Universiteit Leiden
Opleiding Informatica

Detailed crowd simulation

and spatial hashing for large-scale collision detection

Name: Jonathan Robijn

Date: 15/08/2014

1st supervisor: dr. M. S. Lew
2nd supervisor: dr. E. M. Bakker

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Large crowds consist of many individuals who might each react differently to a given situation. We describe a crowd simulator which models a building evacuation scenario and incorporates several new types of behavior. Using these new types of behavior, we give the simulation a more lifelike impression. In addition to this, we also present a comprehensive summary of collision detection performance results for various configurations of a spatial hashing algorithm.

Contents

1 Introduction **2**

2 Related Work **4**

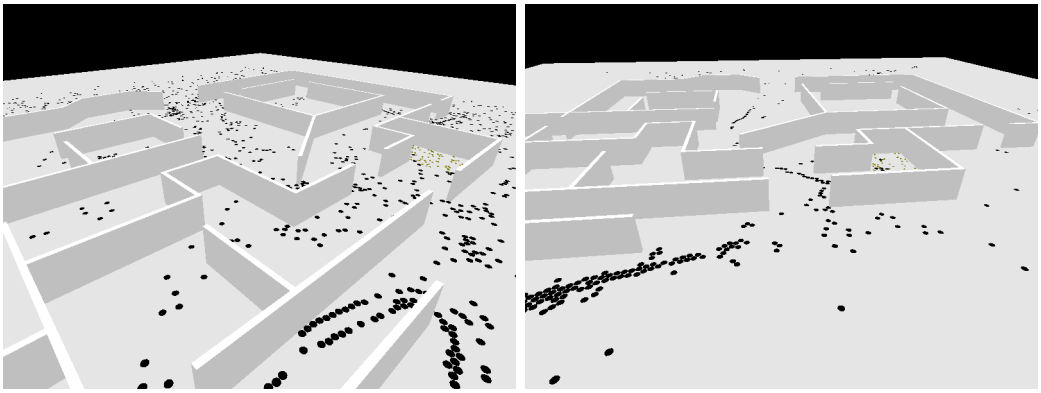
3 Basic crowd simulation **6**
3.1 Global path planner 7
3.2 Local Collision avoidance 12
3.3 Collision detection 16

4 Behavioral model **21**

5 Algorithm summary **24**

6 Results and discussion **25**
6.1 Collision detection performance 25
6.2 Behavioral features 28
6.3 Limitations and future work 32

7 Conclusion **33**



(a) **(b)**

Figure 1: *A crowd simulation in and around a building. In (a) the agents are browsing and calmly walking around. In (b) an evacuation is in progress*

1 Introduction

Human crowds are a very common phenomena in the real world, making their simulation very useful for a wide range of applications. A reasonably realistic and correct crowd model is used in virtual environments, emergency training, urban planning, education and many other areas.

The people in a crowd make their own individual decisions based on their own goals, obstacles in the area, and other agents within close proximity. The collective behavior of a human crowd is driven by these complex individual decisions and creating a model that can simulate all the intricacies of large crowds has proven to be challenging.

A good crowd model has to produce a reasonably realistic simulation which can be compared to a crowd in real life. There is a trade-off between detail and performance. A detailed model which gives incredibly realistic results has limited applications if it can't produce these results within a reasonable execution time for a larger amount of people, so the performance of the model's algorithm is very important as well.

In this paper, we describe the implementation of a crowd simulation which gives believable results for large numbers of people in real-time. To this end, we use several well-known algorithmic solutions to implement the various components of the crowd model.

Existing methods often simplify the problem by dividing it into 3 seperate components for global path planning, local path planning and collision detection. While most local path planning algorithms try to avoid collisions between agents or an agent and an obstacle, they do not guarantee all collisions are always avoided. As such, collision detection is still required as a last line of defense against graphical anomalies where people clip through walls or each other. The collision detection component has to compare the position of every person to the position of other people. A naive approach would be far too computationally expensive for this problem.

We were unable to find much previous work regarding large-scale collision detection performance for simple cases like thousands of agents moving around on a simple 2D plane. As such we will be writing our own algorithm and comparing the performance of various configurations in this paper.

To reduce the execution time of this component, we take a deeper looker at a spatial hashing solution, which allows us to hash agents to specific areas based on their position. Using such an algorithm, a list of agents in a specific area can quickly be retrieved to compare them against a specific agent. To optimize the algorithm, We tweak the spatial hashing method to determine the optimal configuration for fast collision calculations.

Another way existing methods simplify the problem of simulating a crowd is by assuming that agents have homogeneous goals. Usually, these goals can be described as moving to a specific point. In a lot of applications, this is the most important type of behavior for a crowd simulation, but more variety should be able to create more realistic results for specific applications.

Following this observation, we implement and evaluate additional types of general behavior, called *Lost* and *Panic*, which can be thought of as the agent's state of mind. As the names suggest, these new types of general behavior will be useful for evacuation simulations. In addition to these new types of behavior we also implement a new goal, called *RetrieveItem*, which is an extension of the normal *MoveToPoint* goal. After the implementation, we compare the simulation with the more complex behavioral model to the simulation with the basic behavioral model.

While there are other crowd simulators which have implemented some kind of evacuation mode, our crowd simulator encompasses some new detailed behavioral features. As far as we know, very little research has been done regarding these kind of features and we feel it is worthwhile to evaluate

the potential added value of adding more behavioral diversity.

The key results of this work can be summarized as:

- An efficient crowd simulator that can simulate thousands of agents in real-time with reasonably realistic results. (§3)
- A performance comparison between various configurations of a spatial hashing algorithm for collision detection. (§6.1)
- A more detailed behavioral model that is optimized for emergency evacuation simulations. (§4, §6.2)

2 Related Work

Crowd simulation has received increasing attention in the past 2 decades. In this section, we take a brief look at some of the previous work relating to crowd simulation.

In agent-based methods, each individual agent plans its' own movement. This is the most natural way to model a human crowd as real humans also make their own individual decisions based on their surroundings and personal goal. It is also very intuitive to extend these methods to include a more detailed cognitive model which approximates the decisions a normal human would make in similar circumstances.

The work of Funge et al [4] resulted in a simulation in which the agents have several cognitive features, such as learning and knowledge. This model was then further expanded again by Shao and Terzopoulos [16] with visibility and path planning. The result of this work is that agent-based methods can create very realistic virtual crowds that closely resemble real human crowds. However, modelling human behavior with this much accuracy carries a large computational cost per agent, severely limiting the usefulness of these methods for large-scale crowds with hundreds or thousands of agents.

An effort to simplify crowd models to address this issue has lead to the use of local models, which use simpler rules while still trying to maintain a reasonably realistic result. The use of these kind of models can be seen as early as the work of Reynolds [15] who demonstrated that noticeable grouping behavior can be generated from simple local rules. Later efforts

also added support for sociological factors [12], social forces [17], visibility and pathing [16] and many other models.

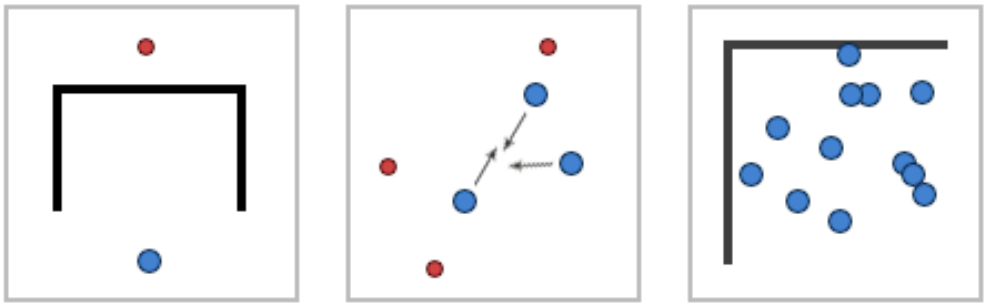
Many collision avoidance methods have been proposed for situations with agents that are close to each other. Among many others, there are geometrically-based algorithms [5, 21, 20] and force-based algorithms [7]. More recently, continuum-based models [19] and models based on fluid dynamics [13] have been developed which can efficiently handle collision avoidance for a larger number of agents while still producing results in real-time.

Most useful simulations will have some form of obstacle in them. Local models alone can generally not handle the task of choosing a path which avoids obstacles as they do not look very far ahead. The result would then be that an agent has a very high chance of "bumping" into an obstacle and unnaturally looking for a way to get to the other side and continue to its' goal. To enable agents to plan a path around any obstacles, local models are often combined with global path planners. These are often implemented by representing the free area in the "world" as a graph and then searching the graph for a path from an agent to its' goal. [1, 8, 10, 17, 18, 9, 14]

Another component that is still very much required for the simulation is the collision detection component. While most cases of potential collision are solved by the local collision avoidance models, at very high crowd densities the collision avoidance models can not always guarantee that no clipping will take place between 2 agents or an agent and an obstacle. Thus, crowd simulations that are expected to run for crowds with very high densities implement some basic form of collision detection. [19, 13]

Collision detection is a very important part of any virtual environment and because of this not all developments in this area are strictly related to crowd simulation. Vigueras et al [22] have developed a crowd simulation collision detection method which is optimized for multi-core and many-core machines. Luque et al [11] have developed a spatial partitioning algorithm for quicker neighbour selection using BSP-trees.

There are also collision detection algorithms for many specific complex problems which are not strictly related to crowd simulation, such as highly deformable tetrahedal models [3] and large amounts of polytopes [2].



(a) *Global path planning* **(b)** *Collision avoidance* **(c)** *Collision detection*

Figure 2: *The 3 main components of the crowd simulation. Blue circles are agents, red circles are their goals and black line segments are walls*

3 Basic crowd simulation

In this section, we describe the implementation of a crowd simulation which uses simple goal-based agent movement. This is a simple, yet reasonably realistic, method of defining human movement on a large scale. Simulating large amounts of agents moving simultaneously towards their own goals through a shared environment is a complex problem. Often, a crowd simulation divides this problem into various components with each component being a step in the problem's solution. A commonly used structure is as follows:

Global path planning component. Any virtual environment may contain physical obstacles such as walls which need to be avoided by the agents that are trying to navigate through the environment. A global path planning component gives the agent a map of the area and allows him to plot a path towards his goal. Without a global path planner, agents could get stuck in local minima of obstacles or at the very least have to follow an unrealistically complex path to find a way around the obstacle, as can be seen in Fig. 2a. We describe a roadmap-based implementation of this component in §3.1

Local collision avoidance component. With a functional global path planner, agents can navigate around the obstacles of an environment and reach their goals in a timely fashion. However, in a crowd simulation there are many other agents that also want to reach their goals. If the paths of 2

agents cross at some point and the agents arrive at this point at around the same time, then we need to adjust their paths, as we can see in Fig. 2b. The local collision avoidance component attempts to adjust agents' paths based on other nearby agents and its' goal is to create a smooth representation of crowd dynamics. This component has a vital role in the perceived realism of the simulation. We describe an implementation of this component in §3.2.

Collision detection component. While most collision avoidance components aim to prevent agents from coming too close to other agents, they usually can't guarantee that this does not sometimes happen. The collision detection component serves as a last validation, ensuring that at any timestep, agent positions adhere to some basic physical rules. For instance, the physical space taken by an agent is not allowed to overlap the physical space of a wall or another agent, so a situation like Fig. 2c is not allowed and needs to be adjusted. We take a closer look at this component in §3.3.

This is a general outline of how crowd simulation is often implemented. However, it is important to realise that it's entirely possible to develop a simulation which has extra components or combines some of the above into 1 hybrid component. For instance, [19] combines the global path planner and local collision avoidance into 1 component in a novel way.

In the rest of this section, we will describe our own implementation based on the above mentioned 3 components.

3.1 Global path planner

The first major component of the crowd simulation is the global path planner. If an environment does not contain any obstacles, an agent can simply move in a straight line towards his goal. However, any meaningful virtual environment will have obstacles, so a more sophisticated procedure is needed. We will represent the obstacle-free space as a graph and then use this graph as a roadmap to compute paths for agents. This is a common solution used in many other simulations (for instance, see [1])

A roadmap-based method can be described as consisting of 2 phases: a construction phase and a query phase.

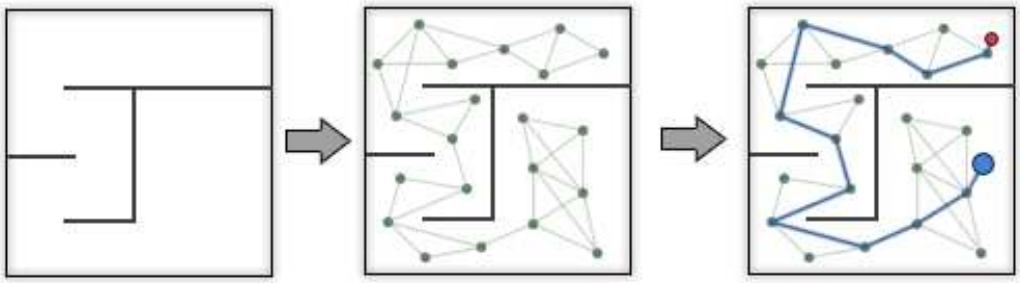
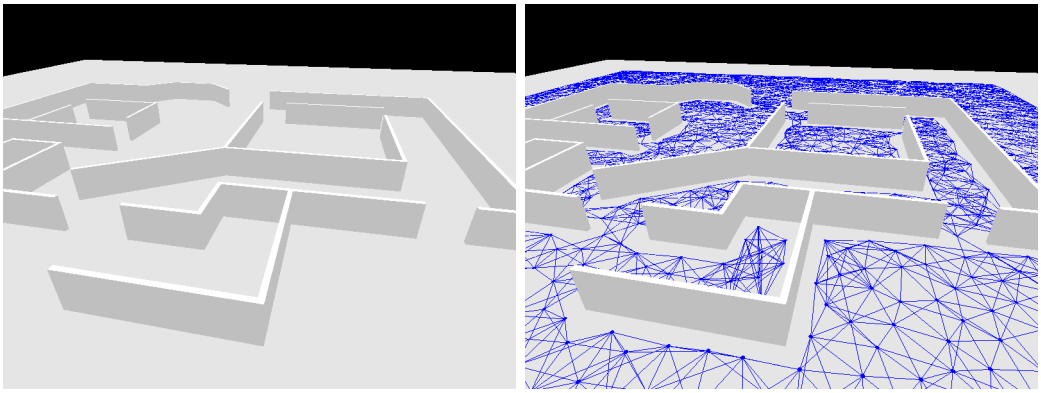


Figure 3: *An overview of our global path planning algorithm and how we use it to calculate a path for an agent.*



(a) **(b)**

Figure 4: *A building in our crowd simulation. In (a) the building is shown without a roadmap. In (b) a roadmap has been generated and is shown on the ground.*

Algorithm 1 Generate a roadmap

```
for i := 1 to N do
  while !valid_point do
    new_point ← rand_point();
    valid_point ← true;
    for j := 1 to num_walls do
      if d(wall[i], new_point) < dist_min then
        valid_point ← false; {Check distance to obstacles}
      end if
    end for
    for j := 1 to i do
      if d(node[i].point, new_point) < dist_node then
        valid_point ← false; {Check distance to placed nodes}
      end if
    end for
  end while
  node[i].point ← new_point; {Add node if point meets requirements}
end for
for i := 1 to N do
  progress ← 1;
  edges ← 0;
  while edges ≠ k and progress ≤ N do
    c ← closestNode(i, progress);
    increment progress;
    valid_edge ← true;
    for j := 1 to num_walls do
      if d(wall[i], line(node[i].point, node[c].point)) < dist_min then
        valid_edge ← false; {Check distance to obstacles}
      end if
    end for
    if valid_edge then
      increment edges;
      Add an edge to node c to set edgesFrom[i];
      Add an edge to node i to set edgesFrom[c];
    end if
  end while
end for
```

Construction phase. A graph which has to be useful as a roadmap needs to have its' nodes placed in such a way that any agent in any valid position can find an unobstructed line to a nearby node.

We use a Probablistic Roadmap Method (PRM) to construct our roadmap. The algorithm randomly samples N points. Each generated point needs to fulfill 2 requirements before it can be added as a node: It needs to be no closer than d_{min} to any obstacles and it can be no closer than d_{node} to any nodes already placed. If one of these requirements is not met, then a new point is sampled and evaluated again.

Depending on the values of N and d_{node}, this could mean the algorithm for node sampling could be an infinite loop! Care must be taken when choosing appropriate values of N and d_{node}

If N is sufficiently large and d_{node} is sufficiently small, the PRM should be able to create a graph which has reachable nodes for every possible agent position.

After we have sampled N points and converted them to graph nodes, edges can be created between the nodes. For each node, we select k nearby neighbours to which we can plot valid lines. We say that a line is valid if at any point on the line the distance to any of the obstacles is at least d_{min}. In other words, there is a radius of size d_{min} around the line in which no obstacles may be present.

For each of those k nearby neighbours to which we can plot valid lines we add an edge from the current node to the neighbour node. We also add an edge from the neighbour node back to the current node, as we are creating an undirected graph. An in-depth description of the procedure is given in Algo. 1.

We have now generated a graph which is saved as a set of nodes and a set of edges for each of those nodes. An example of a generated roadmap for some virtual environment can be seen in the second image of Fig. 3.

Query phase. Our algorithm has now constructed a graph which represents a roadmap. We can use this graph to plot a path from an agent to its' goal. In order to do this, the agent and its' goal must be temporarily connected to the graph. The easiest way to do this is by finding a nearby node for which the line between the node and the agent or goal is unobstructed by any obstacles. We find the closest node to the agent which fulfills this

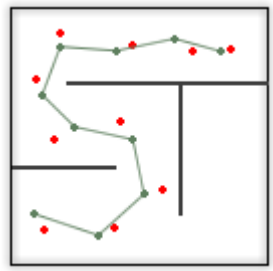


Figure 5: *A possible final calculation of an agent's path*

requirement and find the closest node to the agent's goal in the same way.

Now we have found the start and end node for the path we want to calculate. To find a path from the start node to the end node, a graph search algorithm can be used.

In our implementation we have used A*, a best-first search algorithm which takes into account both the currently travelled distance and the heuristic estimate of distance that still needs to be travelled. We use the euclidian distance between the current node and the end node as the heuristic estimate. For further information on the algorithm, we refer the reader to the original A* paper [6].

A* gives us the needed performance for thousands of agents constantly in need of path calculations to their new goals. A possible path generated using some roadmap is shown in the third image of Fig. 3

Finally, we make some final adjustments to add more variance to the path an agent walks. If we were to use the generated path as-is, then agents who calculate similar paths would make the same moves. Agents who went into opposite directions on the same graph edges would directly run into each other.

In an attempt to create a more natural system, we create a copy of the calculated path and add an element of randomness to each point in the path. Given the i-th path point p_i and a random vector $\vec{r}$ which has a length between 0 and $\frac{d_{min}}{2}$, we modify p_i as follows:

$$p_i = p_i + \vec{x} + \vec{r} \tag{1}$$

where

$$\vec{x} = \begin{cases} unit(p_i - p_{i-1}) * \frac{d_{min}}{2} & i > 0 \\ unit(p_i - startPos) * \frac{d_{min}}{2} & i = 0 \end{cases} \tag{2}$$

The result is that the newly generated point is slightly behind the old point, relative to the previous point in the path. The new point is also moved into a random direction for a certain amount. Given our definitions of $\vec{x}$ and $\vec{r}$, we know that the newly generated point will never be further away from the old point than d_{min}. Our roadmap nodes are not allowed to be closer to obstacles than d_{min}, so we know the new points still have to be inside the obstacle-free space. An example of a path generated by our procedure can be seen in Fig. 5

When a final path is calculated, the agent saves it and walks through it. The first node which hasn't yet been visited in the path is treated as the current subgoal and the agent moves towards it. A node (including the goal of the agent) is considered visited when the agent comes closer than a certain distance value. This generalization ensures that an agent stays flexible and does not have to reach the exact coordinate if the surrounding agents don't allow it. If all path nodes have been visited, the agent's subgoal is set to his actual goal. If the agent reaches his goal, a new goal is determined and a new path is generated to reach this goal. Every timestep, the global planner assigns a preferred velocity vector to each agent by generating a vector of size v_{max} which points from the agent's postion to his current subgoal.

3.2 Local Collision avoidance

The global path planner component has now returned a preferred velocity for each agent. If we were simulating a small amount of agents, we could directly use these velocities to move the agents. As the density would probably be very low, the chance of collision will also be low and moving the agents in this blind manner would still give acceptable results.

In our simulation, we want to be able to simulate thousands of agents in confined areas, so additional work is needed. A local collision avoidance component is often used to adjust agents velocities based on nearby agents, in an attempt to anticipate and prevent collision. There are several options

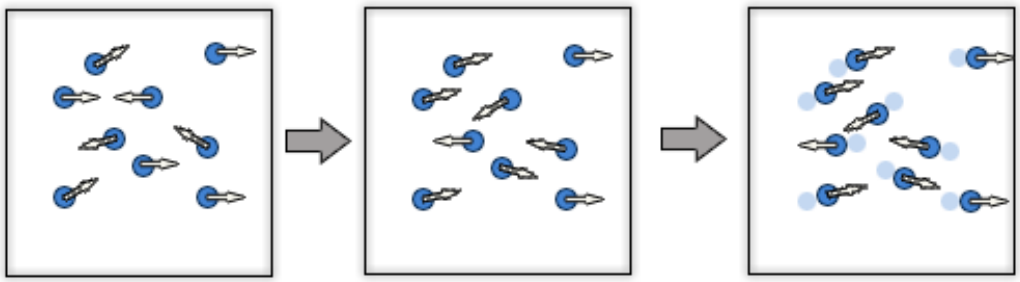
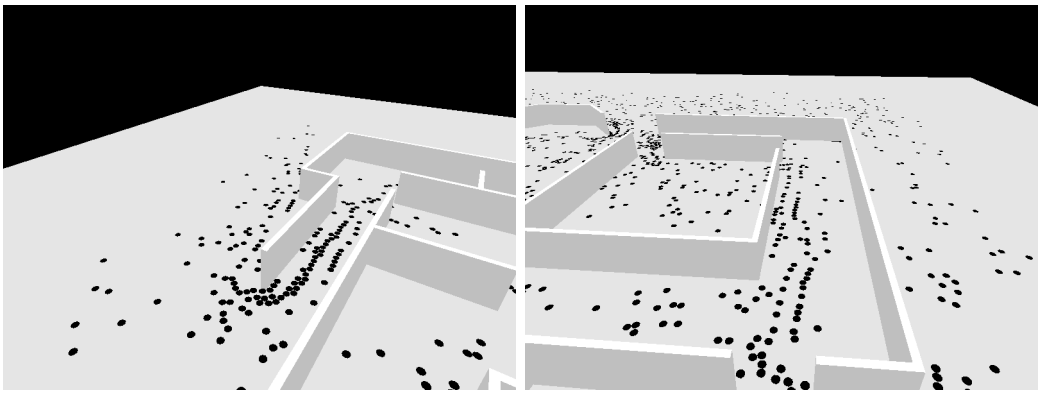


Figure 6: *An example of what our local collision avoidance component does. Adjusting the velocities and then moving the agents using these newly calculated velocities*



(a) **(b)**

Figure 7: *2 hallways near exits in our building scenario. Lanes are being formed by agents moving in opposite directions, avoiding congestion*

available to do this.

First of all, a force-based method (for instance, see [7]) can be used for the collision avoidance. This kind of method uses repulsion and attraction forces to adjust the velocity of each individual agent based on his surroundings. Crowd behavior emerges as each agent adjusts his path based on nearby other agents. This method can be easy to implement, can give realistic results and allows for intuitive implementation of more individuality among agents. However, force-based models are not known to be very effective for large scale simulations, as they require too much computation time for a larger amount of agents.

More recently, a continuum dynamics method [19] was developed which uses a dynamic potential field to handle both the global path planning and the local collision avoidance. This kind of method can handle thousands of agents at real-time framerates while still showing emergent crowd behavior among its' agents.

Even more recently, a fluid dynamics method [13] was explored which converts all agents to a continuous system and then performs all collision avoidance calculations on this system. The computational cost is decoupled from the number of agents and because of this simulations of up to a hundred thousand agents are possible.

The last 2 methods are very efficient for large scale simulations, but they are much more complex and tend to simplify agents to treat them as small parts of the much larger crowd. Because of this, adding more individuality among agents is not as straightforward and intuitive as we would like.

For simplicity, we will use a basic force-based model which includes some repulsive and attractive forces. This choice will bottleneck our system's performance for larger amounts of agents, but is simpler to implement and will allow us to intuitively add more individuality to agents in the crowd later on.

An important emergent behavior in crowds is lane forming. If 2 groups of people want to pass each other into opposite directions, they will naturally form 2 lanes next to each other. We want to be able to simulate dense crowds, so this is an essential behavior to implement into our collision avoidance component. For instance, Figure 6 shows 3 lanes forming so agents moving in similar directions can move together while avoiding agents going

in the opposite direction.

We implement a simple form of this behaviour with some simple rules:

- If a nearby agent's preferred velocity is in the same direction, the current agent is attracted to this agent
- If a nearby agent's preferred velocity is in the opposite direction, the current agent is greatly repulsed from this agent
- If a nearby agent's preferred velocity is in another direction, the current agent is repulsed from this agent

To compare the preferred velocity directions of 2 agents, we take the unit vector $\vec{v_c}$ of the current agent's preferred velocity and the unit vector $vecv_n$ of the compared neighbour agent's preferred velocity, and then calculate:

$$s = \vec{v_n} \cdot \vec{v_c} \tag{3}$$

s is essentially the scalar projection of the neighbour's preferred velocity vector onto the current agent's preferred velocity vector. As we used the unit vector of the neighbour's velocity vector, we know the scalar projection is going to be between -1.0 and 1.0. The closer to 1.0, the more the neighbour's preferred velocity vector points in the same direction. The closer to -1.0, the more the neighbour's preferred velocity vector points in the opposite direction.

Hardly any neighbour agents are going to have exactly the same direction or exactly the opposite direction compared to our current agent. If we now decide on an arbitrary limit, we can decide on scalar projection value ranges in which we consider a vector to move in approximately the same or opposite direction.

In our implementation we used the values 0.8 and -0.8. Every vector whose scalar projection onto the current agent's vector is above 0.8 is considered to move in the same general direction while every vector with a scalar projection under -0.8 is considered to be moving in the opposite direction. Everything in between is considered to be moving in another direction (to the sides). An example is given in Figure 8, where every vector that has an end point above the green line is considered to be moving in the same direction as the first vector. Similarly, vectors which have an end point below the red

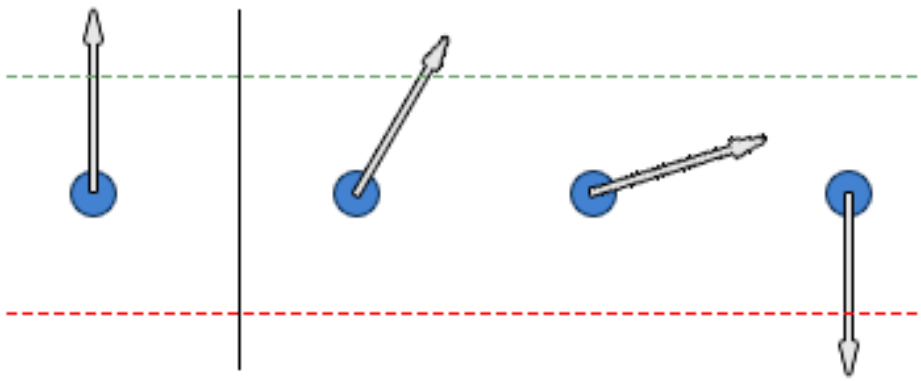


Figure 8: *An agent with his velocity vector, and 3 other agents with their own velocity vectors. Of the 3 compared agents, the first is considered to be moving in the same direction, while the second is moving in a different direction and the third is moving in the opposite direction*

line are considered to be moving in the opposite direction of the first vector.

For each neighbour, we calculate the unit vector from the current agent's position to the neighbour's position. Adhering to the rules describes above, we then add the unit vector to the current agent's preferred velocity vector if he's attracted to the neighbour and substract the vector from the preferred velocity vector if he's repulsed from the neighbour. Finally we take the unit vector of the newly calculated velocity vector and multiply it by v_{max}, the maximum agent velocity. We have now calculated the adjusted velocity vectors and can move the agents to their new positions. Our collision avoidance algorithm is described in Algo. 2 and the agent position update algorithm is described in Algo. 3.

3.3 Collision detection

After the global path planner and the local collision avoidance correction, we still need to check if any agents are colliding with other agents or obstacles. As has previously been indicated, the collision avoidance component does not guarantee this does not happen.

Collision detection can be implemented using a simple pairwise comparison. As pushing 2 agents away from each other also changes their position

Algorithm 2 Calculate adjusted velocities

```
for i := 1 to N do
  new_v[i] = 10.0*unit_vector(agent[i].v); {Add a significant factor so
  agent isnt thrown off course for 1 neighbour}
end for
for i := 1 to N do
  neighbours ← nearbyAgents(); {See §3.3 for selection method}
  for j := 1 to neighbours_size do
    if sameDirection(agent[neighbours[j]].v, agent[i].v) then
      new_v[i] += unit_vector(agent[neighbour[j]].pos - agent[i].pos);
    else if oppositeDirection(agent[neighbours[j]].v, agent[i].v) then
      new_v[i] -= 2.0*unit_vector(agent[neighbour[j]].pos - agent[i].pos);
    else
      new_v[i] -= unit_vector(agent[neighbour[j]].pos - agent[i].pos);
    end if
  end for
end for
for i := 1 to N do
  agent[i].v = unit_vector(new_v[i])*v_max;
end for
```

Algorithm 3 Update agent positions

```
for i := 1 to N do
  agent[i].pos += agent[i].v;
end for
```

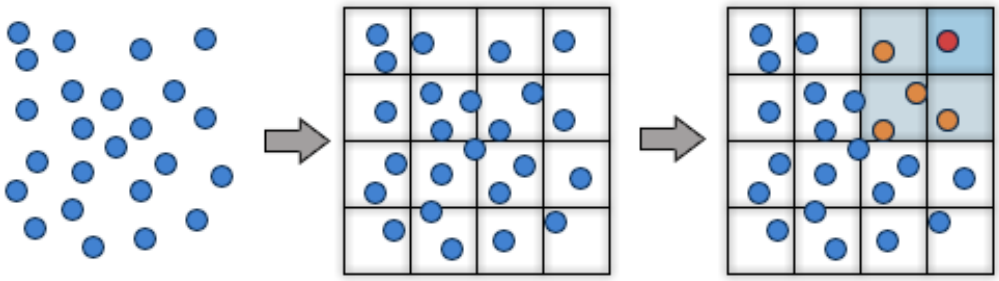
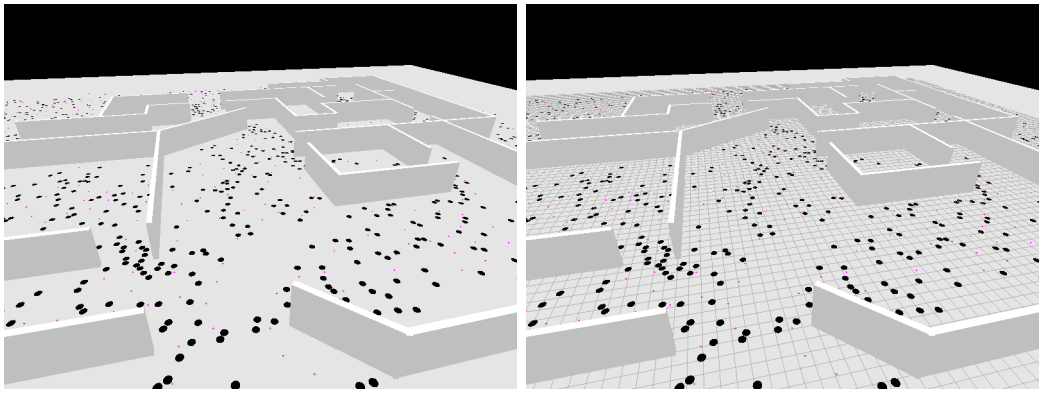


Figure 9: *An overview of our spatial hashing algorithm and how we can use it to select neighbours for comparison.*



(a) **(b)**

Figure 10: *A typical crowd simulation. In (a) the crowd is shown moving around. In (b) the collision grid is visualised on the ground*

relative to other agents, seperation of all pairs is not always guaranteed. However, if the local collision avoidance component can prevent overcrowding, this should still give good results, as every pair of agents is checked (and possibly corrected) at every timestep.

Comparing all agents in the environment to each other is unnecessary and too computationally expensive. Given N agents, we would have a complexity of $\mathcal{O}(N^2)$. For large amounts of agents, this would quickly bottleneck the performance of the entire simulation.

Agents and walls which arent close to each other cannot possibly collide and do not have to be compared. Using this observation, we can come up with a more efficient solution. By placing a virtual grid over the environment, the simulation can manage data on which agent is in which grid cell. For each grid cell, we can maintain a list of all agents whose center is currently in it. Now, finding all the agents or obstacles which can possibly collide with a specific agent is a matter of knowing the cell ID number which is associated with the agent's world coordinates.

We use the following formulas to determine what cell the agent is currently in, based on his position in the world:

$$cell_x = \left\lceil \frac{pos_x}{cellsize} \right\rceil, cell_y = \left\lceil \frac{pos_y}{cellsize} \right\rceil \tag{4}$$

$$cell_{ID} = cell_x + cell_y * (gridrowsize + 1) \tag{5}$$

Coordinates can now be translated to specific cell identification numbers, so it is possible to create a hash table that maps groups of agents to cells on the grid based on the cell IDs. We implement this using an $std :: unordered_multimap$, which is a data structure in the STD library of the C++11 specification. Using this data structure, we can map multiple values to the same key and retrieve lists based on specific keys or a range of keys.

We use 2 unordered multimaps in our implementation: one for the agents and one for the obstacles. The obstacles aren't added, moved or removed while the simulation is running, so we can use a simpler multimap for them which does not need to be constantly updated.

For the agent multimap, we use the $cell_{ID}$ as the key and the $agent_{ID}$ as the value for a mapping. For obstacles, we similarly use the $cell_{ID}$ as the key and the $wall_{ID}$ as the value.

During environment creation we now map the walls to their respective cells and the procedure is analogous for agents during initial agent placement. At the end of the collision detection step we now also recalculate all the agent cell mappings and update them if necessary, as described in Algo. 5. As the collision detection and resolution is the final step in every frame of our simulation, this ensures agents are always accurately mapped at the end of each cycle.

Now we want to look at a selection of agents and obstacles for collision detection for each agent in the environment. As we map agents to cells based on their coordinates, which are based on the physical center of the agent, looking up the list of agents and obstacles in the same cell is not enough. If a specific agent's center is close to a cell border, then there is a chance part of his physical space is on the other side of the border and can thus collide with another agent's physical space in the neighbouring cell, and vice versa. Because of this fringe case, we need to look at not just the agent's own cell, but also the up to 8 directly neighbouring cells. We illustrate our method in Fig. 9 and describe the procedure in Algo. 4.

Finally, we look at resolving situations where we detect agents are colliding with other agents and/or walls. As agents are represented by circles in our simulation, solving a collision between 2 agents is relatively simple: we

Algorithm 4 Detect pairwise collisions

```
for i := 1 to N do
    cellID ← calculateCell(agent[i].x, agent[i].y);
    neighbours ← neighboursInArea(cellID); {An array of agent indices}
    for j := 1 to neighbours_size do
        if circleCollision(agent[i],agent[neighbours[j]]) then
            move_apart(agent[i],agent[neighbours[j]]);
        end if
    end for
    walls ← wallsInArea(cellID) {An array of wall indices}
    for j := 1 to walls_size do
        if circleRectCollision(agent[i],wall[walls[j]]) then
            move_away(agent[i],wall[walls[j]]);
        end if
    end for
end for
```

Algorithm 5 Update agent mapping

```
for i := 1 to N do
    cellX ← calculateCellX(agent[i].x, agent[i].y)
    cellY ← calculateCellX(agent[i].x, agent[i].y)
    if agent[i].cellX ≠ cellX or agent[i].cellY ≠ cellY then
        agents[i].cellX ← cellX;
        agents[i].cellY ← cellY;
    end if
end for
```

can simply calculate the distance each one needs to be pushed away from the other as follows:

$$dist_{actual} = \sqrt{((agent1_x - agent2_x)^2 + (agent1_y - agent2_y)^2)} \tag{6}$$

$$dist_{push} = \frac{2 * agent_{radius} - dist_{actual}}{2} = agent_{radius} - \frac{dist_{actual}}{2} \tag{7}$$

If an agent is colliding with a wall, the closest point on the wall to the agent's center must be calculated, which can easily be done using vector projection techniques. Then we perform almost the same operation:

$$dist_{actual} = \sqrt{((agent1_x - rectPoint_x)^2 + (agent1_y - rectPoint_y)^2)} \tag{8}$$

$$dist_{push} = agent_{radius} - dist_{actual} \tag{9}$$

Effectively, the only difference is that only the agent gets pushed away.

After the collision detection component is done, the entire simulation step is done and we start at the global planner again.

In §6.1 we take an in-depth look at the performance impact the size of the collision grid cells has.

4 Behavioral model

In the previous section, we described 3 major components of our crowd simulation. The global path planner plans movement around obstacles while the local collision avoidance adjusts movement of agents that are close to each other. Finally, the collision detection component ensures no physical properties are violated by the previous 2 components. Essentially, we described how our agents move, but what we haven't yet described is why our agents move.

The behavioral model implements a set of goal assignment rules which aim to simulate some basic human behavior. The model describes when our agents get new goals and what these new goals are.

In this section, we will describe the features of our behavioral model. First of all, we take a look at the normal and evacuation modes we have

implemented in our model and how these affect the agent goals in our simulation.

Secondly, we describe our implementation of personal items and how our agents interact with them in evacuation mode.

Finally we talk about the addition of state of mind to our agents. We will describe our implementation of lost and panicked agents.

Evacuation and normal mode In the examples in the previous sections (Fig. 7a,7b,10a,10b) we generally assumed that agent goals were somehow assigned to simulate constant movement throughout a building scenario. This is our simulation's 'normal' mode and we will now precisely describe how this kind of behavior is created.

Our simulation starts in normal mode by assigning a random goal to each agent in the simulation. This new goal needs to satisfy 2 conditions: the goal must not be placed too close or inside an obstacle such as a wall. Some arbitrary value can be chosen as the minimum distance between goals and obstacles. Secondly, the new goal must not be further away from an agent than a certain value d_{goal}. This restriction is in place to simulate more of a browsing behavior, where agents look around the area and just decide on a new place to visit based on nearby points of interest. Generally, d_{goal} should be scaled based on the size of the scenario environment. If d_{goal} is too big, agents may move across the entire area for seemingly no reason.

The simulation continues in normal mode by assigning a new random goal to an agent whenever he comes within a certain distance $d_{success}$ of his current goal. If $d_{success}$ is too low, this will overly constrain agent movement. If there are many other agents in the immediate area, an agent might not be able to easily reach the exact point of his goal. If we require him to be very close or even on top of the goal before we assign him a new goal, there may be unnatural congestion as a result.

When simulating a browsing behavior, it is a logical condition to only require that a minimum distance to a goal has been reached. When a person wants to take a look at a specific piece of an environment, he does not necessarily have to do so from a specific position. On the other hand, if $d_{success}$ is too high, an agent will constantly get a new goal while he has not been anywhere near his previous goal.

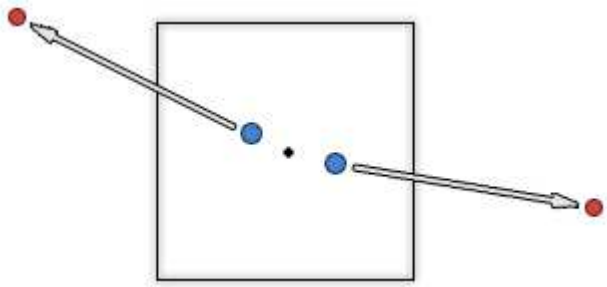


Figure 11: *An example of how escape goals are calculated for 2 agents. The blue circles are agents, the red circles are escape goals, the black dot is the center of the environment and the arrows are vectors with a length equal to the environment size*

The other mode we implement is an evacuation mode. Essentially, this mode has a start and end point. It can be triggered with the press of a key and when it is done, the simulation is over as every agent has left the simulation area. At the start of the evacuation, every agent gets a new goal calculated which has to be somewhere outside the simulation area so the agent moves out of the area.

We calculate this new point by adding some kind of reference point, such as the center of the environment or the center of the building in our scenario. We can then calculate the vector from this reference point to an agent and extend this vector by a certain amount, such as the size of the environment, to get a point which is guaranteed to be outside the environment area. This method is illustrated in Fig. 11.

This simple point generation method has proven to be effective enough at generating escape paths for our agents, as shown in Fig. 1b. The only downside is that the flow of agents exiting the building may needlessly walk to a different side of the building, but this could easily be fixed by recalculating the escape path when an agent reaches an exit.

When an agent reaches a point outside the simulation environment, we deactivate the agent so he stops being drawn on-screen and is not taken into account anymore for any calculations. Essentially, the agent never actually reaches the goal outside the environment and instead gets removed from the simulation as soon as he's no longer important to the simulation.

Personal items For additional realism, we implement objects which have a connection to specific agents. At the start of the simulation, each agent has a chance of being assigned an object. This object is placed in specially designated object spawn areas and these spawn areas can be thought of as being, for instance, cloakrooms where some agents hang their jackets. For simplicity, these objects do not have any collision detection.

Now, we modify the evacuation mode as follows. When the evacuation is started, agents get assigned an escape goal if they do not have an object somewhere in the simulation environment. If they do have an object somewhere, they move to retrieve it first by setting the object's coordinates as their goal. After retrieving their personal object, they get assigned an escape goal and carry on with the evacuation.

State of mind Finally, we add a state of mind to each agent. We implement 3 basic states: *normal*, *lost* and *panic*. The *normal* state contains all the behavior we have described up untill now and essentially means that the agent behaves as expected.

Agents in the *lost* state ignore the global planner and instead move slowly and randomly change direction. Agents have a small chance to enter the *lost* state at every timestep during the normal mode. When an agent goes back to the *normal* state, a path is again calculated to his goal.

As soon as evacuation starts, each agent has a chance to enter the *panic* state. When an agent enters the *panic* state, his movement speed is increased and he runs to his escape goal as quickly as possible. If the agent has a personal object somewhere in the simulation area, it is ignored and left behind.

The collision avoidance algorithm of Algo. 2 is modified so nearby agents are always repulsed from lost and panicking agents, regardless of the direction they are moving in.

5 Algorithm summary

To summarize the previous 2 sections, we describe the crowd simulation loop here:

- Agents get assigned new goals if they reached their previous goal in the last iteration. Goals are based on the agent's current state of mind and whether or not an evacuation is in progress(§4)

- The global path planner assigns a new preferred velocity $\vec{v_i}$ to each agent i, based on their current position, goal, state of mind and any obstacles in the environment.(§3.1, §4)
- The local collision avoidance component adjusts each agent's velocity $\vec{v_i}$ based on other nearby agents, in an attempt to avoid collision and create natural crowd motion.(§3.2)
- Each agent updates its' position using the adjusted velocity as $p_i = p_i + \vec{v_i}\Delta t$.
- The collision detection component detects if any agents are colliding on their new positions p_i and solves any collisions by moving agents away from each other and/or obstacles.(§3.3)
- For each agent, we calculate the current cell they belong to based on its' final position of this iteration p_i and for any agents which have moved to a different cell since the last iteration, we remove their old mapping and add a mapping to the current cell.(§3.3)

6 Results and discussion

In this section we take a look at the performance of the collision detection component. By changing the size of the collison grid we can get different performance results for different numbers of agents. We take a look at some of the results and try to determine an optimal cell size for the agent mapping.

After that we also evaluate our complete crowd simulation by simulating a building evacuation scenario and observing the behavioral features we've added to the agents in our model.

Finally, we discuss the limitations of our crowd simulation and collision detection performance results and provide some idea of how our work could be improved upon in the future.

6.1 Collision detection performance

In this section we take a closer look at the impact the agent mapping grid has on the performance of the collision detection component. Our performance tests were calculated on a machine with the following specification:

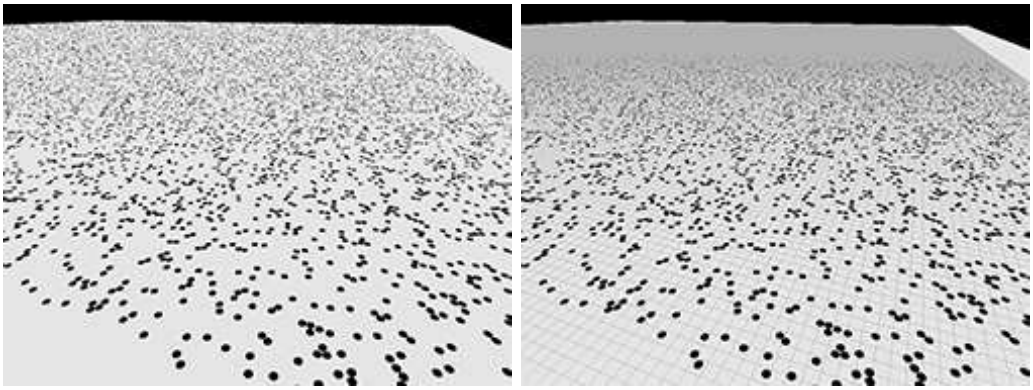


Figure 12: *The environment we have used for our collision detection performance experiments. On the left the environment is shown without the collision grid and on the right a collision grid with cell size 2 is displayed.*

- CPU: Intel Core 2 Quad CPU Q9650 @ 3.00GHz x 4
- RAM: 3.99 GB DDR2
- GPU GeForce GTX 560 Ti/PCIe/SSE2

For this experiment we wanted to focus on the basic agent-to-agent collision, so we used a large environment of 1000 by 1000 units with no obstacles. We placed a variable amount of agents with a radius of 0.6 in the environment and let them perform their normal browsing behavior. In Fig. 12 the test environment is shown with 100k agents.

Global path planning and collision avoidance were disabled for this experiment, as we wanted to focus on the performance of pure collision detection. The lack of global path planning was not noticeable as there are no obstacles in the testing environment. The lack of collision avoidance means that agent clumsily bump into each other to pass each other, but in general they still reach their goals provided the area isn't too compact and major congestions don't take place.

There are 2 important performance factors for our collision detection algorithm: the number of agents and the collision grid cell size. In Table 1 we compare the performance of various combinations of agent numbers and grid cell sizes for the above mentioned test environment. Every row describes

a different number of agents and every column describes a different grid cell size.

The *No hash* column describes a trivial cell size which is equal to the size of the world. In this case, we don't actually map the agents to a specific cell and every agent is compared to every other agent at every timestep. This is the most computationally expensive method ($\mathcal{O}(N^2)$) and is only used to compare more sophisticated cell sizes.

Columns called *Cell-x* describe performance results for a collision grid with cell size x. For instance, as the world size used for this experiment is 1000 by 1000 units, *Cell*-10 describes a collision grid with cell size 10 and a grid size of 100 by 100 cells.

Agents	No hash	Cell-100	Cell-50	Cell-25	Cell-10	Cell-5	Cell-2
100000	86005.20	48143.20	12021.40	2661.93	565.78	261.92	143.00
50000	21755.60	7907.50	1319.65	441.91	131.49	79.23	49.74
10000	875.71	224.51	64.46	27.46	14.35	9.84	7.81
5000	193.08	54.21	28.94	13.87	6.65	4.60	3.91
1000	8.59	3.65	1.94	1.56	0.87	0.74	0.74
500	1.90	1.28	0.80	0.58	0.46	0.41	0.39
100	0.17	0.18	0.16	0.13	0.13	0.14	0.14
50	0.13	0.13	0.13	0.13	0.13	0.13	0.13
10	0.13	0.13	0.13	0.13	0.13	0.13	0.13

Table 1: *Performance (in ms/frame) for combinations of agents and grid cell sizes*

We notice that the collision grid has no significant performance impact on agent numbers up to 100. We anticipated it would actually reduce the performance for low amounts of agents as the continuously updated agent mapping should add extra overhead. Apparently, the C++11 *std* :: *unordered_multimap* is efficient enough to not negatively impact performance at a low number of agents.

The computation time goes noticeably up with more agents. From 1k to 10k agents the time needed is multiplied by 10, but from 10k to 100k agents computation takes about 20 times longer. This makes sense, as we do not scale the environment size based on the number of agents. When more agents

are added to the world, there is an increase in both the number of agents and their density. A higher density means each agent has more neighbours and as such needs to do more collision checks. A more linear relation between the number of agents and computation time would be achieved if we did scale the environment size based on the number of agents.

Our test environment's cell size could not go much lower than 2 as it is bound by the radius of the agents. As we set the agent radius to 0.6, setting the cell size to something smaller than 1.2 would no longer guarantee we're looking at all agents which are candidates for collision by just checking the agents in the current cell and neighbouring cells. Our method would then stop being effective and the algorithm would have to be rewritten.

The cell size also has an upper bound which is equal to the size of the environment. At this trivial cell size the mapping solution is no longer useful and simply comparing every agent to every other agent would be more efficient.

For lower numbers of agents, the diminishing returns are very noticeable. Yet, for numbers higher than 5k agents, the decrease in computation time from Cell-5 to Cell-2 is still significant. Moving from Cell-5 to Cell-2 gives the best results for 100k agents, as would be expected. In general, using a smaller cell size does not seem to have any negative effect on performance, while the performance gains on higher number of agents are significant enough to make it worthwhile.

Our test environment was created to simulate up to 100k agents. If the environment was even bigger, it should be possible to simulate even more agents at a reasonable framerate. The environment should be scaled up in such a way that the density does not increase by a significant amount. The machine used for these experiments is by no means considered a high-end system today, so a more up-to-date machine should be able to run these experiments significantly faster.

6.2 Behavioral features

We tested our crowd simulation algorithm on a custom made building scenario. Using this scenario, we now take a closer look at the features of our behavioral model and evaluate them.

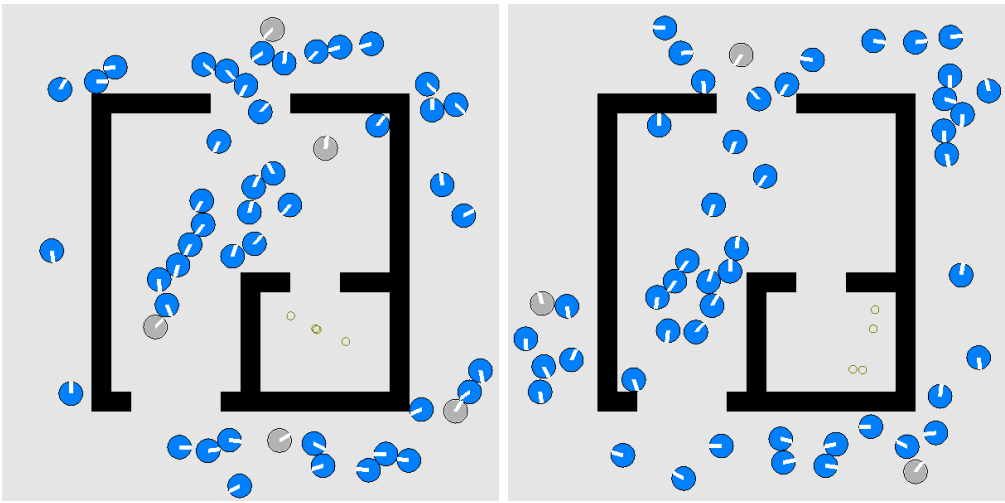


Figure 13: *A crowd simulation in normal mode.*

In Fig. 13 a screenshot of the simulation's normal mode is shown. The agents start with random goals and constantly get assigned new random nearby goals whenever they come close enough to their current goal. The result is that agents are exhibiting browsing behavior and moving in all directions. This often creates significant congestion at the building entrances as many agents try to enter and exit the building at the same time. The agents adjust their velocities and form lanes in an attempt to avoid blocking the agents going in the opposite direction. This is emergent crowd behavior often seen in real crowds.

In Fig. 14 an evacuation has been started. The agents have been assigned escape goals outside the simulation environment. As a result, they are all moving towards the building exits using the fastest possible route and we can see the building rapidly emptying. This is similar to real evacuations, where people generally follow the emergency exit signs to take the quickest route out of the building. Agents are also forming single lanes with their neighbours as they are all moving in the same direction. At a certain point, an agent leaves the simulation environment and is removed. Eventually, all the agents are removed from the environment and the simulation is at an end.

In Fig. 15 an evacuation is shown with people retrieving their belongings from the cloakroom before actually leaving. These agents were assigned goals at the locations of their personal items. Most of the other agents have

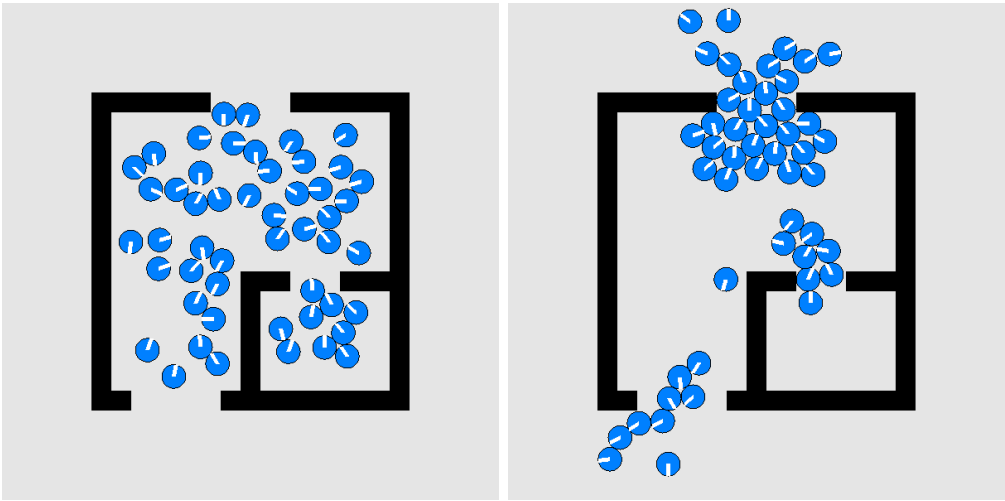


Figure 14: *A crowd simulation in evacuation mode.*

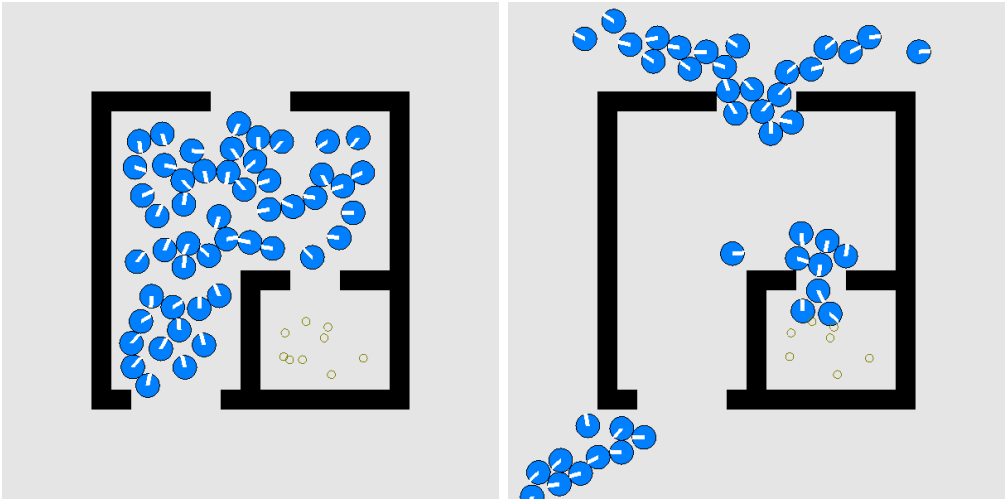


Figure 15: *Agents retrieving their objects from the cloakroom before leaving during an evacuation*

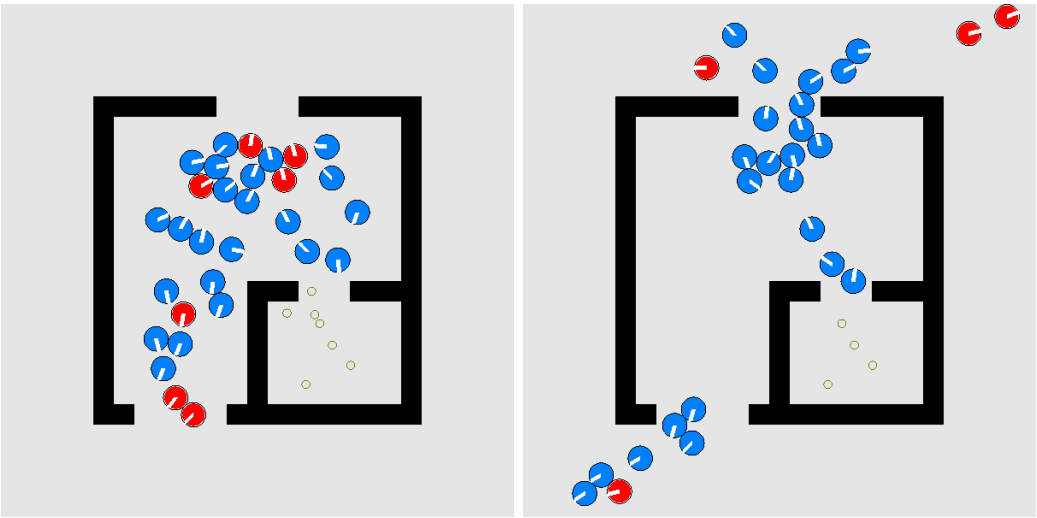


Figure 16: *Panicked agents (marked red) during an evacuation.*

already left and the building is for the most part empty, but some agents are still inside the building to get to the cloakroom for their belongings and are putting themselves in danger. After grabbing their personal objects, the agents do move for the nearest exit, which is close to the cloakroom. The most interesting effect is that people may cross the entire building if they start on the other side, greatly delaying their escape. It is not unheard of that people sometimes value some of their personal belongings enough to put themselves in harm's way to retrieve them before leaving a dangerous place.

In these tests we gave approximately 1 in 8 agents a personal object, but if we greatly increase this number, major congestion occurs in the room containing the objects.

In Fig. 16 an evacuation with panicked agents is shown. The panicked agents still have a goal they move to, but they move twice as fast towards it compared to normal agents. In the first image the evacuation has just started and the panicked agents are spread out over the group of agents in the building. In the second image the evacuation has been progressing for a couple of seconds and we can see the panicked agents have mostly made their way to the front of the group, as they move faster and can also move through groups of agents more quickly as other agents are repulsed by them.

6.3 Limitations and future work

Since our implementation of local collision avoidance is very basic, there are some limitations to what our current implementation of the crowd simulation can effectively handle.

At very dense situations, our collision avoidance component is often not able to effectively prevent deadlocks. Our building scenario tests were done with 2000 agents, which runs well most of the time, but if we try to place 4000 agents in the scenario, there is a high chance of a deadlock at one of the entrances at some point. As our pairwise collision detection component relies on the collision avoidance to prevent high densities, deadlocks can cause graphical artifacts.

In addition to this, our collision avoidance component creates relatively unnatural results. Lanes are formed if groups of agents want to move into opposite directions, but these lanes are often long lines of agents following each other. Human crowds generally look much more irregular.

Another major downside of our local collision avoidance component is that it's relatively slow. Depending on how many neighbours an agent has, a lot of calculations may have to be done. As such, it does not scale very well for even larger amounts of agents.

For future work, the collision avoidance component could be extended to create more natural results which can handle any amount of agents. However, the current method's performance is still heavily tied to the number of agents in the simulation. For a more scalable solution, the collision avoidance methods of potential fields [19] or fluid dynamics [13] could be implemented instead. These methods decouple the computational costs from the number of agents and can be used to perform more efficient, yet still realistic collision avoidance. As these methods significantly simplify the cognitive model of agents, the challenge would then be to keep a degree of diversity in the decision-making of each agent.

The new behavioral features we've added are an interesting step towards crowd simulations with even more heterogenous agents and we anticipate that it would be worthwhile to continue exploring this area. The current behavioral features could be further expanded upon with, for instance, more detailed and specific behavior for panicked or lost agents. Many new features

could also be added, such as agents searching for other agents before leaving during an evacuation or reduced vision due to smoke.

Conflicts between agents, such as trampling, could also be a realistic addition. However, most collision avoidance methods used today are very focused on perfect, non-colliding crowd movement. The challenge would thus be that adding in agent conflicts is not very straightforward or intuitive. It might be worthwhile to research the possibilities in the future.

Finally, we've tried out one collision detection method in this work. It might be worthwhile to instead try many different schemes for mapping agents to cells. For instance, we could map the agents to all the cells their physical space is partly in, instead of only cell in which the agent's center is located. This way, we'd only have to perform up to 4 cell look-ups instead of 9. It might even be worthwhile to try completly different mapping structures, such as [11].

7 Conclusion

Simulating large human crowds is a complex problem. Previous work has almost always focused on creating simulation methods with homogenous sets of agents which do not display many different kinds of behavior beyond an universal *moveToGoal* behavior. We have described a crowd simulator which specializes in simulating evacuation scenarios with some new special types of behavior, such as panicking and retrieving personal items before leaving the building. These new features add a more lifelike experience to the crowd simulation.

Additionally, we have evaluated the performance of a spatial hashing algorithm for large scale collision detection. The implementation of this algorithm uses the *std* :: *unordered_multimap* data structure of the C++11 specification and we noted that the overhead of accurately maintaining this hashing is negligible. As such, we have determined that using the smallest cell size possible results in practically no performance reduction for lower amounts of agents. On the other hand, the smallest cell size drastically improves the performance for higher amounts of agents by an compared to no spatial hashing.

References

[1] O. B. Bayazit, J.-M. Lien, and N. M. Amato. Better group behaviors in complex environments with global roadmaps. *Proc. 8th Intl. Conf. Artificial Life*, pages 362–370, 2002.

[2] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-collide: an interactive and exact collision detection system for large-scale environments. *I3D '95 Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff, 1995.

[3] M. Eitz and G. Lixu. Hierarchical spatial hashing for real-time collision detection. *IEEE International Conference on Shape Modeling and Applications*, pages 61–70, 2007.

[4] J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. *Proc. of ACM SIGGRAPH*, pages 29–38, 1999.

[5] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, and P. Dubey. Clearpath: highly parallel collision avoidance for multi-agent simulation. *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–187, 2009.

[6] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[7] L. Heigeas, A. Luciani, J. Thollot, and N. Castagne. A physically-based particle model of emergent crowd behaviors. *Proc. Graphikon '03 2*, 2003.

[8] A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. *Proc. of ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 19–28, 2004.

[9] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. *IEEE Int. Conf. Robotics and Automation*, pages 2138–2145, 1994.

[10] F. Lamarche and S. Donikian. Crowd of virtual humans: a new approach for real-time navigation in complex and structured environments. *Computer Graphics Forum*, 23(3):509–518, 2004.

[11] R. G. Luque, J. L. D. Comba, and C. M. D. S. Freitas. Broad-phase collision detection using semi-adjusting bsp-trees. *I3D '05 Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186, 2005.

[12] S. R. Musse and D. Thalmann. A model of human crowd behavior: group inter-relationship and collision detection analysis. *Proc. Eurographics Workshop*, pages 39–51, 1997.

[13] R. Narain, A. Golas, S. Curtis, and M. C. Lin. Aggregate dynamics for dense crowd simulation. *ACM Trans. Graph.*, 28(5):122:1–122:8, 2009.

[14] M. Overmars and P. Svestka. A probabilistic learning approach to motion planning. *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 19–37, 1994.

[15] C. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[16] W. Shao and D. Terzopoulos. Autonomous pedestrians. *Graph. Models 69*, pages 246–274, 2007.

[17] A. Sud, R. Gayle, E. Andersen, S. Guy, M. Lin, and D. Manocha. Real-time navigation of independent agents using adaptive roadmaps. *Proc. ACM Symp. Virtual Reality Software and Technology*, pages 99–106, 2007.

[18] M. Sung, M. Gleicher, and S. Chenney. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3):519–528, 2004.

[19] A. Treuille, S. Cooper, and Z. Popovic. Continuum crowds. *ACM Trans. Graph.*, 25(3):1160–1168, 2006.

[20] J. van den Berg, S. J. Guy, and D. Manocha M. Lin. Reciprocal n-body collision avoidance. *Proc. Intl. Symposium on Robotics Research*, 2009.

[21] J. van den Berg, M. C. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1928–1935, 2008.

[22] G. Vigueras, J. M. Orduna, M. Lozano, J. M. Cecilia, and J. M. Garcia. Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *The International Journal of High Performance Computing Applications*, 28(1):33–49, 2014.