Crowd Simulation for Emergency Response Planning

By Daniel Flower
Supervised by Dr Burkhard Wünsche and Assoc-Prof Hans Werner Guesgen

Table of contents

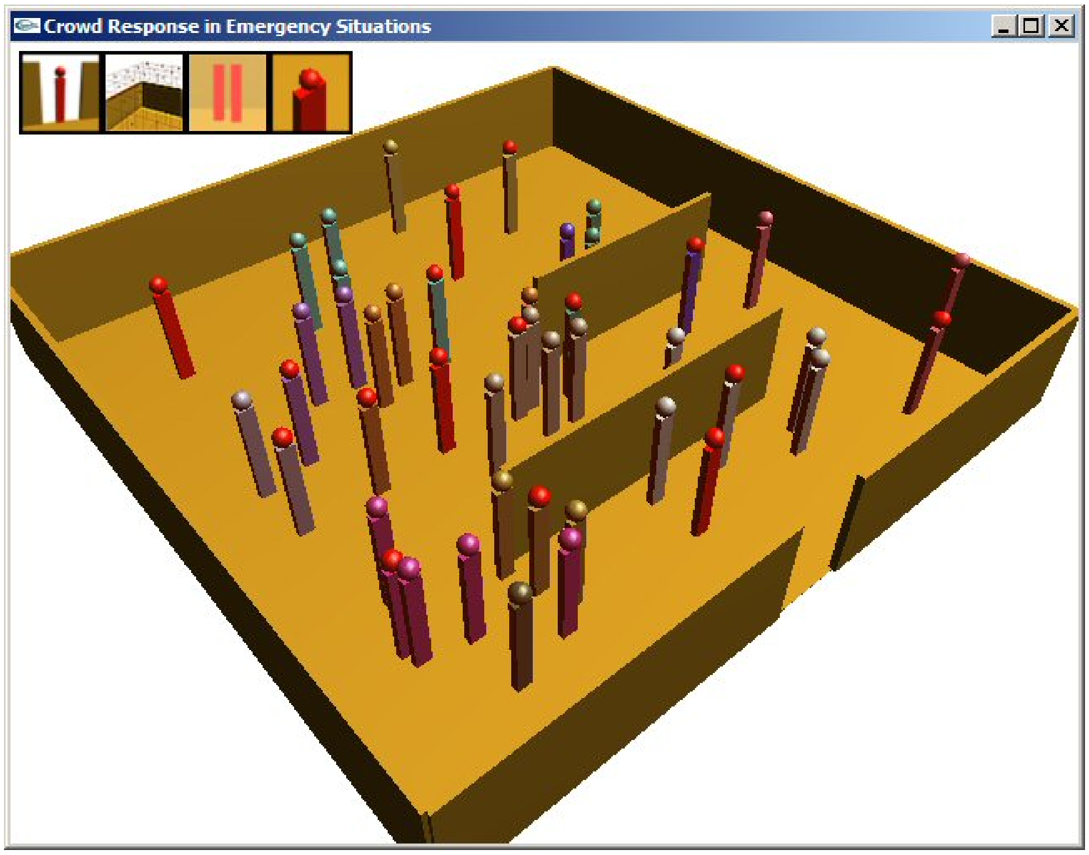
Executive Summary

Realistically simulating crowd behaviour is an important goal because the behaviour of crowds can often be unpredictable, and can lead to injury and loss of life when a panicking crowd is not managed in an acceptable manner. If designers were about to simulate a crowds, then they could better design buildings, stadiums, and events to minimise the dangers that can occur.

However, to simulate human beings is not an easy task. Therefore, the goal of this project was to make a simple simulation of a crowd trying to escape from a room with a single door. This was achieved by trying to simulate individuals, and giving them the ability to plan which route to take and how to avoid other agents. The "crowd behaviour" emerged from the individual agents' behaviours, much as it does in real crowds.

The result was a programme that reasonably simulated a crowd trying to escape a room. This included the ability for the individuals to decide which route to take to the door based on the length the route and the number of others taking the same route, and the way that bottlenecks such as doorways cause crowds to bunch together, and spread out.

This report will go over the motivation in more detail, existing crowd simulations, the design and implementation of this simulation, along with results and future improvements to be made.



Introduction

The goal of this project is to create a realistic crowd simulation. The people (herein referred to as "agents") in the crowd would need to plan where to go, avoid other agents, and act as human-like as possible.

Motivation

The motivation behind this project comes from the continuing disasters occurring in crowded situations. Commonly a real threat such as a fire breaks out which induces panic in the crowd, where people are in danger of getting crushed to death. It is this panic or poor management, rather than the original threat, which often claims the vast majority of injuries and deaths. For example, on December 31st, 2004, 169 people died when a fire started in a night club in Buenos Aires. In this case, the high number of deaths was blamed on fire exits being locked. On 25th January, 2005, between 150 and 300 people died during a religious procession in western India when a small fire caused a stampede[1].

These unnecessary dangers could be better prevented if the creators of buildings and events could test their designs under emergency situations. To hire a crowd for this purpose would not only be expensive, but dangerous. Therefore, a computer simulation of a realistic crowd, where the designer can test different designs (e.g. placing emergency exists, adding or removing walls etc), would enable the creation of crowd-safe events and buildings.

Aside from the safety aspect, a realistic crowd simulation could be useful in areas such as computer games, movies, virtual reality heritage simulations, urban planning etc.

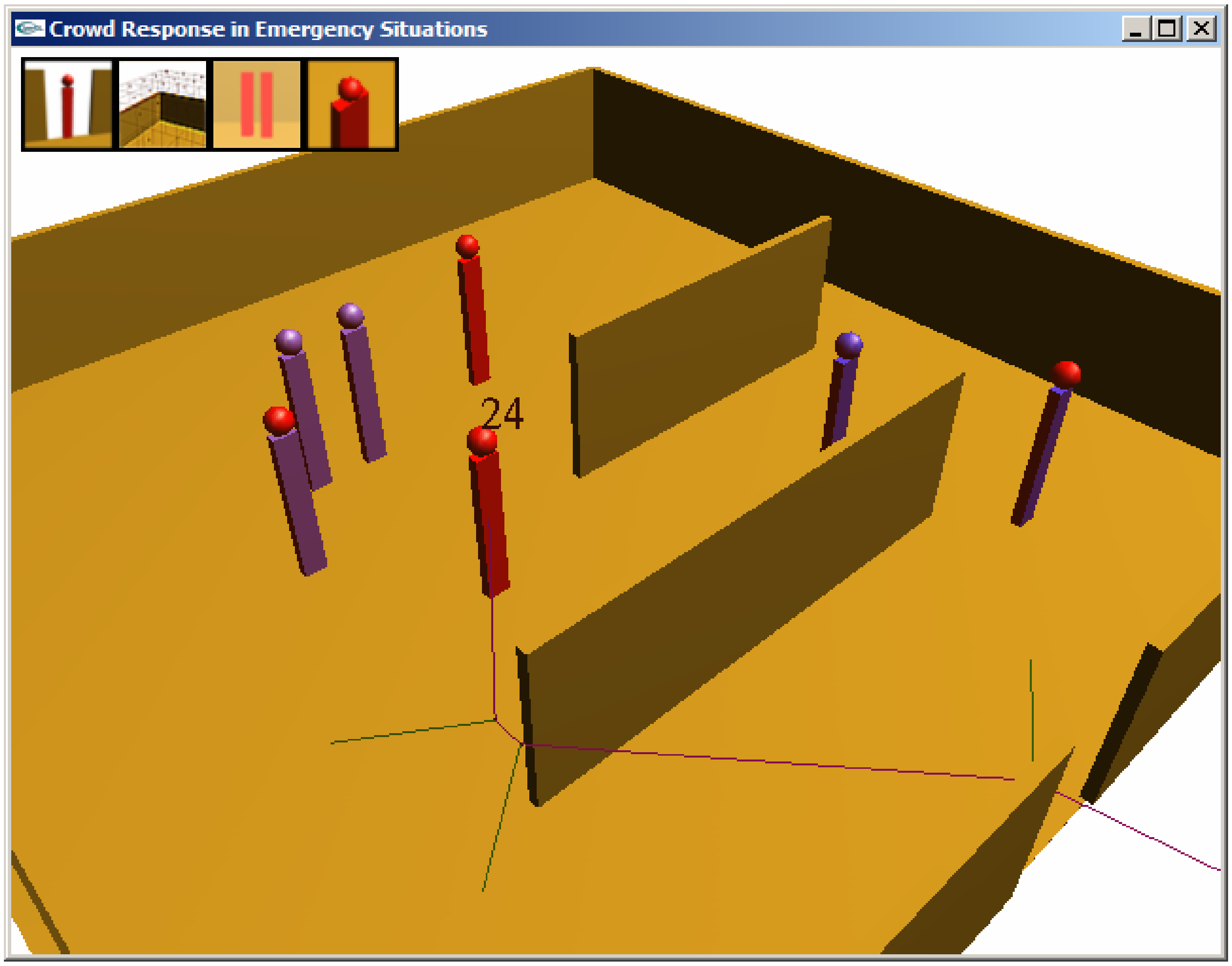
Goals of the Project

The goal of this particular project is to make a simple scenario and begin to investigate effective techniques for producing realistic movements of individuals within a crowd. Specifically, the program will be a single room with a couple of obstacles, populated with a crowd. The crowd will try to run to the door, requiring them to plan their path around the obstacles, and to avoid other agents.

Furthermore, the simulation should include some aspects not explored much in existing simulations, such as having groups of people (such as families and partners) where the group tries to stick together.

The agents in the simulation should have characteristics which can change over time. The most important characteristic in this simulation is panic level, which would affect the speed that people try to move at, how much space to give to other agents, and should affect their decision to either help a fallen person up, or whether to trample them.

Finally, it is very important that the project is extendable. The agents should be able to be ported to other simulations without too much trouble.



Literature Review

Empirical Data

Real life studies of crowds have shown that the following are common in crowds:

- As panic increases, the speed people try to move at increases. This increase in desired speed can actually decrease the overall time it takes for egress.
- Panic also affects things such as the amount of space people try to keep free around them, whether to act selfishly or in an altruistic way, and whether to act as an individual or to just follow the crowd.
- Any bottleneck – such as a door or a thinning walkway – results in slow-downs of the crowd. The crowd bunch around the bottleneck in a circular shape.

Any good crowd simulation ought to reproduce these empirical observations.

Particle Systems

In these simulations, individuals are treated as particles, and physical laws based on ideas of fluid motion are used. For example, a person will tend to flow in the same direction of nearby people. There are also repulsive forces between individual agents to make sure they don't come too close to each other.

An example of such a system is "Panic: A Quantitative Measure" by Helbing, Farkas and Vicsek[2]. Their system portrayed agents in a very simplistic way: physically they were circles; they used repulsion laws to avoid other agents; they either followed the crowd or looked for an exit themselves; and they tried to go faster when panicking. This lead to good results which agreed with empirical studies: the faster people tried to go, the slower on average they could leave; small openings became clogged; and that escape routes were better to stay uniformly thin than thin with some widenings.

Although particle systems can give realistic results, they treat the agents as physical objects rather than free-thinking individuals. People will often stop abruptly, head in opposite directions to the rest of the crowd, and act in ways cannot be simulated unless there is some sort of simulation of the way people think.

Matrix Systems

Path planning, avoidance, and decision making can become difficult when dealing with real numbers because there are an infinite[1] number of possibilities (for example, should you head 34.1112 or 34.1113 degrees east?). When trying to implement large crowds in

[1] Strictly speaking, it is not infinite on a computer because real numbers are only approximations. Nevertheless, it is still an extremely large number.

real-time, the computational cost can become too great, and hence matrix systems are used.

These systems greatly simplify decision making by splitting up the environment into cells, with one person per cell. When deciding whether an agent can move forward one unit, they just check if the adjacent cell is free or not, and path-planning is just a matter of finding a number of cells that are joined together and aren't occupied by static obstacles (such as walls).

Obviously, the real world is not partitioned into cells, and therefore matrix-based systems suffer from a lack of realism.

Behavioural Systems

In these systems, crowd behaviour emerges from the individual agents within the crowd. Each agent has attributes (such as whether or not they are in a hurry) and goals (such as "Exit the room") and they try to achieve their goals. Interactions (such as collisions) occur with other agents, which each of the agents must deal with individually.

In these systems, the main concentration is on creating individuals that can react realistically. Despite the fact that the crowd isn't directly controlled by the program, the behaviour of the crowd emerges from the individuals, much like it does in real life.

The implicitly created crowd behaviour can then have an influence on the individuals. For example, if most of the crowd is moving in a certain direction, then the individual may decide to move that way too. Importantly, they may decide *not* to move in that direction.

This type of crowd simulation is now the dominant system in use as it can produce the most realistic results. This is the system used in the simulation created for this project.

Design of the System

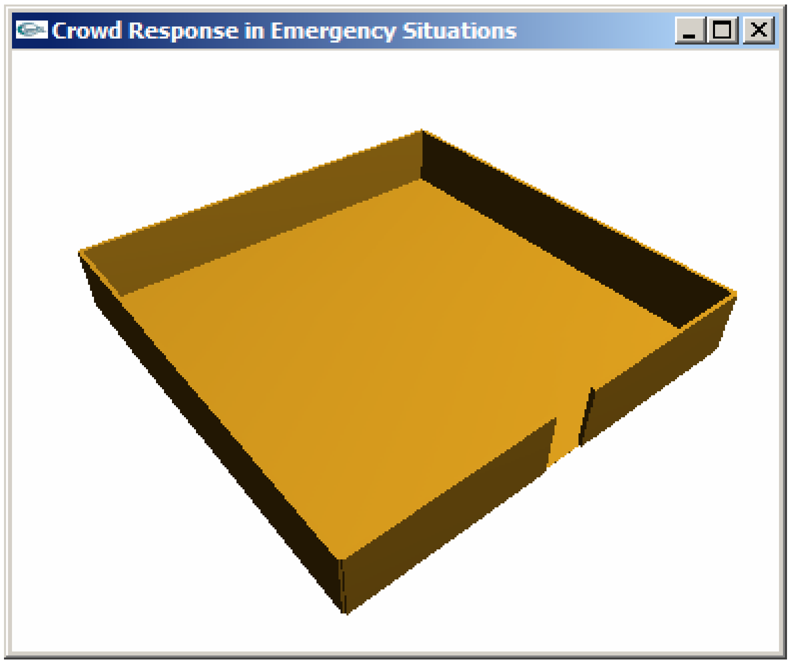
In this section, we will explore the design in fairly abstract terms. In the next section, "Implementation", we will go into detail about how specifically the design was implemented in the computer language.

The main goal of the design was to keep the simulation relatively simple while allowing room for later expansion. For this reason, it was decided that spatial calculations would be done in 2D. This greatly simplifies calculations to do with path planning and collision detection, however situations such as multi-level buildings are impossible to simulate due to the fact that if two agents were in the same location but on different floors, they would be seen by the program to collide. Graphically, it can still be displayed in 3 dimensions to make it look more realistic.

The system also had to be designed to be easy to use. The mouse was to be used to do things like adding people in real-time to the simulation and clicking on people would give statistics. This requirement to have real-time changes in the simulation therefore requires the simulation to be rendered in real time, rather than pre-computing everything and then displaying the result as an animation. Therefore, performance was an issue that had to be considered throughout the design of the simulation, and many simplifications were made for this reason.

Although this simulation takes place in a room, the goal was to make the agents and environment as generic as possible to allow changes to be made easily, and the possibility to port them into other simulation systems.

World Objects



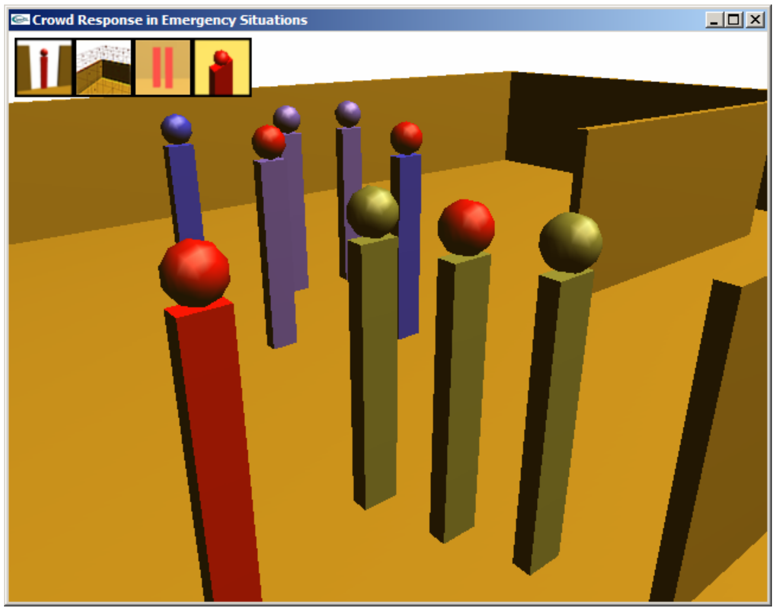
Every object in the world is referred to as a "World Object". The space every object takes up is represented as a 3D box. The drawing of an object and its physical representation are completely separate, so it is still possible to have things *appear* to be rounded or triangular etc. Another imposed limitation is that these bounding boxes cannot be rotated, so therefore you cannot have things like walls that are on angles. While this is a huge restriction on what can be created in the world, it simplifies everything tremendously, and is something that can be changed in the future.

There are currently two types of world objects: walls and people.

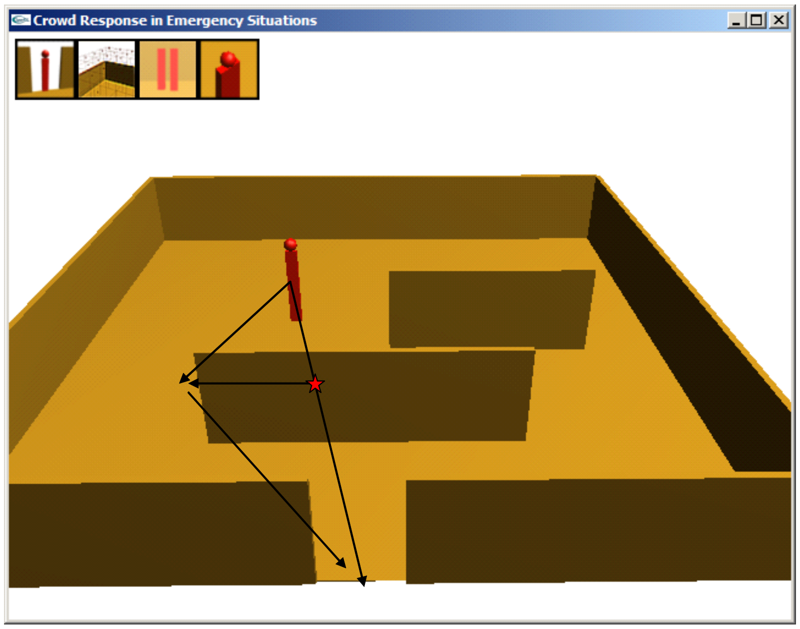
Walls

Walls are simply obstacles to block the agents. As mentioned above, they can only be placed horizontally or vertically. The floor is also a wall, which is stretched to be short and wide, rather than tall and thin, and is also (internally) marked as a floor. When adding a new agent to the simulation, they can only be added to where a floor is.

People



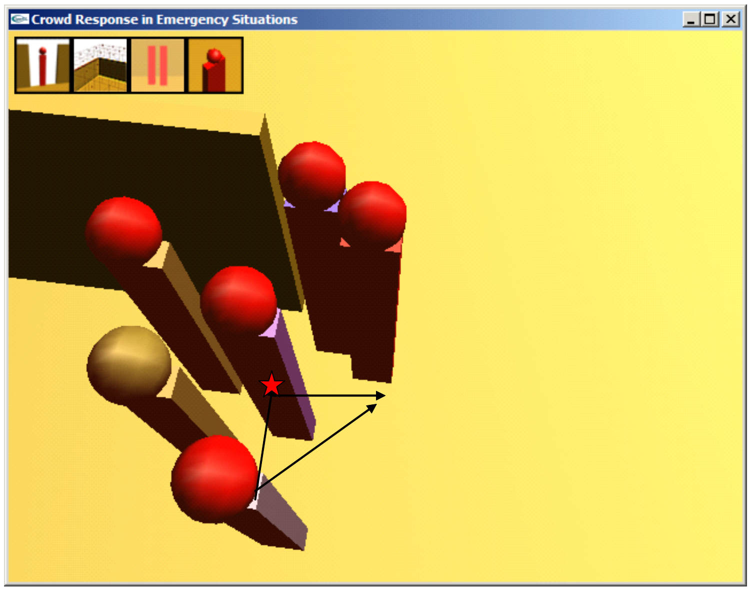
The people are obviously the main focus of this project. Being a “world object”, they share the properties of other world objects, such as being represented by a bounding box. However, they have extra properties to help them move around the world. Specifically, they need to have a destination in mind, along with current velocity. They also need decision making abilities, such as deciding which way to go, whether to speed up or slow down, etc. Each person will also be given a set of attributes that affect how they act in the world. An example is the ‘panic’ attribute, which causes the person to act with less caution and be in a bigger hurry.



When people are added to the world, currently their one goal is to try to leave the room through the door. To do that, they need to construct a path around any obstacles. To do this, they see if there is anything blocking them between their current location and the door. If there is, they check whether they can travel to the side of the object blocking them, and from there try to get to the door. This is repeated recursively until a path is built up. The created path is called an 'itinerary'. An itinerary is a list of locations in the world that the agent must pass through to reach their final destination.

When the agent has to decide between two paths, it takes into account the length of each path, and also the relative density of agents in the area. If there are many other agents in one path, then the agent is less likely to choose that direction.

In the path-finding step, although the relative density of the crowd is taken into consideration, the other individual agents are ignored. This is because people move around so if they are blocking the path now, by the time the agent gets there they have probably moved away. However, once they are moving they need a strategy whereby they can keep to their itinerary while avoiding other agents. The avoidance algorithm is very simple in this simulation. When an agent detects that it is about to walk into another agent (say, 1 meter before they collide), the agent will change direction slightly. From the agent's point of view, they can either go to the left or to the right of the other agent. Currently, the choice is made by determining the direction that requires a smaller change in angle.



Groups

A group is just a group of agents that want to stay together, and represent families, partners, and groups of friends that appear in real-life crowds. Including groups was an important part of this project because while they are very common in real life, they are not commonly implemented in crowd simulations.

Each group is made up of a group leader and a small number of followers. A leader follows their itinerary as explained above, however the followers in the group just try to move closer to the group leader. In fact, this is achieved by having the followers create an itinerary where the destination is the current position of the group leader, which enables followers to walk around walls and other people if they need to, rather than just blindly walking in the exact direction of where the leader is. Because recalculating the itinerary each time the group leader moves is computationally expensive, the itinerary is only updated whenever the group leader moves a significant distance (say, 1 meter).

In real life though, a group leader will not just ignore the followers and trust that they will keep up, but rather they will try to ensure that everyone stays close together.   For this reason, if a follower has got too far behind the group leader (say, 2 meters behind them) then they will tell the group leader this and the group leader will temporarily stop. Once all the followers are close to the group leader again, the leader will continue.

Implementation

This section goes into detail on how the simulation was implemented, exploring the programming techniques used. Some of it may be difficult to understand for those not familiar with algebra and programming.

The simulation was created in C++ using OpenGL for the graphics. As stated above, every object that is part of the world is derived from the abstract World Object class, which contains, among other things, a bounding box to represent the location of the object, a draw method, and a processing method.

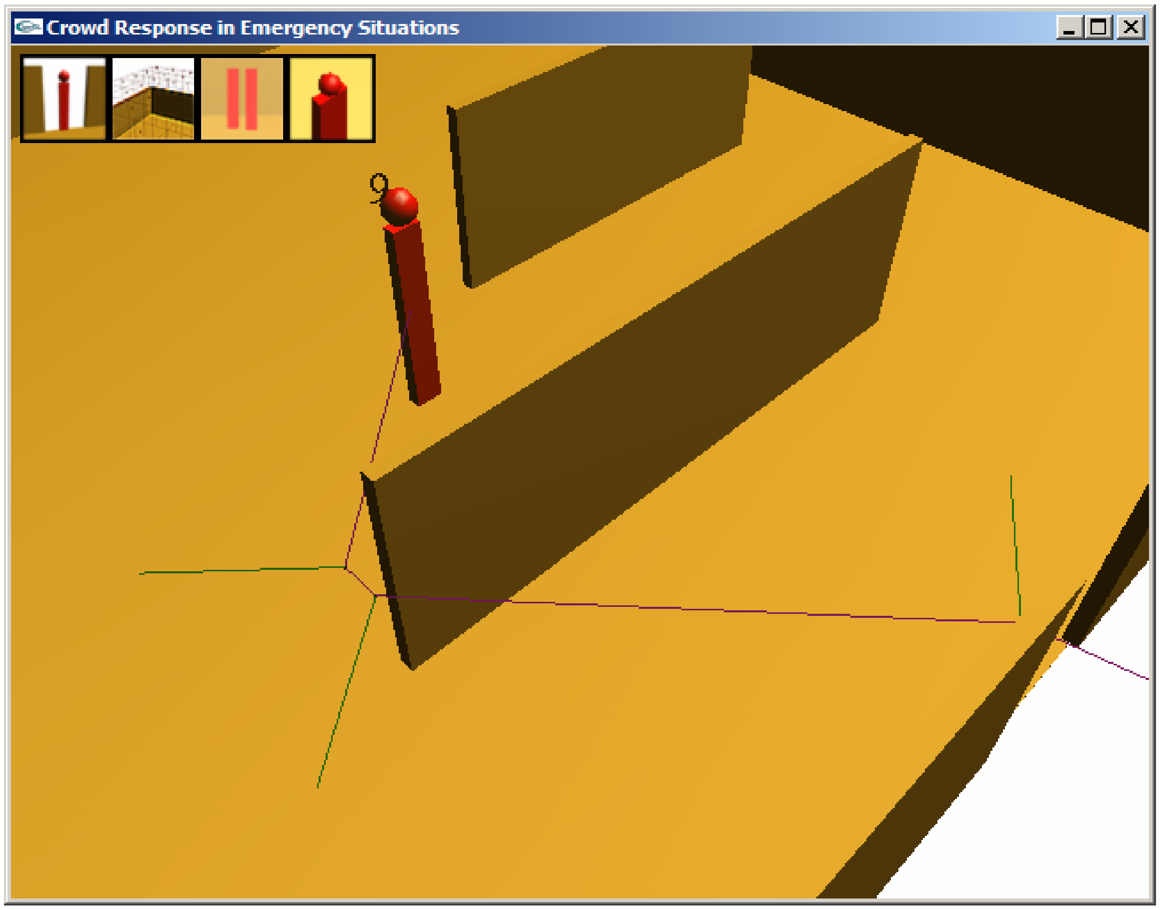
The main simulation class therefore has a linked list of every world object, and at each time step the processObject() method of every object is called to take care of things such as decision making and actually moving the object, and the draw() method to display each object.

Being inanimate objects, the walls do nothing in the processObject() method, and for the draw method they just draw a 3D box. The people, of course, aren't so simple to handle.

Path Planning

The first step after a person is added to the world is to plan the path they will take. This is not calculated every step, rather it is calculated at the beginning with the possibility to recalculate it later on if they need to. As mentioned in the previous section, the path is represented as an Itinerary object, which is just a linked list of locations that the agent needs to reach. The locations are always at the edges of walls because the path planner's job is to find a path around any obstacles (currently only walls) to get to the goal destination. However, the location is not just a point in space, it is a line segment. This is because in real life, when we walk from one point to another, we do not move to specific points, we are more approximate: for example it doesn't really matter if you travel around a corner tightly or take it widely; it's just important that you get around the corner. Therefore, each destination in an Itinerary object is represented as a point with a vector, and the line that this creates I have called a "finish line".

The image below shows the itinerary of a person trying to walk to the door. The purple lines show the most direct route, and the green lines are the finish lines that the person must cross before they try to reach the following finish line. I will next talk about how the point is decided upon, and how the vector is decided upon.



Creating Points in the Itinerary

To create the points we use ray tracing to check for collisions. A ray going from the person to the destination is created. The fact that every object is represented by a box means every part of every object can be represented as a plane. Each plane is represented by the equation ax + by + cz = d, and we can test the distance from a point to a plane by taking the dot product of the plane normal (which is just [a,b,c] transposed) and the point in question, then adding 'd'. If the value of this calculation is positive, then the point is in front of the plane (i.e. the normal points towards the point), and a negative value indicates that the point is behind the plane.

So, to see if we've potentially hit something, we classify the start point and the destination point against the planes of every object, and see if the classifications are different. If they are different, then there *might* be a collision; we still need to see if the collision point is within the bounds of the actual object, rather than the infinite plane of one side of the object.

To do this, we need the exact point of intersection, so we start by representing the ray as a parametric equation: ray = startPoint + t*vectorToDestination for $0 \leq t \leq 1$ and then find what the value of 't' is when the intersection occurs. The parametric equation gives the intersection point for the wanted value t, and rearranging the plane equation "n·p = d" gives the intersection point to be equal to 'd'/planeNormal[2]. We now have two equations for the point, and rearranging and simplifying finally gives us the equation we want to find the value of 't': t = (-startPoint·planeNormal + d) / (vectorToDestination·planeNormal). Plugging 't' into the parametric equation gives us the exact intersection point, and then we check if the intersection point is within the bounding box. To do this we check if each component of the intersection point is greater than each respective component of the back, left, bottom point of the bounding box of the object we may have collided with, and the make sure it is less than the front, right, top point of the bounding box (or to put it another way, we make sure that the intersection point is bigger than the smallest point of the box, and smaller than the biggest point of a box, where the size of a point is the sum of the components). Note that this works because bounding boxes cannot be rotated, and that this method would need to be improved to allow rotations of world objects.

If we have detected a collision, then from the agent's perspective, it needs to walk to the right or to the left of the object. The collision detection method returns the face of the box that we collided with, and using this face we can get a point which is to the left and to the right of the box. The method for finding these points are hard coded in (e.g. if (face == 1) return vertices[2] – where vertices is an array of the 8 vertices defining the box). This rather ugly method only works because we do not have rotations of world objects and because we are only concerned with 2 dimensions (depth and width – so we do not allow the ability for agents to climb over other objects).

So, we pick one of the points, and then recursively build two itineraries: one to get from the starting location to the intersection point, and another to get from the intersection point to the goal destination. This will build up as many locations as are necessary. But how do we choose which way to go around a wall?

One possibility would be to pick both ways, and then recursively check every single path and choose the one with the least total length. However, this could become computationally expensive, and is not the way that most humans choose their path. Instead, I used a greedy algorithm: pick the path that that looks to be the best. A score is given to each possible path, with points being awarded on the basis of the total length to get to the destination, and the number of other people who are currently near the point we are walking to. The lowest score is the winner, and specifically the score is generated by the following formula:

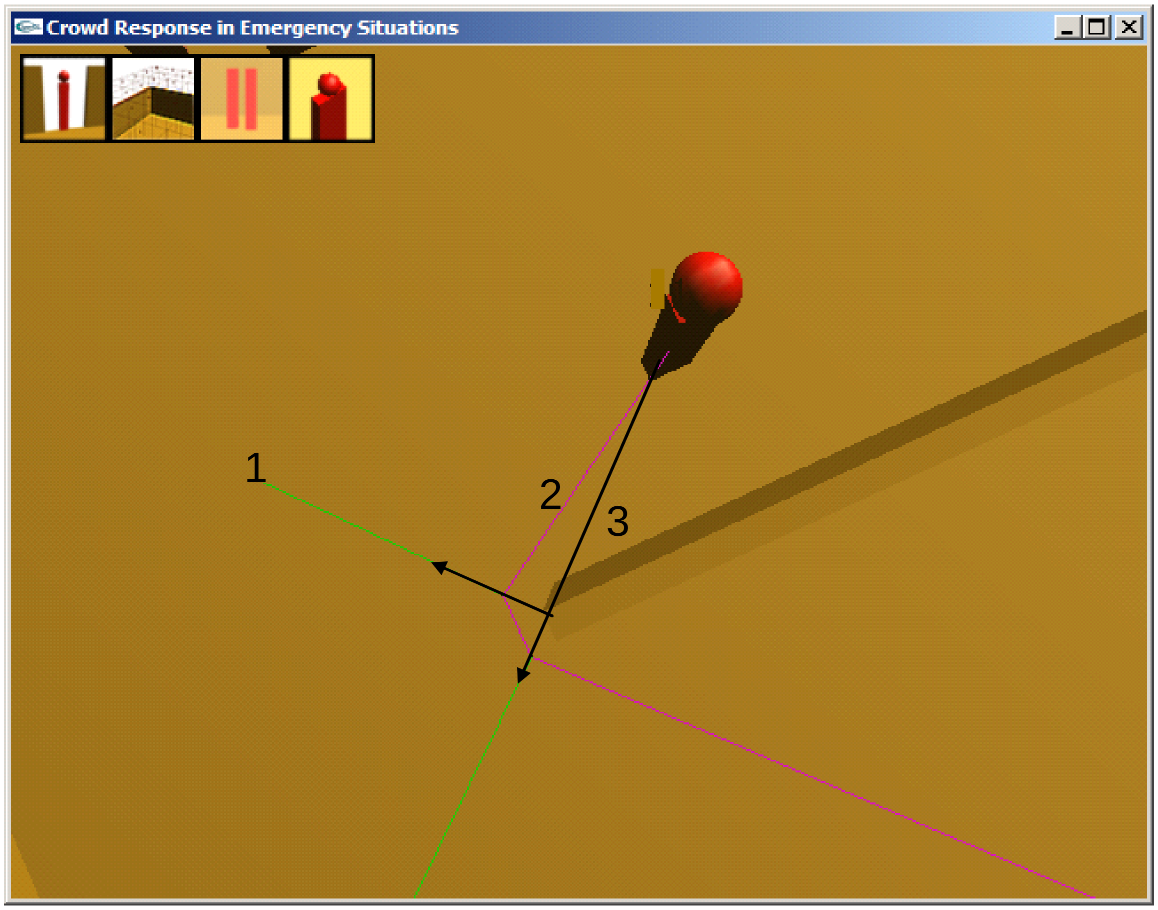
Score = (1 * totalDistance) * 1.2^(number of people near point)

The numbers in this formula are have no mathematical basis; they are just numbers that seem to work well (I've included the '1' to show that the total distance could be weighted, even though in practice it's not).

Creating the "finish lines"

[2] Really, it is 'd' divided by each component of the normal.

The finish lines need to point "away" from the object that we are walking around, however they do depend on where the agent is coming from and where they are going. This is best visualised with a diagram:

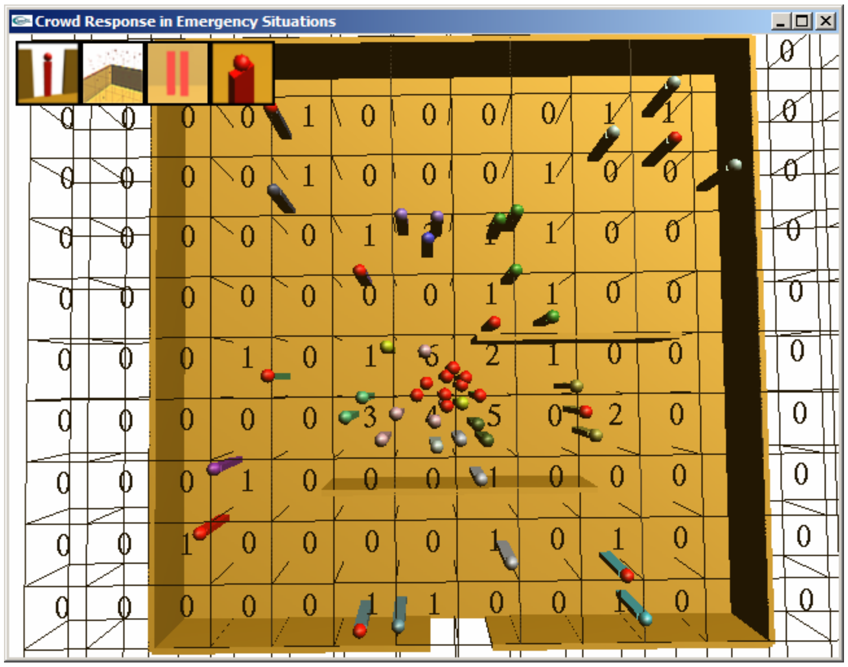


We are looking at how the vector marked as '1' is calculated (ignore the other green line). If vector 2 is projected onto vector 3, we can simply subtract this new vector from vector two to get a vector pointing in the direction of the vector marked as one. We can then normalise this vector, and multiply it by whatever length we want (currently a length of 1 meter is used). To calculate the vector marked as '3', we simply get subtract the point where the person currently is from the destination point after the next destination. This cannot be used to calculate a finish line for the very last destination in the itinerary as obviously there is no destination after the last destination. In this case we just use the previous finish line, which while not being an optimal solution works pretty well in practice.

The Grid

I have rather casually mentioned that the number of people near a point is taken into consideration; however coming up with this value is not a trivial task. To achieve this, the world is split up into cells which make up the grid. Each cell has a linked list of all the world objects that are in it (specifically, a world object is considered inside a cell if the centre of that object is inside the cell, even though the object may span several cells). Whenever a person moves, they check to see if they have changed cells, and the cell lists get updated accordingly.

In the design stage, it was mentioned that the followers of a leader have itineraries with the final destination being the group leader's position, and to save computational time the itineraries are updated not every time the group leader moves, but only when the group leader moves a significant amount. Specifically, every time a group leader leaves one cell and enters another, all the followers are told to re-compute their itineraries using the new position of the group leader. This is because it's not important that followers are in the exact same place as the leader – this is indeed impossible – rather the followers just need to be near their group leader. Therefore, having the followers trying to get to the current cell that the group leader is in is close enough.



The grid is created in the beginning by finding the smallest and largest points in the world, and then creating cells which span every object.

While currently the grid is used in a rather small way, the grid has a lot of potential uses, including optimisations (to only do collision detections with objects in nearby cells, rather than every object in the world) and for use in higher level AI decision making (for example, if the agent wants to follow other agents, then check the velocities of those agents in nearby cells and go in approximately the same direction and speed as the others).

Moving Around

Each person has information on the velocity they would *like* to be travelling at, the velocity they *are* travelling it, and the direction in which they are travelling. After their itinerary is calculated, their direction points to the middle of the next finish line and as they cross each line, the direction gets updated to point to the next finish line in the itinerary. When there are no other agents around the egress goes very smoothly, but when there are too many other agents around the agents cannot just stick to their simple itinerary.

The first requirement is to check to see if the agent is about to hit any other object. To do this, we stick out a “feeler” and run the collision detection algorithm – as used to create the itineraries – to see if it is possible to move, say, another meter in the current direction. This is similar to the way an insect uses its antennae to navigate its way around objects.

Avoiding People

If the agent detects that it is about to collide with another agent, as described in the design stage, it simply creates a vector that goes from the centre of the agent about to be hit (the “hit object”) to the current location of itself. This vector will be pointing away from the hit object, and by making this vector very small (e.g. making it 5 cm) and adding it to our current direction, the agent will turn a small amount away from the hit object, and over a few steps will avoid it completely. To match the observation that people slow down when they come close to others, the current velocity of the agent doing the avoiding is decreased slightly, and this slow-down can contribute to the whole slowdown of a crowd if it occurs in a congested area.

Avoiding Walls

If an agent strictly follows its itinerary, they will never hit any walls, but because agents change their direction to avoid other agents, they will sometimes collide with walls. When this happens, their itinerary is simply recomputed, and the new itinerary will make them head away from the wall.

Avoiding Nothing

If there will be no collisions, then, feeling confident, the agent can increase its speed a little (until it reaches its desired velocity). The agent then checks to see if it is going in the right direction (i.e. if it is heading towards any point on the finish line). It does this by creating a vector from the current agent position to the middle of the finish line and checking for the angle between that vector and the current direction vector. If this is smaller than the angle between the edge of the finish line and the first vector calculated, then the agent is heading for the finish line; if not, then the current direction is rotated slightly towards an acceptable level.

Results

Realism

The main goal was to realistically simulate the movement of a crowd by implementing individuals. To this end, there was some success. The agents can smoothly leave the room when there aren't many other people, and when there is a big crowd, bunches of people form around bottlenecks such as doorways. At a doorway, the crowd takes a semi-circular shape, with a few agents trying to move to better locations, and some staying patiently still, waiting to get out. This mirrors a real life crowd quite well, despite the fact that this was not explicitly coded in (all that the agents are told to do is try to move towards the door and step around other agents). However, there still remains an unrealistic element to this situation.

The idea of crossing a finish line rather than trying to cross an exact point was made so that the agents – just like people – could walk around not with pin-point accuracy, but in a more approximate method. That is, the goal was to move away from the exact world of computers into the more 'fuzzy' world of humans. While the finish line method was a good start towards doing this, it perhaps doesn't go far enough. When bunched around the door, there are almost always a few agents that could walk straight out the door, but instead try to walk backwards to cross their finish line. This creates unnecessary and unrealistic slowdowns of the crowd. Perhaps replacing the finish line with a 2 dimensional 'finishing area' could help in this regard.

The implantation of groups of agents trying to stay together worked well. When a group leader stops when someone gets too far behind, they can slow down the other agents around them. If you've ever walked down a busy street in a hurry you will appreciate that groups of people walking together normally travel the slowest, so in this regard the addition of groups of people should be considered an important part of any crowd simulation.

The graphics are currently very simplistic. While the concentration of this project was not on the graphics, a much greater feeling of realism would be achieved if the people and the environment looked more realistic, and if the agents were animated when they walked.

The biggest failure of this project was to implement a good way to allow agents to give themselves "personal space", with the size of this space depending on density of the crowd and the agent's own panic level. While the agents were able to maintain a level of personal space, this spaced was never compromised by another agent. This meant that people tried to maintain large amounts of spaces around the doors and other obstacles, which lead to deadlock situations where no one moved. I believe that to achieve a realistic implementation of this, agents will have to cross over into other agents' personal spaces, and each agent will need to be less strict in enforcing their personal space. The current situation in the simulation is that there is no idea of personal space, however in general when agents are moving they do stay away from each other due to the fact that they try to avoid each other, and it is only when the crowd slows down that the agents bunch up together.

User Interface

A major goal was to make the program easy to use. The ability to place new agents in the world using the mouse goes a long way to achieve this goal, although moving around the world using the keyboard is not as easy as it should be. Although not yet implemented, the ability to place walls with mouse-clicks would not be too difficult to implement, and would allow much greater experimentation by the user.

Efficiency

The simulation can run with a reasonable frame rate when there are fewer than 200 agents in the simulation[3]. While there are simulations that boast the ability to simulate 10,000 agents in real-time, they normally work on a more approximate "macro" level (for example, by using a matrix based system), rather than the more detailed level that this simulation works on. However, I still feel that on modern computers, 200 is too few, and future improvements in graphics and intelligence will only slow the simulation down further. Therefore, optimisations and perhaps more clever use of graphical techniques will be needed. An obvious example is to either use the grid or to create an oc-tree to cut out unnecessary collision detection with other agents.

Extendibility

The program was always written with expectation that features would be later added and improved, and with the possibility that, for example, the agents would be ported to work within another framework. Using object-oriented programming techniques achieves these goals, with the further benefits of keeping code fragments small (and hence easy to debug and tweak) and making the code easier to read for humans.

An example is the collision detection. Currently in 2D handling only rectangular shapes, it could be extended into 3D using any type of shape by just changing the collision detection methods. Indeed, it would be a good idea to remove the collision detection from this simulation and use an existing collision detection library to allow for all sorts of shapes in the world. The current collision detection methods would then just need to be changed so that they called the library's methods.

[3] The computer used was a 3.2GHz Pentium 4 with an NVIDIA GeForce FX 5700 video card and 1 gigabyte of RAM.

The encapsulation of code which object oriented programming brings also means that the code controlling the agents for example is completely separate from other code, and therefore to port it to another framework would be possible with minimal changes.

Overall

Keeping in mind that this was a relatively short project, the lack of good graphics and some unrealistic behaviour is not a negative, rather it is something that can be easily improved by future work. Overall then, I feel that the main goals of the project were achieved, and it was a successful first step to creating a realistic crowd simulation.

Future Work

As has already been explained in this report, there is much room for improvement. This section will highlight the most important areas that need improvement, and leave out ideas that have already been discussed in this report, such as improved graphics, better representation of personal attributes, and personal space, etc.

Force

Much like a wave, a crowd can become a powerful force when the individual, not necessarily strong agents all walk in one direction. To be able to simulate the force created by a crowd is extremely important in accurately simulating injuries and death. In a more advanced simulation, crowds may also be able to push down barriers and other obstacles.

The idea is that when an agent is panicking, and they are unable to move forward because another agent is blocking them, they would "push" the said agent. The force would be calculated using Newton's laws, with values coming from the agent's own attributes (for example, they could have a "strength" attribute). Further to their own calculation, they would also add the force that they are being pushed with. In a crowd where everyone was close together, this force would propagate from the back to the front, grower with strength. The force on each agent could decrease the health if the force was too great, and hence the idea of crushing could be simulated.

Without any idea of force in this simulation, the current situation is possible (and indeed, often occurs): A group leader almost reaches the door, however some followers in the group are too far behind, so the leader stops. The leader is then blocking the door, and therefore a large crowd is stuck behind them. This idea of the leader stopping at the door is realistic, however with the idea of force included, the leader would try to wait for his followers, but be pushed out by the force of the crowd anyway.

Artificial Intelligence Improvements

The artificial intelligence of the simulation is currently very basic, and there is still much work to improve even the low level areas of intelligence, such as moving around in the world, and trying to avoid and predict what other agents are doing. However, this section will give a brief overview on just 2 areas: planning and knowledge.

Planning

Currently, the agents have only one goal in mind, which is to leave the room. It would be desirable to be able to have different goals, perhaps implemented with something like a priority queue, which the agents could keep in mind simultaneously. For example, if while walking towards the door the agent encountered another agent in need of help, they could create a plan to help them, and when complete they could continue their original plan.

Knowledge

It is assumed in this simulation that the agents know where the door is. However, as was discussed earlier, much of the death and injury in buildings arises from the fact that the people are unaware of emergency exits. Therefore, each agent should have its own knowledge of the world, which could be added to by exploration. When wanting to leave a room, they would search their knowledge for all the doors that they know about in addition to trying to look for other doors and taking into account the movement of others in the crowd.

An interesting extension to this is the ability for agents to share knowledge. A possible way to implement this would be for an agent to make its intentions public (for example, "I want to find a door") and any agents near to that one could make available the list of locations of doors that they know about.

Senses

The agents currently have direct access to information on every object in the world, which they use to decide where and how to move around. This is unrealistic: for example people cannot plan their way through a building if they do not know it well; instead they take into account that which they are able to see, move around, and then revise their plan when they find more information. Therefore, to make it more realistic, the agent should learn about the world by rendering the world from their perspective, and then analysing the objects that the agent can "see" from their position.

The aural sensory input is also extremely important to humans. Therefore, certain objects must be able to make noise, and agents must be given the ability to "hear" these noises, and then react to the situation accordingly. This could be implemented using and event-driven system. When an object makes a noise (eg. when a bomb goes off), the type of noise and volume could be broadcast to agents (the volume, of course, depending on the distance from the source to the agent). The "type" of noise could be something like "danger noise", which the agent would want to perhaps have a look at, or simply run away.

References

[1] http://www.crowddynamics.com/Main/Crowddisasters.html
[2] Helbing, D., Farkas, I., and Vicsek, T. (2000). Panic: a Quantitative Analysis. Available at http://angel.elte.hu/~panic/ (Sept. 2003).