiple agent types within a crowd simulation, the situation composition test was expanded upon. Zones were added where two situations overlapped. The agent types used in the simulation were straight agents and diagonal agents. A *straight agent* can move in four directions, forward, backward, left, and right, in the same situations as in Test 2. A *diagonal agent* can move in four directions, up-left, up-right, down-left, and down-right. In particular situations, a diagonal agents can move up-left or up-right in the *Forward* situation, back-left or back-right in *Backward*, up-left or back-left in *Left*, and up-right or back-right in *Right*.

Nine zones were used in the simulation (see Figure 6). For testing, five agents of each type were placed in each zone. Figure 7 shows the expected and observed

probabilities for agent actions within the zone where the *Backward* and *Right* situations overlap. Overall, the difference between the expected and observed probabilities for agent actions was under four percent. Such differences are usual since the distribution is repeatedly sampled in an independent manner.


**Figure 6: Zones for testing multiple agents**

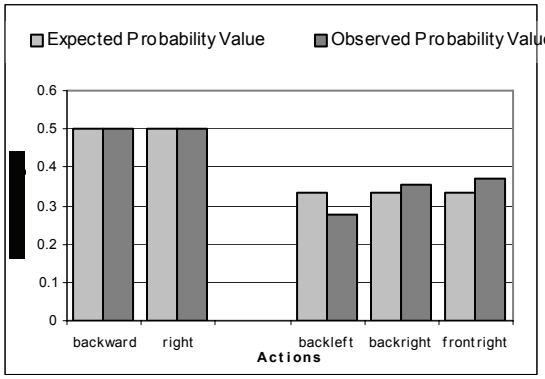



**Figure 7: Expected and Observed Compared**

## Test 4 Store Situation

A toy store was simulated to test a case where agents have multiple types with varying sets of possible actions. The two agent types are adult and child. The adults perform 13 possible actions, including getting a cart, going down an aisle, and putting an item in a cart. The children perform 11 possible actions, including playing with toys when they are in an aisle where toys are sold, falling down, and having tantrums. We used about 20 adults and 15 children in our simulations.

As shown in Figure 8, the 6 situations in the simulation are *Aisle*, *Toy aisle*, *Checkout*, *Cart pick-up*, *Cart drop-off*, *Stumbling block*, and *Move with cart*. Testing (not shown) demonstrated that composition of situations works correctly, with agents consistently performing only permitted actions in situations.

Three inclinations were used in the simulation. In the *Pickup cart* situation, the *Get cart* inclination increased the probability of an adult picking up a cart and the *Orientation to checkout* inclination helped move adult and child agents toward the aisles. In the *Move with cart* situation, the *Move to checkout* inclination also helped moved agents toward the aisles. Testing showed that these inclinations composed correctly.

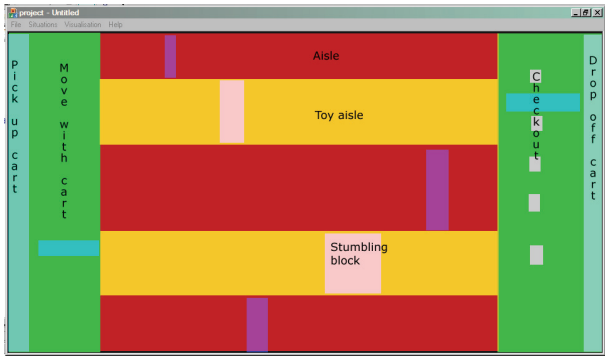



**Figure 8: Situations for shopping simulation**

## Conclusions

This paper describes the CrowdMixer situation-based framework for creating crowd simulations with multiple types of agents. With CrowdMixer, it is possible to create simulations using more than one type of agent, with agents of each type having distinct actions available to them. As simulated agents move into different spatial areas, they act appropriately for their current situations, without requiring individual cognitive simulation. An appropriate probability distribution for each situation is generated on the fly, allowing actions to be selected non-deterministically while maintaining the desired overall crowd behavior.

In the future, the interface needs to be extended to show 3-dimensional animations. As well, the effectiveness of the system in actually producing desirable crowd simulations should be demonstrated further.

## References


Farenc, N.; Boulic, R.; and Thalman, D. 1999. An Informed Environment Dedicated to the Simulation of Virtual Humans in Urban Context. In *Proceedings of EUROGRAPHICS 1999*, Vienna, Austria.

Funge, J.; Tu, X.; and Terzopoulos, D. 1999. Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters. In *Proceedings of ACM SIGGRAPH 99*, Los Angeles, CA.

Reynolds, C. 1987. Flocks, Herds, and Schools: A Distributed Behavior Model. In *Proceedings of ACM SIGGRAPH 87*, Anaheim, CA.

Motta Barros, L.; Tavares da Silva, A.; and Raupp Musse, S. 2004. PetroSim: An Architecture to Manage Virtual Crowds in Panic Situations. In *Proceedings of CASA 2004*, Geneva, Switzerland.

Sung, M.; Gleicher, M.; and Chenny, S. 2004. Scalable Behaviors for Crowd Simulation. *Proceedings of EUROGRAPHICS 2004*, Grenoble, France.