Room Capacity Analysis

Using a Pair of Evacuation Models

Table of Contents

Introduction

We present two models for determining the amount of time it takes a given number of people to evacuate a given room. A room's maximum capacity can be derived from this by imposing a maximum evacuation time. The maximum evacuation time must take into account factors such as the fire-resistance of the room and should be calculated, for example, by the Fire Marshall.

We developed a graph-based network flow simulation. People are modeled as a compressible fluid which flows toward and out the exit. This model assumes people's interaction properties, based on industry research.

We also developed a discrete particle simulation. In this model, people are modeled as disks that attempt to reach the exits. In this model, people's interaction properties emerge from local, per-person assumptions.

In this paper, we develop and analyze both models. We then compare and evaluate the models' outputs, and finally analyze the capacity of a local dining hall, gymnasium, lecture hall, and swimming pool.

Statement of the Problem

We wish to determine the maximum safe capacity of a room. This number is derived from a variety of factors, but a significant influence is the amount of time it would take for everyone to evacuate in the event of an emergency such as a fire. We wish to develop a model that, given a room and starting population, computes the total evacuation time.

Goals

Our goal is to create a single-space evacuation model using current fire safety and human behavior research and to evaluate the model under various conditions. Behaviors we wish to simulate include:

- Accurately model rate of egress for exits as measured by industry research.
- Accurately model the routing of pedestrians toward exits. Pedestrians should tend to avoid distant and congested exits for near, less-congested exits according to the relative distance and congestion of those exits.
- Accurately model the effect of obstacles (furniture, railings, walls) that may impede pedestrian flow.

There are many factors that influence the maximum safe capacity of a given room. We will ignore:

- The specific disaster. This means we will ignore smoke and fire modeling, earthquake damage, and poison gas.
- Structural load-bearing and resonance considerations. This is, for example, the primary limitation on the capacity of an elevator.
- Air exchange. OSHA requires 20 ft^3/min of air exchange per person in public workplaces.
- Public health. For example, pools should maintain an acceptable ratio of lifeguards to swimmers and high water filtration rate.
- Fire alarms and emergency lighting.

Background—Current Occupancy Limits

Occupant capacity and fire safety issues are inextricably related. Local fire departments and marshals are the designated enforcers of capacity limits. Capacity limit laws are generally included with other fire safety codes.

The maximum group occupancy of a room in a building is specified in the Uniform Building Code (UBC). The UBC requires that the space be classified according to its potential uses, structure and design. The code then dictates a square footage to occupant ratio that is used to determine the room's group occupancy limit. The UBC uses the group occupancy limit to specify the number and capacity of the exits. The UBC also places constraints on the relative positions of the exits in a room. The number obtained by this process is not a final figure. The Uniform Building Code also specifies the maximum occupancy of a floor of a building based on the exit capacity of the building as a whole. The capacity of the hallway leading from the room to the exit route is also a factor in determining maximum group occupancy.

The Uniform Fire Code (UFC) places additional constraints on the occupancy of a room based on its construction and fire safety measures. Other fire safety concerns are listed in the UFC that can affect maximum group occupancy. The Americans with Disabilities Act introduced new limitations to maximum group occupancy and modified the specifications for room exits.

The UBC and UFC change yearly. Each year, the federal government endorses a current version of the documents, with revisions to reflect advancements in the field of building safety. The UNB and UFC are not uniform from state to state. Different states can choose to implement sections of the codes. Also, localities are free to impose additional restrictions on maximum group occupancy. There are professionals that specialize in the interpretation and application of these rules and regulations. Much of the precedent is set by the International Conference of Building Officials, a professional organization dedicated to this process.

Background—Current State of the Art

Some commercial models currently in use focus on modeling a large number of spaces in a building configuration. The primary use of these models is to test the evacuation capacity of a building configuration as a whole. Two commercial models that use this approach are EXODUS, a particle-based system, and EVAC4NET, which models the building as a network.

This whole-building type of modeling is beyond the scope of this paper, but leaves an important area open to further exploration. In a situation involving a single large space that is the primary consumer of building evacuation route capacity, the evacuation capacity of the space may be the critical factor in evacuation time. In this case, a model which focuses exclusively on evacuation from that space may fill an important gap in fire safety modeling.

We now present the development of each model, starting with the graph-flow model.

Graph Flow Model

Overview

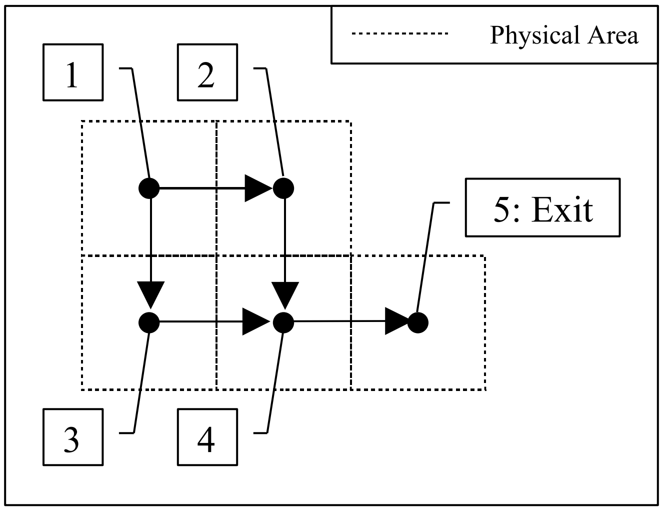
The graph flow model is a pool-flux model that operates on a graph representing areas of open space within a room. The graph consists of a set of nodes N and a set of directed edges E. Each node represents an area and a population. Each edge represents the direction of traffic flow from one node to another node. Due to the interdependent flow equations, the graph must be acyclic.

The ability of occupants to exit a node is constrained by the congestion in the node and the *bandwidth* of the edge leading to another node. Bandwidth is a concept borrowed from network theory and passenger terminal flow models. It represents the rate that people can move between nodes. There are two values for bandwidth used in our model: a high bandwidth indicates that there are no obstacles or doors between nodes, and a lower bandwidth indicates slower flow due to constriction at an exit.

The number of occupants that enter a node is constrained by the number of people leaving the node and the tendency of a node to pack tighter. This tendency is referred to as *fill rate*.

Because there are interdependent relations, each time step of the model is calculated in a cascading pattern from the exits. After the flow rate out of a node is calculated, it becomes possible to calculate flow into the node. By this method the flow rate calculations can be determined for the entire graph.

Sample Graph: An Acceptable Order of Calculation:



1. 4 to 5 flow
2. 3 to 4 flow
3. 2 to 4 flow
4. 1 to 2 and 1 to 3 flow

Assumptions

There is a body of existing research that models the movement of people from one space to another. This model attempts to create patterns of movement consistent with that research. By iteratively applying those patterns of movement, the model extrapolates to movement over a larger area.

1. All humans are aware of the emergency, and all attempt to exit.
2. People move toward a single exit.
3. People move only towards the exit.
4. People are safe, and removed from the simulation, as soon as they reach an exit node.
5. People in crowds move at a speed determined by the density of the crowd.
6. People's movement is restricted by the width of the area they are trying to move over.
7. People will move to the exit as quickly as possible, without regard to the effect on crowd density.

8. The increase in crowd density over time is limited.
9. People are treated as continuous populations, allowing for fractions.

Weaknesses - Assumptions

Assumption 1 is not completely supported by the literature—not everyone is aware of or willing to leave during a real emergency (p. 716, [FAHY]).

Assumption 2 and 3 imply that people pick a single exit and head for it. In reality, people might observe that an exit is less congested and choose that exit as their new target. This assumption precludes the existence of barriers directing traffic flow or preset fire escape routes within a room.

Assumption 4 ignores the exit discharge capacity. In reality, the amount of people leaving by an exit will affect the total evacuation time for a building. With this assumption, the model is limited to a single room.

Assumptions 5 and 6 are based on literature describing pedestrian movement in a transportation terminal [BENZ]. Movement in a transportation terminal is not an escape or evacuation situation, so may involve different dynamics.

Assumption 7 does not take into account human intelligence or the possible presence of authorities regulating traffic flow.

Assumption 8 is not based on the literature. It was included to reduce the tendency of nodes to fill from empty to maximum capacity in a very short period of time.

Assumption 9 can create situations that are contrary to reality. A person cannot split into fractional parts and flow in two different directions. This assumption is loosely justified by the fact that people can be partially across the boundary of two nodes. This assumption is required by the model mechanics when using small time steps.

Mathematical Structure of the Graph Flow Model

Graph Structure:

N_i = graph node i: representing a patch of floor space

E_i = exits: the set of all nodes N_i may exit to

I_i = inputs: the set of all nodes that exit to N_i

Spatial Values:

P_i = number of people at N_i, [persons]

A_i = area of N_i, [ft^2]

Constants:

W_{ij} = bandwidth: flow rate from N_i to N_j, [persons/ft] (parameterized)

s_α = base movement constant for S_i, [dimensionless]

$s_\alpha = 58.678$ [BENZ]

s_β = movement multiplier constant for S_i, [dimensionless]

$s_\beta = 58.669$ [BENZ]

s_{min} = minimum movement at maximum compression, [feet/sec]

$s_{min} = 2.5$ feet/sec [EGAN p. 180]

T = maximum (terminal) compression of an area, [persons/ft^2]

$T = 3$ persons/ft^2 [LSC] 101-256)

r_α = fill rate constant, [feet/sec] (parameterized)

t = the time step of the model, [second] (parameterized)

Derived Constants:

A_iT = maximum occupancy of node, [persons]

Flux Capacity Equations

Let S_i denote the walking speed inside a node due to congestion, [ft/sec]

$$S_i = S(P_i, A_i) = MAX\left(s\alpha + s\beta * \ln\left(\frac{A_i}{P_i}\right), S_{\min} \right)$$

Let FR_i denote the fill rate: the maximum number of people that can be added to N_i over time t, [persons]

$$FR_i = FR(N_i, t) = r\alpha * t * \frac{A_iT - P_i}{A_iT}$$

Let OF_{ij} be the desired (maximal) outflow: the number of people capable of moving out of N_i into N_j, [persons]

$$OF_{ij} = OF(N_i, N_j, t) = t * S_i * W_{ij}$$

Let IF_i be the maximum inflow: the number of people that can enter a node from any direction in t, [persons] Note that IF_i cannot be calculated until FFA_{iEj} is calculated for all N_j in E_i.

$$IF_i = IF(N_i, t) = \sum_{E_i} FFA_{iE_i} + FR_i$$

Let FF_{ij} be the final flow: the number of people capable of moving from N_i to N_j that N_j is capable of accepting, [persons] Note that FF_{ij} cannot be calculated until IF_j is known.

$$FF_{ij} = FF(N_i, N_j, t) = \left\{ \begin{array}{ll} if & N_j \text{ is an exit} : OF_{ij} \\ if & IF_j < \left(\sum_{N_k \in I_j} OF_{kj} \right) : OF_{ij} * \frac{OF_{ij}}{\sum_{N_k \in I_j} OF_{kj}} \\ if & IF_j \geq \left(\sum_{N_k \in I_j} OF_{kj} \right) : OF_{ij} \end{array} \right\}$$

Let FFA_{ij} denote the actual number of people who move from N_i to N_j. Note that FFA_{ij} cannot be calculated until FF_{iEj} is calculated for all N_j in E_i.

$$FFA_{ij} = FFA(N_i, N_j, t) = \left\{ \begin{array}{l} if \quad P_i \geq \left(\sum_{k \in E_i} FF_{ik} \right) : FF_{ij} \\ if\ P_i < \left(\sum_{k \in E_i} FF_{ik} \right) : P_i * \frac{FF_{ij}}{P_i < \left(\sum_{k \in E_i} FF_{ik} \right)} \end{array} \right\}$$

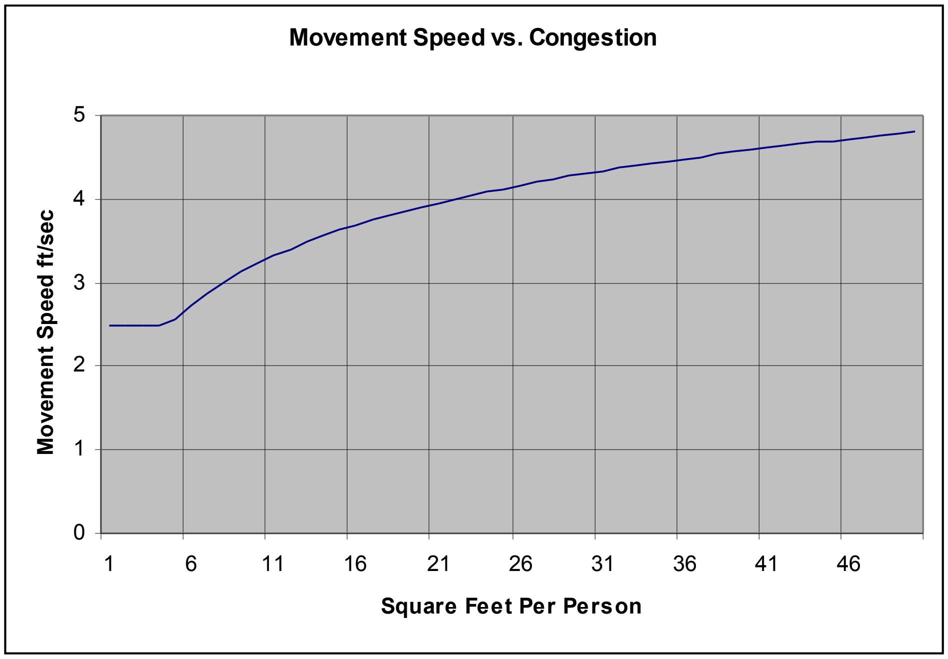
This is a pool-flux model. P_i are pools. FFA_{ij} is the only flux that is ever applied to a pool.

Development of the walking speed function

Walking speed is a compromise between the regression obtained using data from the Benz paper, which describes a speed function that reaches 0 for very tight congestion and the guidelines found in Egan, which specify a range of 2.5 ft/sec to 4.0 ft/sec.

A speed of 0 would result in an infinite total exit time. In order to ensure that traffic cannot come to a complete stop, the lower bound of the speed function is set to 2.5 ft/sec. Since the Benz speed function is logarithmic, and gives realistic speeds that fall within the bounds set by Egan, no upper bound is proscribed by the model.

This results in the following speed function:



Development of Bandwidth Parameter

Benz observed that the flow of pedestrians through space decreased sharply as congestion increased. Our model matches that congestion using the speed function, but requires a parameter to determine the maximum flow between nodes. Benz observed a maximum flow of 26 pedestrians per foot width per minute with 5 square feet area per pedestrian. This is approximately .4333 persons per foot width per second, which is our time scale. Our bandwidth constant is in persons per foot of linear movement, so we set W_{ij} so that OF_{ij} for 5 X 5 foot grid areas is consistent with the Benz data at 5 ft^2 per person. This yields:

W_{ij} = .541 for the 5 foot bandwidth of two nodes

W_{ij} = .325 for the 3 foot bandwidth of a single exit.

These values will allow .4333*5 = 2.1665 people to flow across a 5 foot edge in one second at a 4.0 ft/sec flow and .4333*3 = 1.299 people to flow across a 3 foot edge in one second at a 4.0 second flow.

Based on the very limited real life experiments carried out by this team, we find these flow rates to be plausible. In addition, these flow rates create flows that are consistent with the available data.

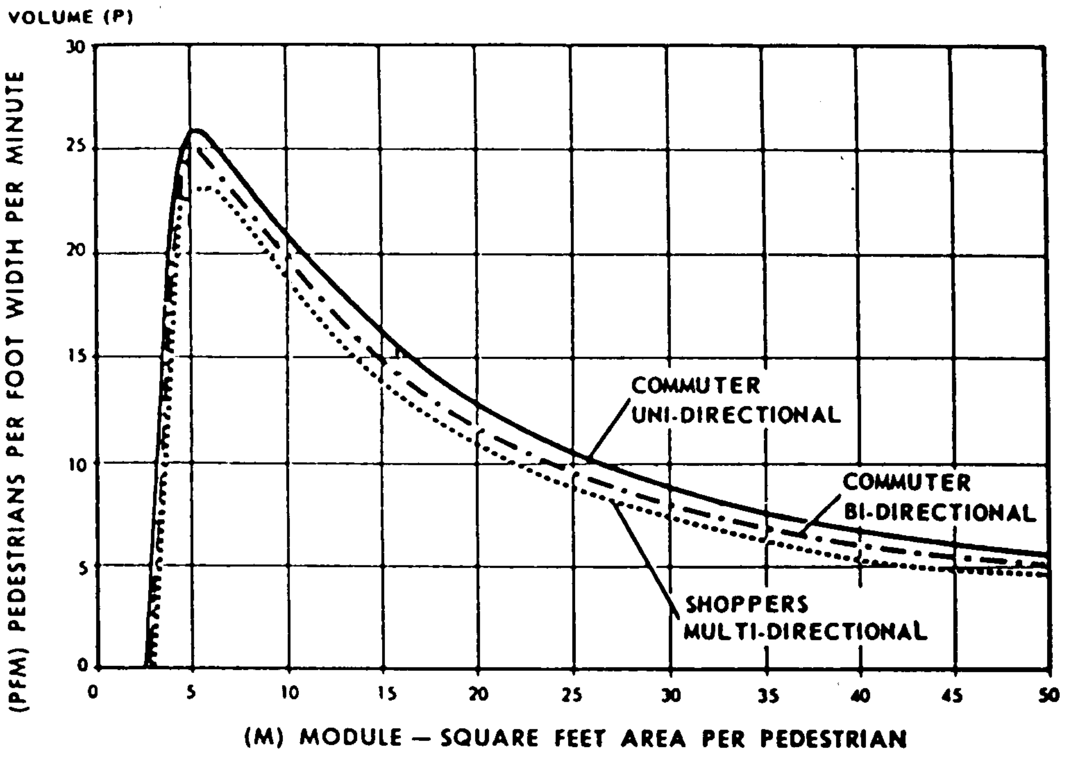
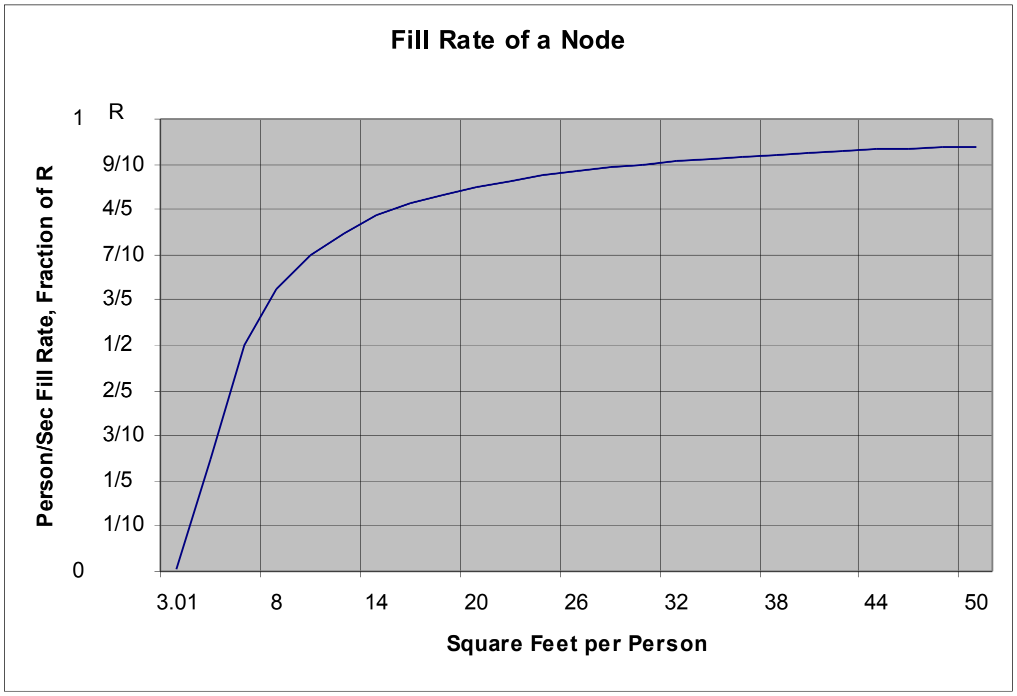


FIGURE 2 **Relationship between pedestrian flow and space.**

*[BENZ, 1986]

Development of the Fill Rate Function

The fill rate function is designed to represent the pattern of increasing congestion. In general, an empty node should fill quickly, but as the congestion inside a node increases, the fill rate should decrease. The fill rate equation satisfies this property, but requires a fill rate parameter.



As the above graph demonstrates, the fill rate parameter represents the maximum fill into an uncongested node. Therefore, the fill rate should W_{ij} * (the length of the border from N_i to N_j). Ideally, the fill rate should be a function of N_i and N_j. Since our nodes represent square areas and exits do not use the fill rate equation, then fill rate can be constant for all nodes and not contradict any of the assumptions.

For the simulations carried out in this paper, fill rate was set to:

$$\mathbf{r_\alpha = 4.333 \text{ persons/sec}}$$

This corresponds to the flow across two edges with 5 foot bandwidths.

Development of the Flow Functions

Recall that OF_{ij} is the desired (maximal) outflow: the number of people capable of moving out of N_i into N_j, [persons]

$$OF_{ij} = OF(N_i, N_j, t) = t * S_i * W_{ij}$$

For constant t and W_{ij}, OF_{ij} is linearly proportional to walking speed due to congestion (S_i). For constant t and S_i, OF_{ij} is linearly proportional to bandwidth (W_{ij}).

Recall that IF_i is the maximum inflow: the number of people that can move into a node from any direction in t, [persons] Because IF_i is a function of the actual final flows out of a node, IF cannot be calculated until these actual final flows have been calculated first. These final flows are a function of the IF for the nodes that N_i flows into. Because of this dependency, the node graph must be acyclic. If the graph contains a cycle, then no IF_i for any node that is a member of the cycle can be calculated because it is dependent on IF for another node that is a member the cycle.

$$IF_i = IF(N_i, t) = \sum_{E_i} FFA_{iE_i} + FR_i$$

IF is equal to the total number of people that flow out of a node plus the fill rate for that node.

Recall that FF_{ij} is the final flow: the number of people capable of moving from N_i to N_j that N_j is capable of accepting, [persons] FF_{ij} is a function of IF_j, unless Nj is an exit.

$$FF_{ij} = FF(N_i, N_j, t) = \left\{ \begin{array}{c} if \quad N_j \ is\ an\ exit : OF_{ij} \\ if\ IF_j < \left(\sum_{N_k \in I_j} OF_{kj} \right) : OF_{ij} * \dfrac{OF_{ij}}{\sum_{N_k \in I_j} OF_{kj}} \\ if \quad IF_j \geq \left(\sum_{N_k \in I_j} OF_{kj} \right) : OF_{ij} \end{array} \right\}$$

FF depends on IF_j and the total OF of all nodes that flow into N_j. If the total desired flow from of all notes that flow into N_j is greater than IF_j, then the final flow from N_i to N_j is the desired flow from N_i to N_j times the ratio of desired flow from N_i to N_j to the total desired flow from all nodes that flow into N_j.

Recall that FFA_{ij} is the actual final flow: number of people who move from N_i to N_j. FFA_{ij} cannot be calculated until the final flow from N_i to all nodes that N_i flows into is calculated.

$$FFA_{ij} = FFA(N_i, N_j, t) = \left\{ \begin{array}{c} if \quad P_i \geq \left(\sum_{k \in E_i} FF_{ik} \right) : FF_{ij} \\ if\ P_i < \left(\sum_{k \in E_i} FF_{ik} \right) : P_i * \dfrac{FF_{ij}}{\left(\sum_{k \in E_i} FF_{ik} \right)} \end{array} \right\}$$

If the total final flow from N_i to all nodes that N_i flows into is less than the population of N_i, then the actual final flow is equal to the final flow for all notes that N_i flows into. If the total final flow from N_i to all nodes that N_i flows into is greater than the population of N_i, then the actual final flow is equal to the population times the final flow from N_i into N_j divided by the total final flow from N_i to all nodes that N_i flows into.

The relationship between final flow and actual final flow is straightforward. Final flow calculates the number of people that can flow out of a node. However, if final flow is more than the population of the node, then this population is divided evenly among the available final flows. Otherwise, actual final flow is equal to final flow.

Particle Simulation Model

Overview

The particle simulation model is based on bottom-up assumptions. The simulator models humans one at a time as discrete, independent entities, instead of treating a flow of people as an undifferentiated group.

The simulation begins with a single, 2D room at the start of an emergency. People in the room each choose a visible nearby exit and walk toward it. People navigate obstacles such as furniture, and, if crowded together, interact with one another. The simulation continues until everyone has reached an exit.

Individual humans (especially during an emergency) are primarily concerned with getting to an exit, greedily maximizing their own chance of survival. This model thus operates on a local level, allowing the overall global properties (such as total exit time and walking speed vs. congestion) to emerge. In addition, human flow rate values, which must be assumed in the graph-flow model, are natural results of the interactions of the particle simulator.

Assumptions

Although human behavior is in general very complex, the modeling task is substantially simplified in a crowd during an emergency. Still, the primary weaknesses of this model lie in its restrictive and somewhat arbitrary assumptions.

1. All humans are aware of the emergency, and all attempt to exit.

2. People pick exits based on congestion (number of people near that exit), distance, and visibility—people cannot see through walls. Occasionally, people check for a better exit.
3. People are safe, and removed from the simulation, as soon as they reach an exit.
4. People walk at 4 ft/second.
5. People may change direction and speed instantly.
6. If a person's intended path would pass through another person, that person stops and tries to go in some other direction.
7. People cannot walk through walls or furniture. For these purposes, people are treated as disks.
8. People plan a path around furniture to reach an exit.

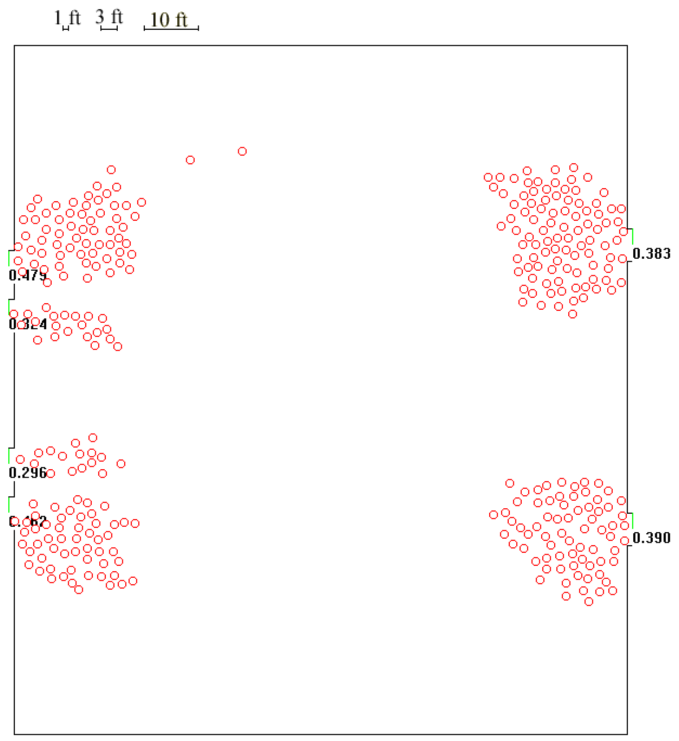
Weaknesses—Assumptions

Assumption 1 is not completely supported by the literature—not everyone is aware of or willing to leave during a real emergency (p. 716, [FAHY].) Assumption 2 is more restrictive than reality—humans remember the location of out of view exits, and often "follow the crowd" to an exit they can't see. Assumption 3 neglects the finite person-handling capacity of many exits (e.g. narrow stairwells.)

Assumption 4 neglects the very young, old, or handicapped, who may move more slowly, as well as the panic-stricken, who move more quickly. Assumption 5 is contrary to basic physical principles, but significantly simplifies interactions. Assumption 6 neglects people's sophisticated path planning, which allows us to (usually!) avoid walking into each other without stopping. Assumption 7 treats people as hard, inelastic 2D disks. Assumption 8 neglects the panic-stricken, who may in fact run directly into furniture.

Example

Despite the disadvantages outlined above, these assumptions produce behavior which is remarkably crowd-like, and consistent with research data. Also see the Appendix--Applications.

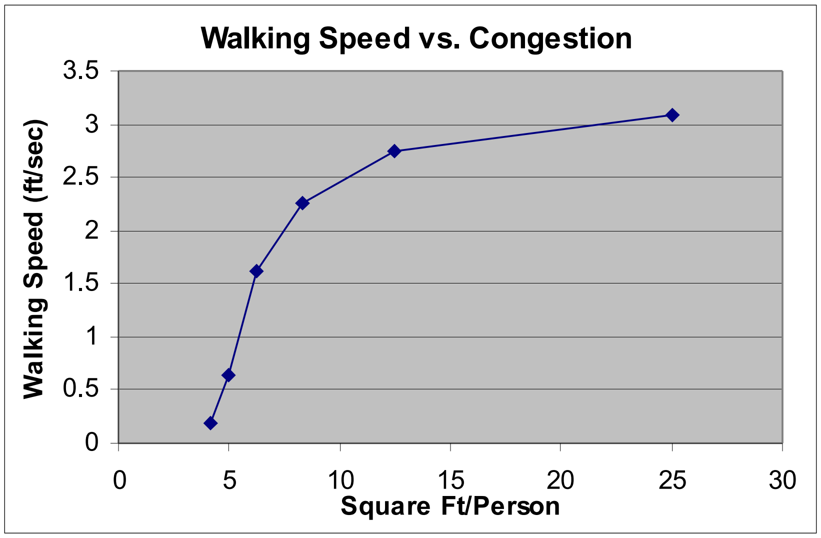


The simulation above shows 400 people in the process of leaving a gym. The loose clumping around the exits is a natural result of people's desire to go towards the exit, but aversion to running into each other. In this example, the people begin the emergency distributed evenly across the 110x120 foot floor. Moving independently, the people quickly form groups near the exit. As people near the exit flow out, the groups shuffle around to bring more people to the exit. There is a substantial amount of movement, where people try to advance but, blocked by other people, shift a bit and try again. All this emergent behavior is consonant with our experience in groups of people.

Walking Speed

Research on airline terminals (p. 17, [BENZ]) indicates that human walking speed decreases quickly as the amount of floor space per person decreases. This results from people slowing down and taking smaller steps when someone is immediately front of them.

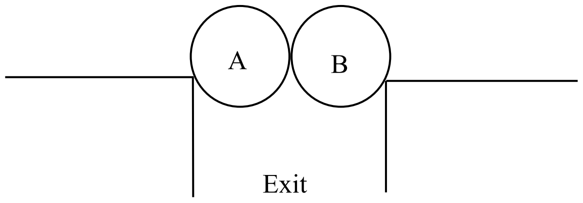
In an open-floor simulation, we recorded the total distance traveled towards the destination and approximate amount of floor space per person during one evacuation. Averaging the results leads to the graph below (see the data for this graph in the table appendix.) This graph is nearly identical to the observed biometric data in [BENZ] (see Appendix—Walking Speed.)



Human Interaction in Crowds

Note that the overall result emerges only from our single, simple assumption about how people interact. That assumption—if your intended path will intersect another person, stop and try another (random) direction—drives behaviors similar to those of people in crowds. We analyzed many potential ways for people to interact, but other assumptions produced decidedly non-human behavior.

We would have preferred to pick a deterministic interaction, because we would rather not have the results of our model change with each execution of the model. For each deterministic interaction we considered (e.g. if your path will intersect someone, go around them to your right), we could always find cases that created a circular-wait condition. This situation, in which object A waits for object B, who in turn waits on object A, is known to computer scientists as deadlock.



Example: A and B have wedged themselves into an exit. A can't go forward because of B; and B can't go forward because of A. Someone needs to back up.

By suitable arrangement of furniture and exits, we found a way to produce deadlock with each deterministic interaction. Thus, with hundreds or thousands of people participating in an evacuation, we would often find cases where dozens of people had locked themselves together against the walls or furniture, each waiting for the other to move—this is not at all what humans do.

People don't do the same thing every time—if one attempt doesn't help, we try another. But while real people use consciousness to avoid deadlock, our simulated people use randomness. In the situation above, A and B would see that their intended paths were blocked (by one another), stop, and try another, random direction. If that didn't help (for example, if they both decided to move forward again) they would try again. Eventually (normally within a few seconds) A and B would blunder their way free, then make their way out the door successfully.

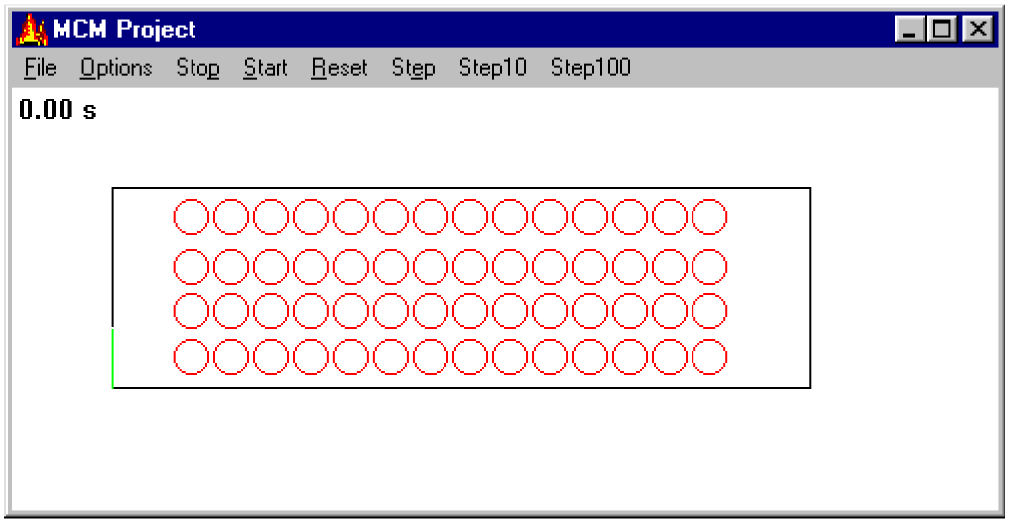
We can make the model give us the same results each time by using a <u>pseudo</u>random (deterministic, but still spatially uniform and statistically uncorrelated) number generator to pick directions.

Thus our randomized interaction scheme runs the same way each time, yet produces behavior that is reasonably similar to that of actual people—for example, they don't deadlock.

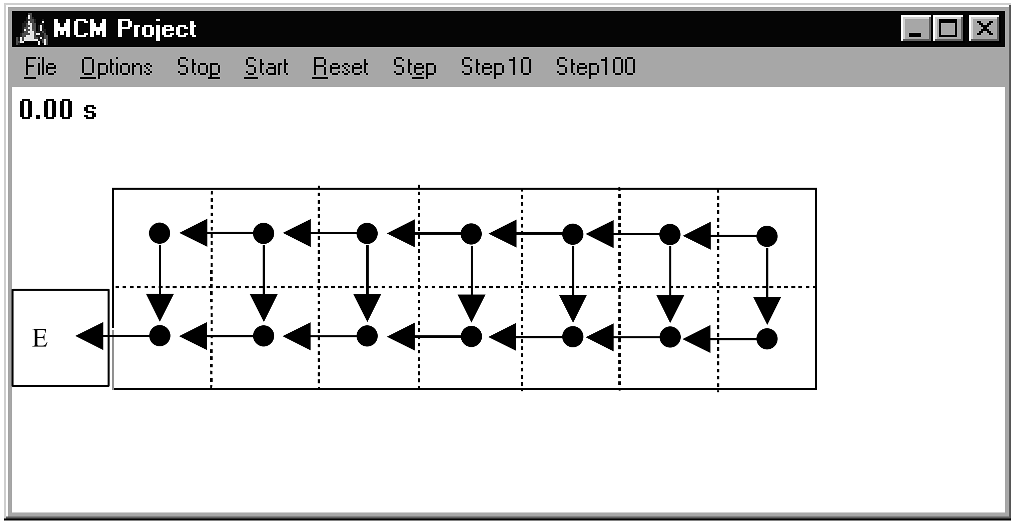
Test Scenarios and Model Validation

Both the particle model and the graph flow model were used to evaluate exit time from two test rooms. Each test room has one exit. One test room is 10 feet by 35 feet and has a 3 feet wide exit at one end. The other test room is feet 15 by 15 feet and has a 3 feet wide exit in the center of the left wall. Each model was run repeatedly, using a different occupancy for each run.

Below is a sample initial state of the particle simulation model for the 10 feet by 35 feet room:

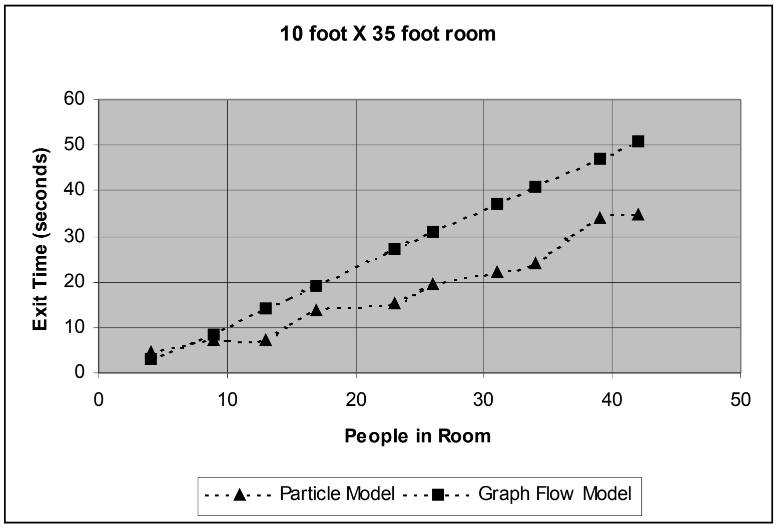


The graph flow model was applied to a space that was equal in size to the particle flow model space for the 10 feet by 35 feet room. Nodes and edges were created as indicated in this diagram:

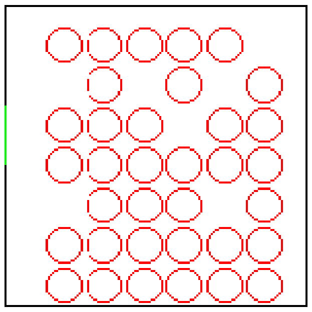


After both models were executed repeatedly for different room occupancies, the following results were obtained for the 10 feet by 35 feet room:

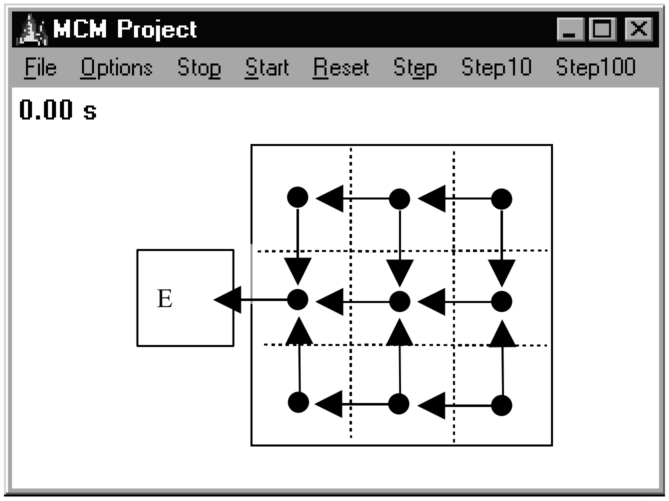
Number of People	Particle Exit Time	Graph Exit Time
4	5.2	2.8
10	9.2	10
16	13.6	17.8
24	19	27.8
33	21	38.9
35	28.8	41.4
41	32.2	48.9
42	34.4	50.1
47	37.4	56.2
56	44.2	67.4



Below is a sample initial state of the particle simulation model for the 15 feet by 15 feet room:

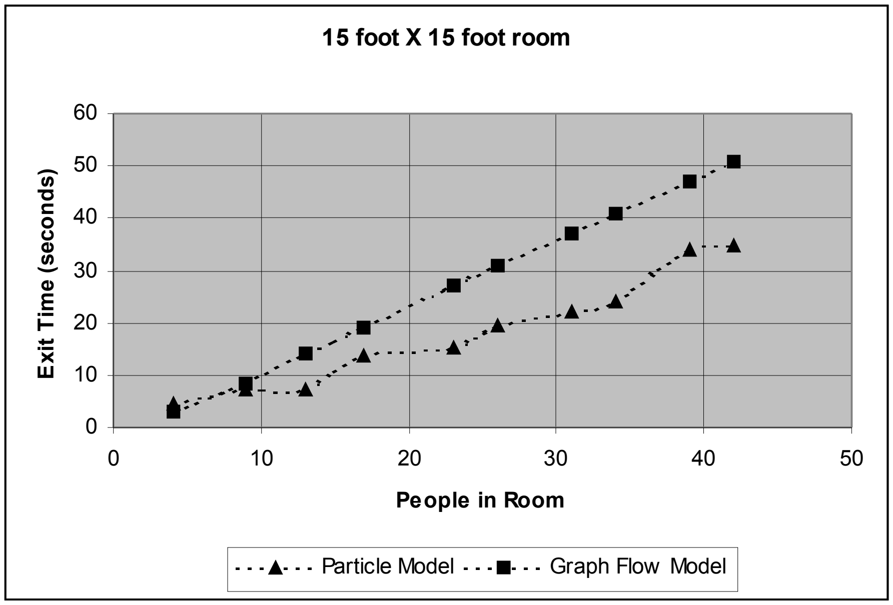


The graph flow model was applied to a space that was equal in size to the particle flow model space for the 15 feet by 15 feet room. Nodes and edges were created as indicated in this diagram:



After both models were executed repeatedly for different room occupancies, the following results were obtained for the 10 feet by 35 feet room:

Number of People	Particle Exit Time	Graph Exit Time
4	4.4	3.1
9	7.2	8.5
13	7.4	14
17	13.6	19
23	15.4	27
26	19.4	31
31	22	37
34	24	41
39	34	47
42	34.6	51



Analysis of test results

We found a simple, straight-line, linear regression for each set of model results. The results of the regression were:

10 feet X 35 feet room:		
Graph Flow Model	$y = 1.2728x - 2.4338$	$R^2 = 0.9997$
Particle Flow Model	$y = 0.8106x - 1.093$	$R^2 = 0.9599$
15 feet X 15 feet room:		
Graph Flow Model	$y = 1.2444x - 2.199$	$R^2 = 1$
Particle Flow Model	$y = 0.7559x + 1.2175$	$R^2 = 0.9785$

The results of both models appear to be very nearly linear for both test rooms. However, the slope of the nearest linear approximation of each model differs. Since both models are driven by arbitrary parameters, specifically bandwidth for the graph-flow model and person radius for the particle model, it is not suprising that this difference exits.

We consider it significant that both models display similar trends. Each model was derived from an independant set of driving assumptions and data, but the behavior trends of the models are strongly correlated. This reflects positively on both models.

Strengths/Weaknesses

The graph-flow model has several weaknesses. It treats people not as indivisible entities, but as a fluid. The results of the graph-flow model depend on an arbitrary choice of bandwidth. The results of the graph-flow model depend on the source graph, and we did not address the problem of building this graph.

However, human behavior in the graph-flow model is deterministic. Much of the mathematical structure of the graph-flow model is driven by actual research.

The particle simulator model has several weaknesses. Its results are a function of an arbitrary choice of radius. Its decisions are non-deterministic, so they can vary significantly for tiny input changes. The model is also occasionally subject to pathological, non-human behavior—for example, people occasionally lose sight of a nearby exit and travel a long distance to a visible exit.

The particle simulator model, however, also has several advantages. It models people as individual, indivisible entities. People can move independently of their neighbors. No assumptions need be made about the global flow in the room.

Conclusion

We have presented two models for determining the amount of time it takes a given number of people to evacuate a room. Despite their very different approaches and assumptions, both models substantially agreed on our test cases.

Based on our test cases, and the analysis of several local rooms (Appendix—Applications) we noted that a time-to-exit vs. initial population graph is nearly linear. The actual slope of the line depends on the layout of the room and the size and number of exits. Still, evacuating 200 people from some space takes very nearly twice as long as evacuating 100 people. This implies that the average exit rate is nearly constant.

We expected the exit rate to decrease as more people tried to pack into the exits; but the actual exit rate (for both models) remained constant. We attribute this to the fact that exits become congested very quickly, even if only a few dozen people are attempting to exit. This is in agreement with our experience—it doesn’t take many people (under a dozen) to effectively block an exit.

We found that the posted maximum occupancies of the local buildings we simulated were adequate. At maximum occupancy, everyone evacuated under 3 minutes; an acceptable evacuation time [LSC, Section A-21-1.3].

To determine the maximum occupancy of a room, we suggest first consulting a Fire Marshall to determine the maximum acceptable time for a total evacuation. Then, using the simulator experiment with different occupancy rates until you find the largest number of people who can escape in less than the maximum time. This is easy because the function relating the number of occupants to evacuation time is nearly linear.

Future Plans

The most significant addition to our models would be to take into account the actual disaster that is causing the evacuation—a nearby fire creates a very different evacuation (smoke avoidance, flame fleeing) than an earthquake or bomb threat. This might eliminate our dependence on the Fire Marshall. We would also like to be able to model other situations, such as the sloping floors of a stadium or rowdy fans at a soccer game.

The area with the biggest room for improvement in the particle model is human behavior—the simulated people need to be made smarter.

The graph-flow model needs a good way to contruct the overall graph. In addition, this model's parameters (such as bandwidth) and equations (such as OF) could be adjusted to better match reality.

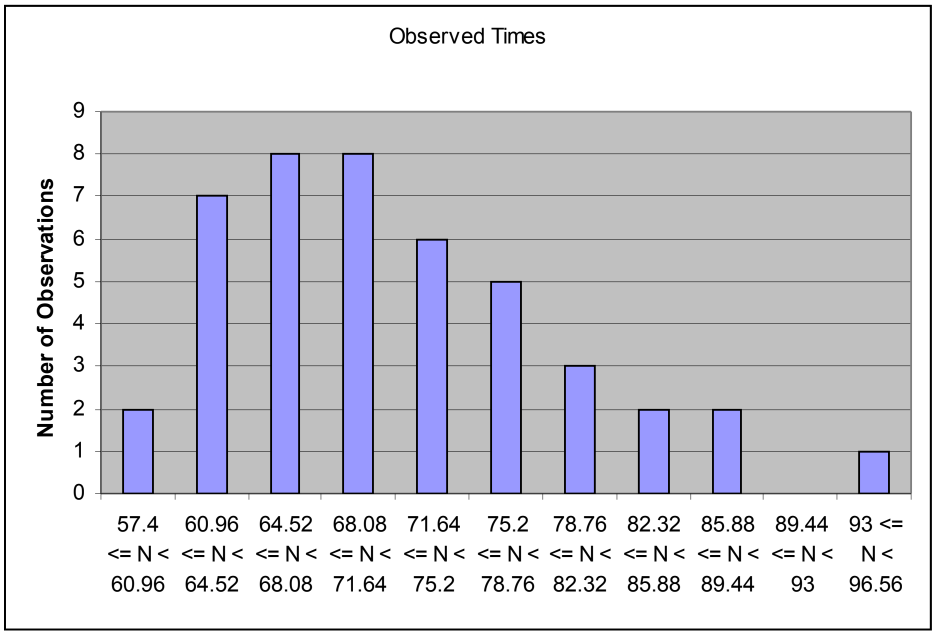
Stability Analysis

The graph-flow model is purely deterministic and its senstitivity to parameterization at any given point is also purely deterministic. These attributes make it unsuitable for stability analysis.

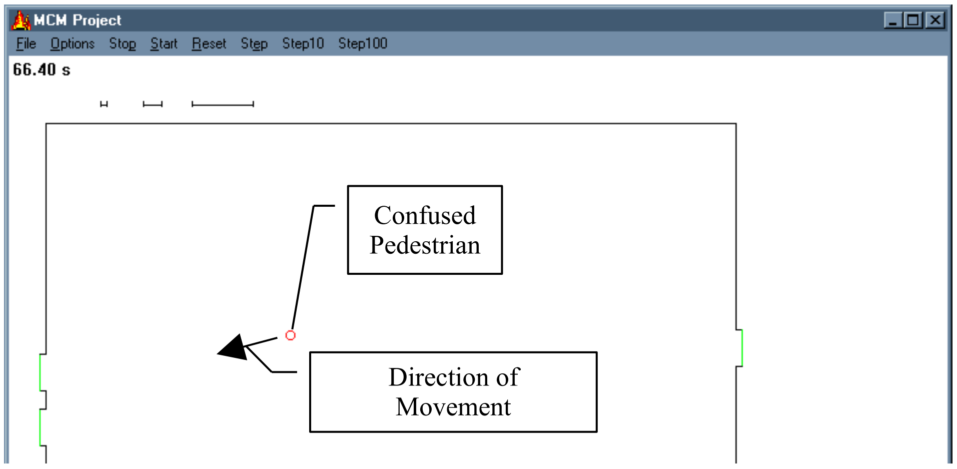
The particle model is not purely deterministic. The pedestrian collision avoidance algorithm used random vectors. Pedestrians that cannot see an exit move in a random direction. Pedestrians that collide with a wall attempt to move away in a random direction. Pedestrians retarget at random intervals. These attributes make the mode suitable for stability analysis.

For stability analysis, the particle model was run for the gymnasium with 435 people roughly evenly distributed over the floor space. The model was run 43 times, and the standard deviation of the final exit times was calculated. Final exit times were plotted as a histogram.

Minimum exit time was 57.4 seconds.
Maximum exit time was 93 seconds.
Mean exit time was 70.6 seconds.
Median exit time was 71.37674419 seconds.
Standard deviation of the runs was 7.782423334.



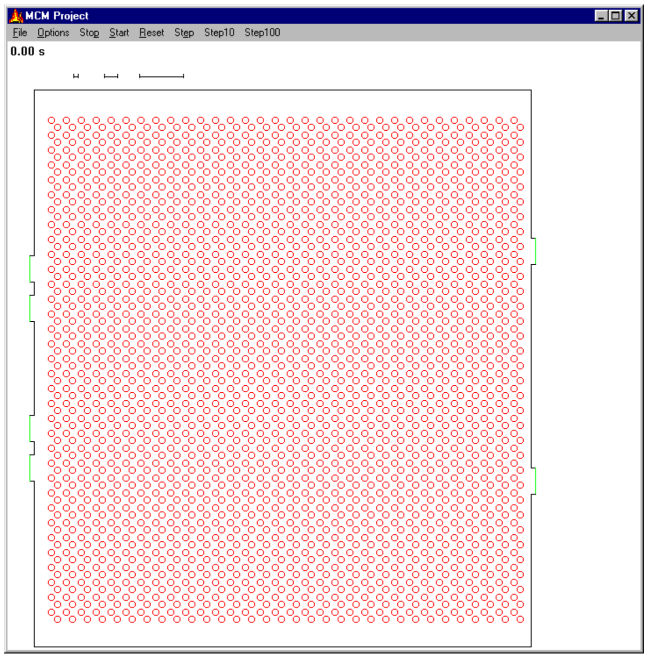
The histogram of observed times appears skewed. After repeated observation of the model, we identified a behavior that produced this skew. In the later stage of the model, a person at the back of a crowd sometimes retarget away from the congested exit. Then, when the person is nearly halfway across the room, the exit clears and the person retargets and heads back to the original exit. This results in the following situation:

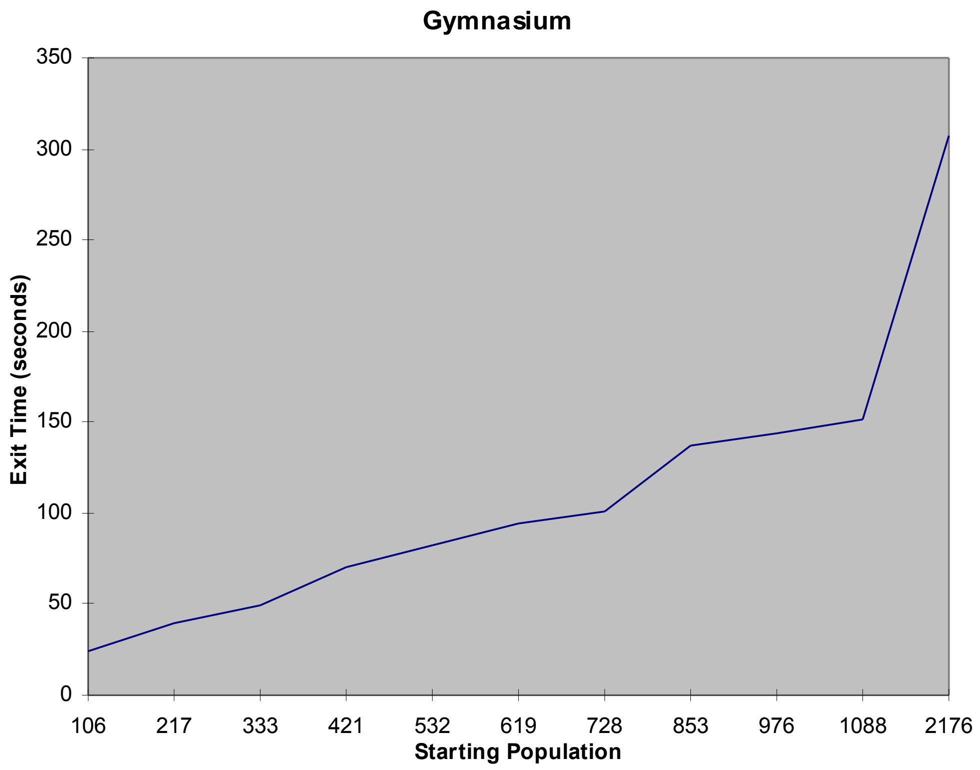


Note that all of the exits have been cleared, but the final exit time is delayed until this last straggler leaves the room. This uncommon condition will inflate the final exit time over the final exit time that would otherwise be obtained.

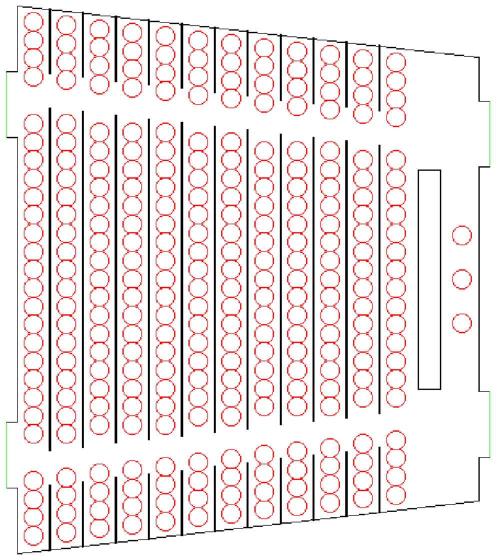
Appendix-- Applications

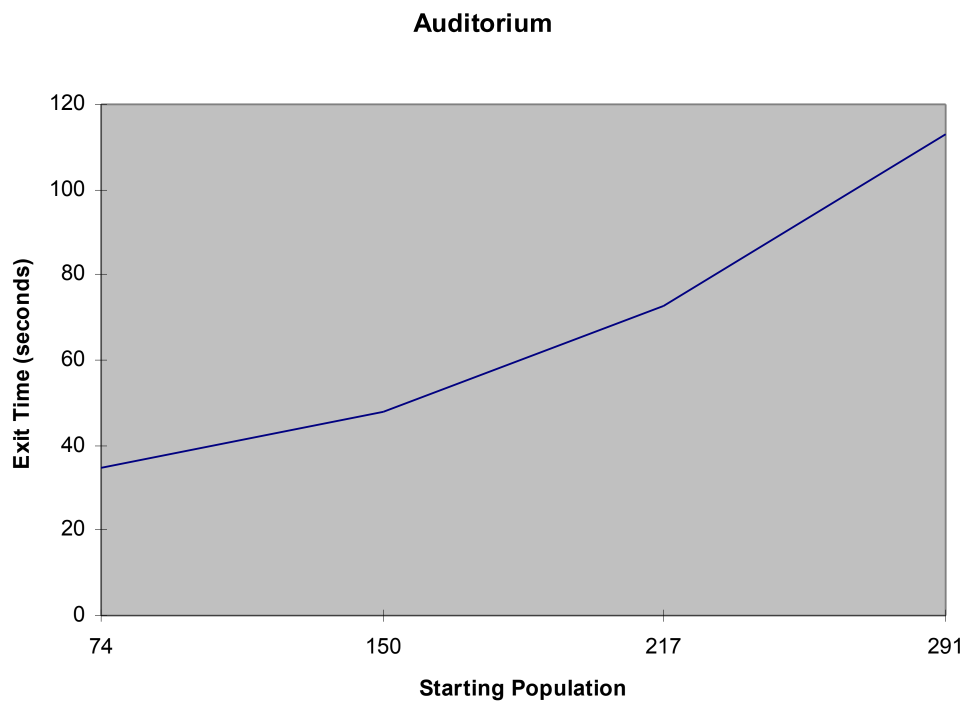
Gymnasium



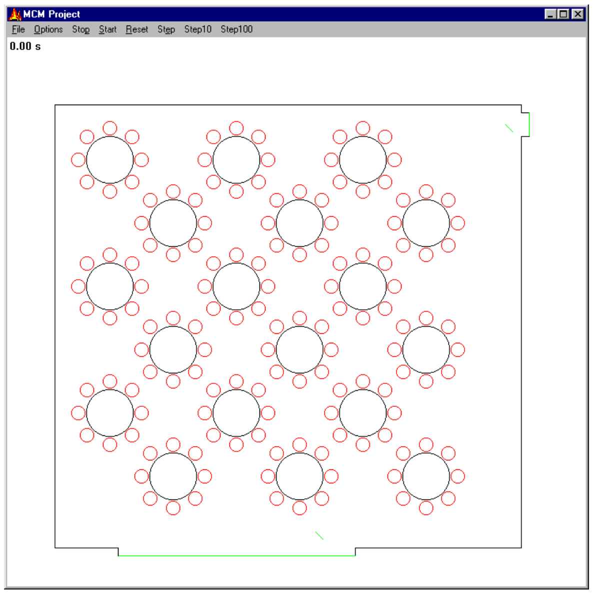

Gymnasium
Exit Time (seconds)
0
50
100
150
200
250
300
350
Starting Population
106
217
333
421
532
619
728
853
976
1088
2176

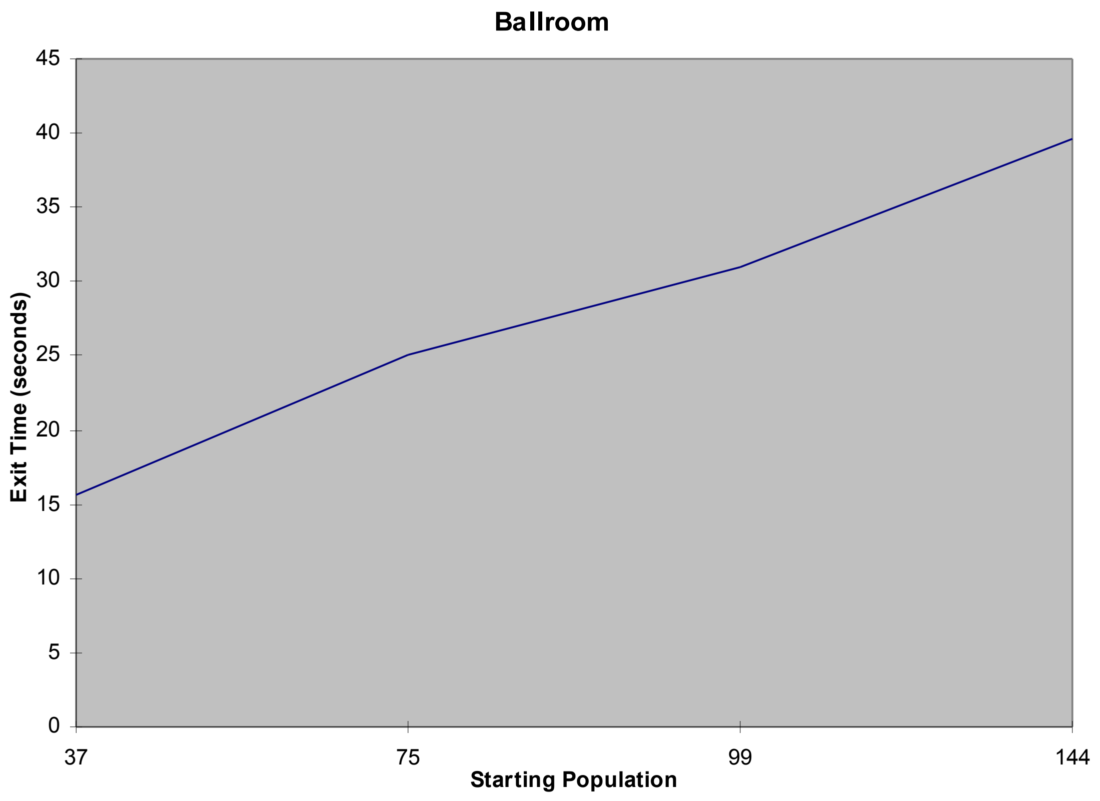
Auditorium



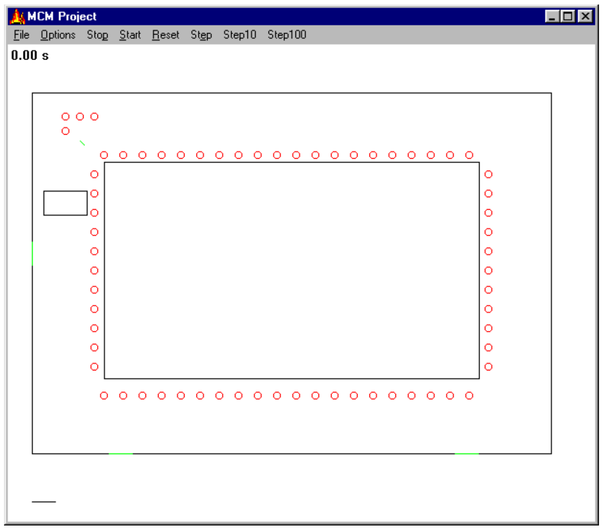

Auditorium
Exit Time (seconds)
0
20
40
60
80
100
120
74
150
217
291
Starting Population

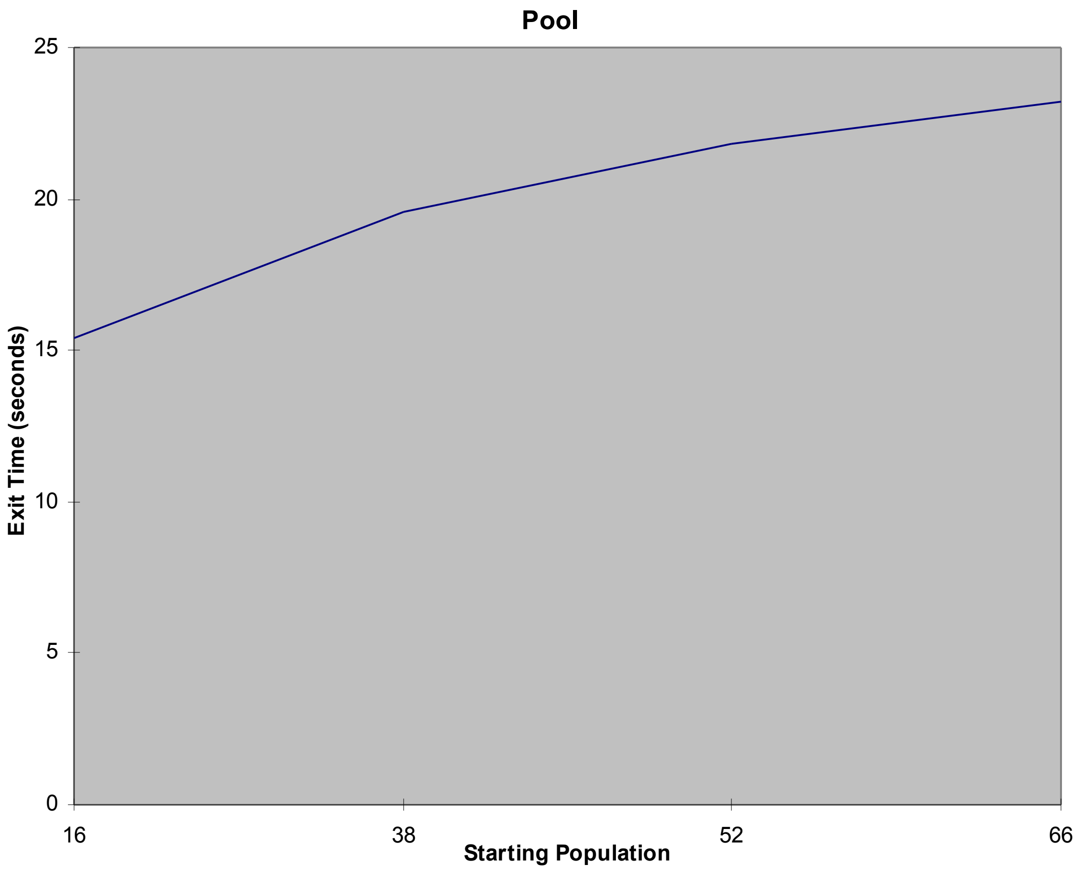
Ballroom


MCM Project
File Options Stop Start Reset Step Step10 Step100
0.00 s


Ballroom
45
40
35
30
25
20
15
10
5
0
Exit Time (seconds)
37
75
99
144
Starting Population

Pool


MCM Project
File
Options
Stop
Start
Reset
Step
Step10
Step100
0.00 s


Pool
Exit Time (seconds)
Starting Population
0
5
10
15
20
25
16
38
52
66

Appendix—Legal Maximum Occupancy for our Spaces

Case Studies

- Gymnasium:

  With bleachers closed:

  floor area = 114 ft X 127 ft = 14,478 ft^2

  capacity = 14,478 ft^2 / 15 ft^2 per person (*1) = 965 people

  With bleachers open:

  floor area = 65 ft X 114 ft = 7,410 ft^2

  bleacher area = 2 sides * 9 bleachers/side * 112 feet = 2,016 linear bleacher feet

  capacity = 7,410 ft2 / 15 ft^2 per person (*1) + 2,016 linear bleacher feet / 18 linear inches per person (*2) = 1839 people
- Lecture Hall

  fixed seating = 288 seats

  lecture area = 4 ft X 25 ft = 100 ft^2

  capacity = 288 seats * 1 person / seat (*3)+ 100 ft^2 / 20 ft^2 per person (*4) = 303 people
- Pool

  water surface = 75 ft X 108 ft = 8100 ft^2

  deck area = 1528 ft^2

  capacity = 8100 ft^2 / 50 ft^2 / person (*5) + 1528 ft^2 / 20 ft^2 / person (*6) = 212 people
- Ballroom Hall (Banquet Seating)

  fixed seating = 144 seats

  capacity = 144 seats * 1 person / seat (*3) = 144 people

(*1) 8-1.7.1 (b), p101-65 of LSC, assembly area of concentrated use
(*2) 8-1.7.1 (c), p101-65 of LSC, bleachers, pews, and similar bench-type seating
(*3) 8-1.7.1 (d), p101-65 of LSC, fixed seating
(*4) 10-1.7.1 (a), p101-97 of LSC, non-vocational classroom area
(*5) 8-1.7.1 (g), p.101-65 of LSC, swimming pools, water area
(*6) 8-1.7.1 (g), p.101-65 of LSC, swimming pools, deck area

Comparison of Calculated Maximums and Maximums Currently in Use

Area	Our Calculated Maximums vs.	Posted or Actually Used Maximums
Gymnasium	965, 1839	2200
Lecture Hall	288	N/A
Pool	212	60
Ballroom Hall	144	150

Appendix—News Article

MCM Team #375
Modeling HQ
Modeltown, Earth

Editor
The Daily News-Cycle

Dear Editor,

As part of the annual SIAM Mathematical Contest in Modeling, we have performed an analysis of the maximum capacity of some of the prominent spaces on our campus. We would greatly appreciate your consideration of the following editorial article that summarizes our results.

Sincerely,
MCM Team #375

We of MCM Team #375 believe that your safety is in good hands. We have tackled the problem of maximum room capacity and determined that it requires only a simple multiplication to estimate the amount of time it takes for people to safely exit a room.

We have developed two independent mathematical models for room evacuation, and both models predict that people exit at a constant, steady rate for the entire evacuation. This mirrors reality; in an evacuation, the exits tend to be more packed than any part of the room and the slowest part of an evacuation is getting people out of the exits.

We modeled our local Lecture Hall, Ballroom, and Swimming Pool. In all three cases, when the space was filled to the legal capacity, the evacuation proceeded smoothly in under 3 minutes, and as any Fire Marshal will tell you, that's a good evacuation time.

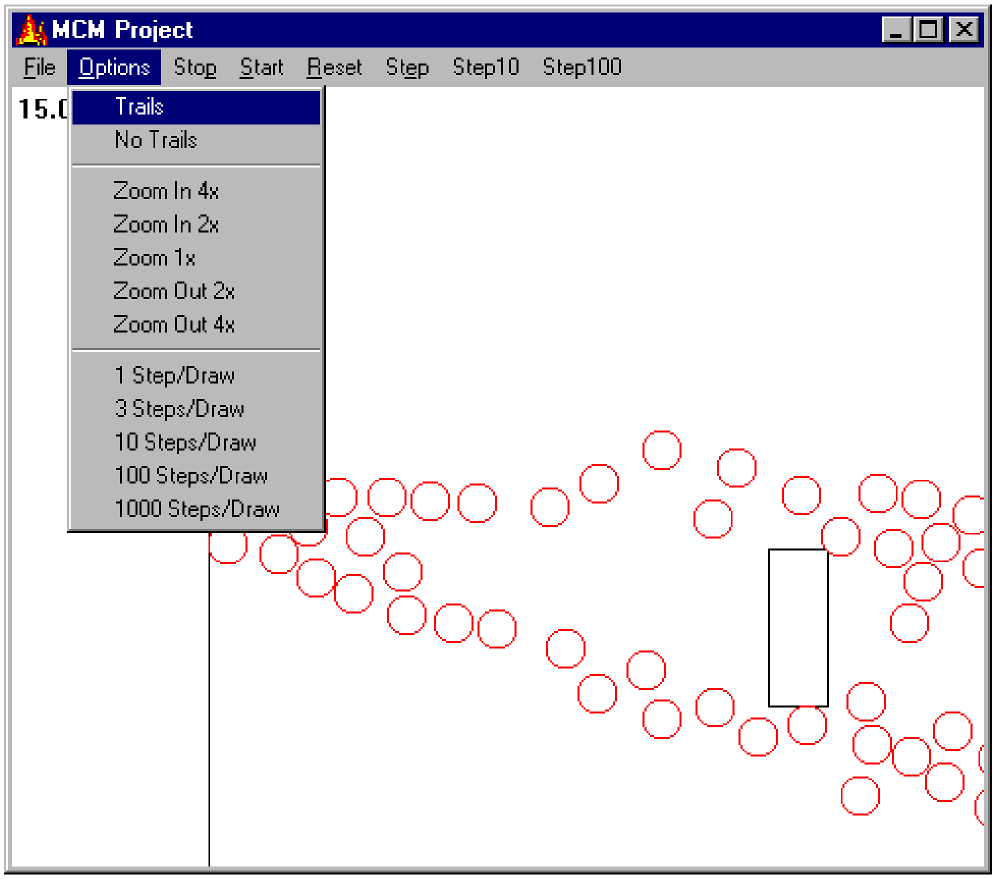
Even when modeling our local gymnasium, with people packing the bleachers to capacity and a court full of athletes, evacuation took just over 5 minutes, which is an acceptable evacuation time.

Our conclusion is that the maximum capacities enforced by our local chapter of the International Conference of Building Officials are safe.

Appendix—Particle Simulator Model Implementation

Interface

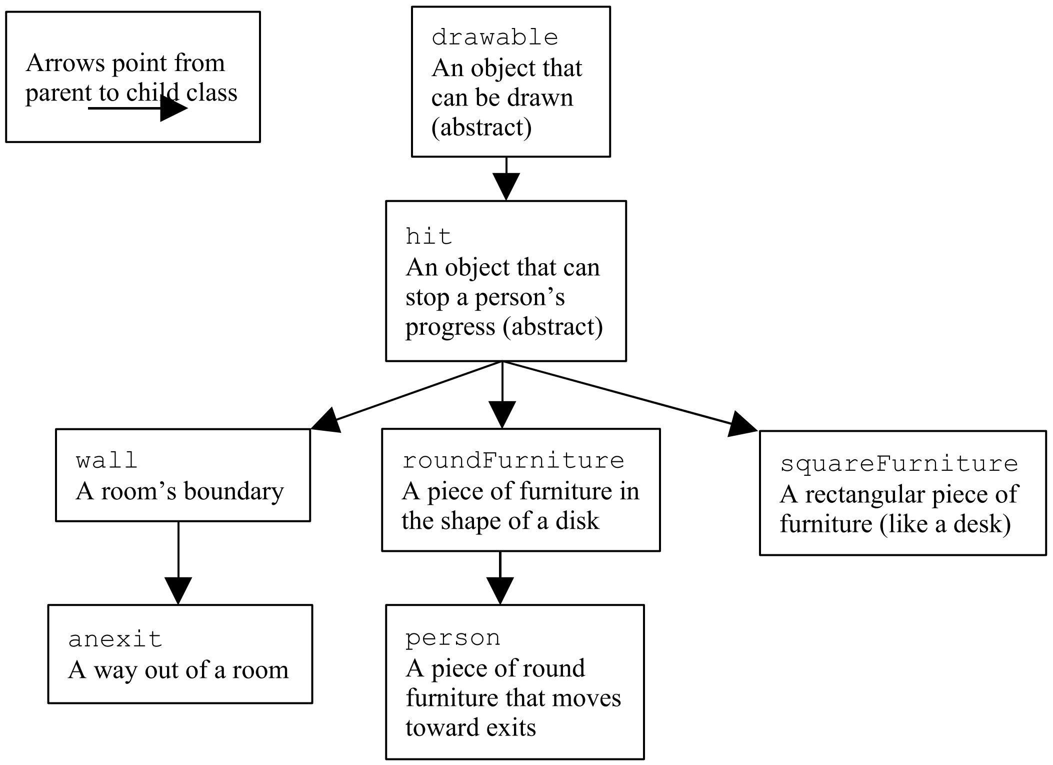
The particle simulator is written in C++, and uses Win32 API calls to keep a live picture of the simulator progress on the screen. User interaction is performed via standard Windows menus.



Object Structure

The simulator is comprised of a group of C++ classes. One overriding class, `disaster`, contains lists of the other classes. The other classes are organized in the inheritance structure on the next page.

C++ Object Inheritance Structure


Arrows point from parent to child class
drawable
An object that can be drawn (abstract)
hit
An object that can stop a person's progress (abstract)
wall
A room's boundary
roundFurniture
A piece of furniture in the shape of a disk
squareFurniture
A rectangular piece of furniture (like a desk)
anexit
A way out of a room
person
A piece of round furniture that moves toward exits

All objects that can be drawn are subclasses of `drawable`. All objects that people can run into are subclasses of `hit`. Since every object shows up on the screen and can stop a person, this describes every class. The object "family tree" then splits off into three branches—walls and two types of furniture. `person` is a subclass of `roundFurniture`, because people are modeled as disks.

Simulation Main

The simulator begins by reading a text input file which describes the location of each wall, exit, piece of furniture, and person. In the main loop of the simulator, each person determines their desired destination, ensures that no object is in the way, and moves one "step" toward the destination. Time advances by 1/5 of a second at each iteration.

The simulator pseudocode is then:

```
create disaster class
read input file, placing objects into disaster
while people still in simulation
    for each person
        pick a destination
        plan one step to the destination
        if a wall or furniture impedes our next step
            plan a shorter step
        for all other people
            if other people impede our next step
                don't move
        take step
        if we have reached exit
            remove person
```

Note that the simulator, as written above, checks every person's step against every other person, and hence runs in O(n^2), where n is the number of people in simulation. With several thousand people, this becomes a severe computational bottleneck.

We eliminate this bottleneck by superimposing a grid on the simulation, and maintaining a list of people in each grid cell. Since people only interact if they are next to one another, each person needs only to check their next step against the people in adjacent grid cells. The performance is then O(bn), where b is the number of people in adjacent grid cells. With a 5x5-foot grid resolution, since people take up at least 3 square feet, no more than 9 people may fit in each of the 9 adjacent grid cells. Thus *b* is never more than 81—a definite an improvement over *n*, which may be hundreds or thousands.

Including the check against each inanimate object in the room, we arrive at a per-iteration computational complexity of $O((h+b)n)$, with *h* the number of inanimate objects, *b* the average number of people in adjacent grid cells, and *n* the number of people. On a modern PC, the model runs faster than real time until there are several hundred closely packed people and a few dozen inanimate objects. Thus checking one exit configuration for acceptability amounts to at most a few minutes of computational effort.

Exit Selection

People select exits based on three factors—visibility, proximity, and congestion. This way, people prefer exits they can see, nearby exits, and less-crowded exits.

If an exit cannot be seen (that is, a wall lies between the person and the exit) it is ignored. The best remaining exit is selected according to the weighted strength value

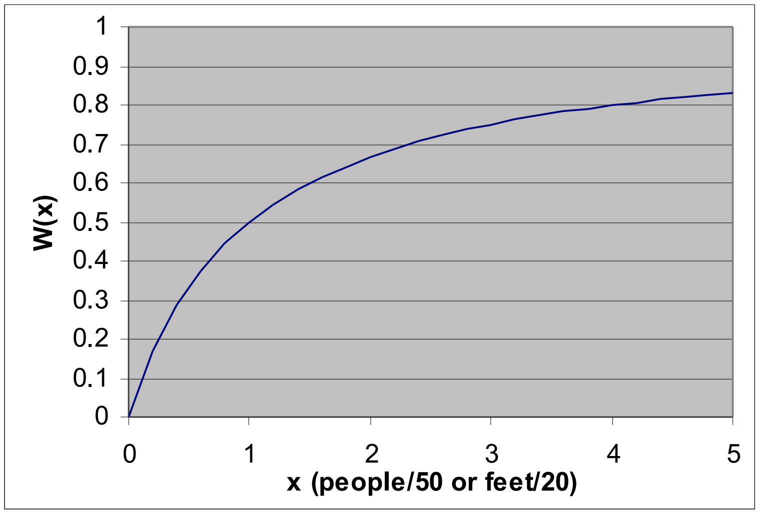
$$\text{strength} := S - W(\text{number of people}/50) - W(\text{distance}/20)$$

where *S* is the initial weight of the exit. Exit signs are simply exits with a low initial weight *S*. This way, if no actual exits are visible from some location, people will select the exit sign as their destination.

The weighting function

$$W(x) = \frac{x}{1+x}$$

has graph

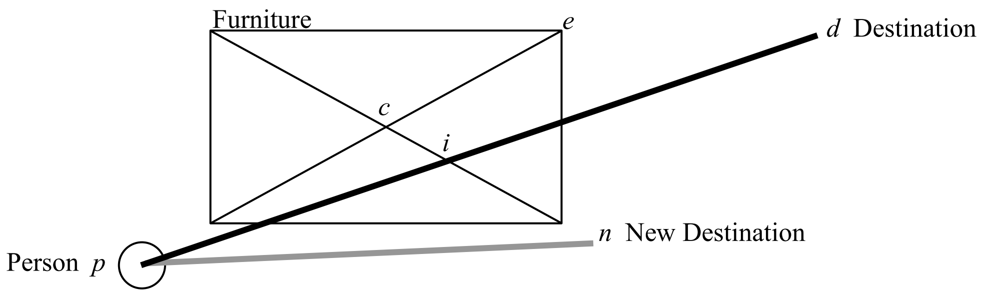


This weighting function makes very nearby or very uncrowded exits especially preferable. This prevents a person very near an exit from abandoning his choice for a marginally less crowded exit nearby.

Furniture Avoidance

Once a person has selected an exit, they check to see if any furniture lies on the straight-line path between them and the exit. If so, an intermediate destination is selected so the person's path will substantially avoid the furniture. Since people slide past furniture easily, it is not necessary to calculate a path that will completely avoid the furniture; an easy-to-compute approximation will suffice.

For rectangular furniture, this check consists of determining if either diagonal of the furniture intersects the path. If a diagonal intersects, the rectangular furniture lies in the person's path. The person's positive radius is neglected in this calculation for efficiency.



Pick the nearer of the two intersections, and label it *i*. The person must choose a new, halfway destination *n* so their path will not intersect the furniture. This new destination is chosen on the ray *ci* so the distance from *c* to *n* is half the rectangle's diagonal plus the person's radius.
That is

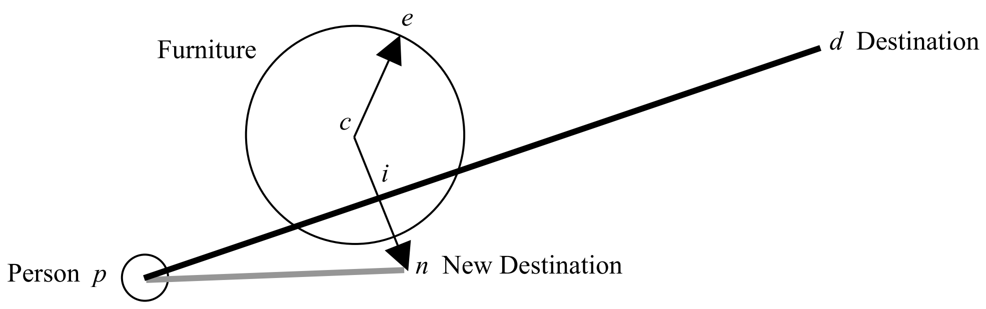
$$\overline{cn} = \overline{ce} + r$$

For round furniture, we find the point *i* on the path *pd* closest to the furniture's center *c*. Since *ci* must be perpendicular to *pd*, we can find *ci* by subtracting the projection of *pc* onto *pd* from *pc*.

That is

$$\overline{ci} = \overline{pc} - \overline{pd} * \left(\frac{\overline{pd} \bullet \overline{pc}}{\overline{pd} \bullet \overline{pd}} \right)$$

If *ci* is less than the sum of the person's and furniture's radii, then the person's path will intersect the furniture. In this case, the intermediate destination is calculated exactly as with rectangular furniture.

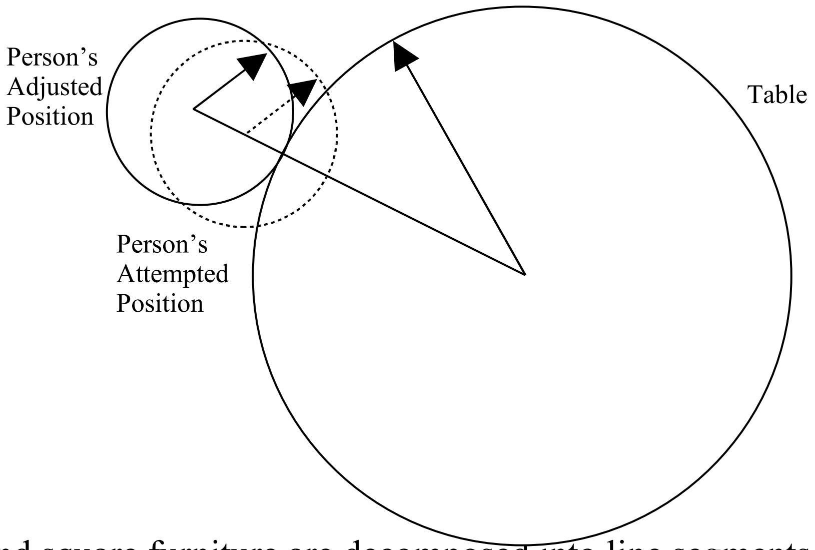


Hit Testing

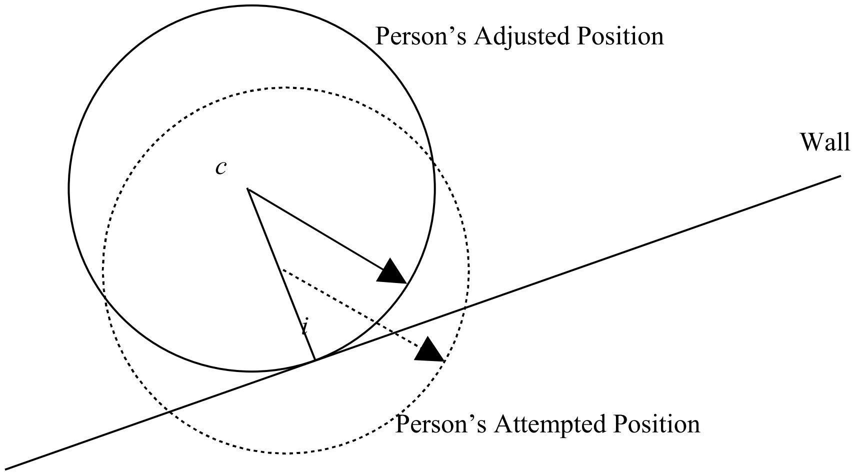
Our simulated people must be kept from moving through walls and furniture. In game programming, checking for these intersections is called hit testing.

Once a person has determined their desired next step, we check to see if that next step would make them intersect a wall or furniture. Thus the problem of keeping people out of the walls can be broken down into two parts: detecting an intersection with an inanimate object, and then repelling them out of the inanimate. Note that this process occurs before the person is allowed to take a step, so people never actually intersect walls or furniture.

It's easy to perform hit detection on round furniture, because two disks intersect iff the distance between their centers is less than the sum of their radii. To make the disks no longer intersect, we must move the person directly away from the center of the table. The distance the person moves is just the sum of the radii minus the distance between the centers.



Walls and square furniture are decomposed into line segments which are hit tested individually. To determine if a person intersects a line, we first find the point *i* nearest to the person and on the wall (this is done exactly as in round furniture avoidance). If the distance from the person's center *c* to *i* is less than the person's radius, the person is intersecting the line. As with round furniture, we need to move the person directly away from *i* until *ci* exceeds the person's radius.



Appendix—Tables

Particle Simulator: Walking Speed vs. Crowding

These numbers were derived from a particle-model simulation of an evacuation of the 110x120-foot gym. They show that as the number of people nearby increase, the average walking speed decreases.

Number of People Nearby	1	2	3	4	5	6
[ft^2/person]	25.00	12.50	8.33	6.25	5.00	4.17
Average Walking Speed [ft/sec]	3.10	2.75	2.26	1.63	0.64	0.19

Appendix—Walking Speed vs. Congestion

A key driving factor in egress time is the speed at which human beings walk. Fire safety texts estimate human walking speed at 4 feet per second for a person who is not in a crowd. But when personal space is reduced to less than 5-10 sq ft, then walking speed slows to 2.5 fps. [EGAN, p. 180]
A transportation study [BENZ, 1986] measured pedestrian travel speed as a function of square feet per pedestrian and the results supported the generally accepted estimates. Our model made use of the study results to simulate pedestrian movement limitations imposed by congestion. Below are raw observations and a regression provided by the study.

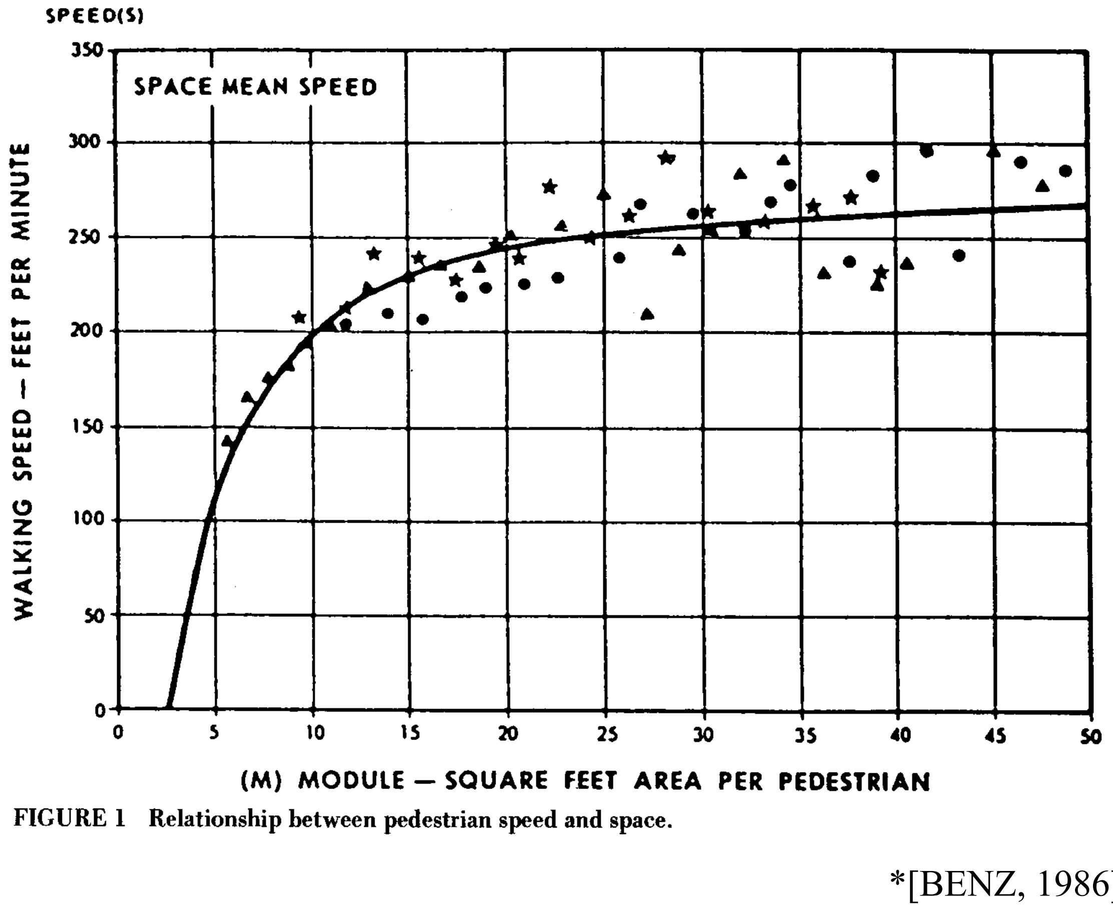


FIGURE 1 Relationship between pedestrian speed and space.

*[BENZ, 1986]

The point values and regression used in this paper was not listed, so we digitized the image, measured the points and obtained the regression y = 58.669 LN(x) + 58.678, which closely approximates the regression published in the paper. (R^2 for points taken from the digitized regression and our regression was 1.) This yields the following equation for pedestrian movement based on congestion:

S = pedestrian movement speed
A/P = square feet per pedestrian

$$S = 58.669 * LN(A/P) + 58.678$$

The graph-flow model uses this function to relate congestion to pedestrian movement rate. The particle simulator model merely verifies its assumptions to this curve. Because the paper by Benz contained few observations below 2.5 fps, P values less than 2.5 fps are clamped to 2.5 in order to maintain some flow of pedestrians in a congested situation.

Appendix—Pedestrian Size

The easiest way to keep track of the space occupied by a single person is to use a circle. The particle simulation model represents people as circles.

A naive method for determining the radius of this circle is to measure the physical size of a person to be represented. By measuring the farthest projecting point on a human, the minimum radius of a circle that contains the person can be determined. Using this method, the most significant factor in determining circle radius is the shoulder width of an adult male, which is 20 to 30 inches including sway. [LSC A-5-5.1.4]

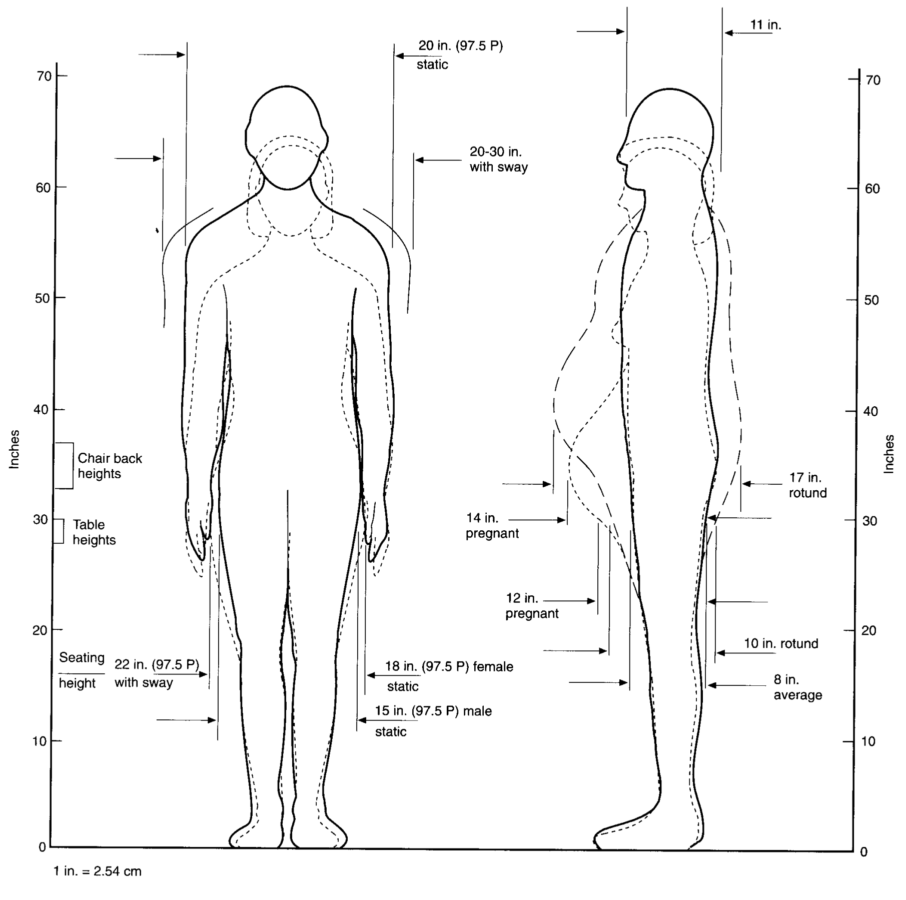


Figure A-5-3.4.1 Anthropometric data for adults. The male and female figures depicted here are average, 50th percentile, size. Some dimensions apply to very large, 97.5 percentile, adults (noted as 97.5 P).

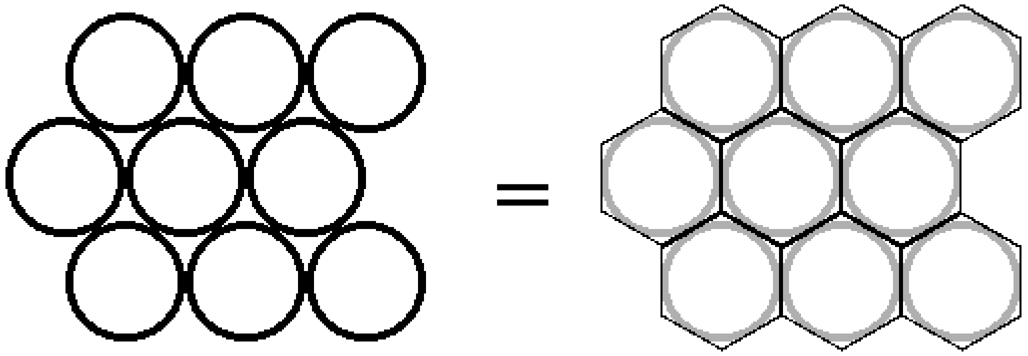
* LSC p. 101-257

This yields a radius of 1.25 feet to 1.4166 feet, including allowance for rotund individuals. This yields a simple area of 4.9 ft^2 to 6.3 ft^2. However, the USC allows for waiting areas (lines and other directed traffic areas, such as subways) to have one person per 3 ft^2. [Life Safety 101-256, Table A-5-3.1.2] If people are represented as disks with a radius equal to the most prominent projection of the person, it would not be possible to model bench seating or waiting lines, both of which are common situations in the large areas modeled by this paper. Clearly, allocating an area of 5 or 6 ft^2 per person is not an appropriate measure of personal space in crowded situations.

Another approach to finding circle radius is to specify the amount of space consumed by a single person in ft^2 and then set the radius of a person to reflect that standard area. In this case, we chose to set the amount of space consumed by an average person to 3 ft^2, which requires a radius of .977 feet. This approach also proved to be too naive to accurately model human density.

The tightest packing of disks in the plane is in hexagonal formation. Even in this configuration, there is a significant amount of free space left around the circles. Thus the circles do not pack to one per 3 ft^2 as desired. To find a

Hexagonal Packing of Circles



final size for circles, we found the radius of a circle inscribed in a hexagon of 3 ft^2 area.

The area of a hexagon circumscribed about a circle of radius R is $\frac{6R^2}{\sqrt{3}}$. So a circle of radius .9306 can be inscribed inside a hexagon with 3 ft^2 of area. This is the size of a circle representing a standard individual in the particle model.

Appendix: Graph Flow Model Source Code

// grideval.cpp: runs the Graph Flow Model

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define LEN 255
int GRIDSIZE;

#define NORTH 0
#define SOUTH 1
#define EAST 2
#define WEST 3

#define CLEANUP if(!ustdin) fclose(in); if(!ustdout) fclose(out)
#define xtoi(x) (int) ( (x) / GRIDSIZE - .5 )
#define ytoj(y) (int) ( ( width - (y) ) / GRIDSIZE - .5 )

extern int errno;

const int nodes_per_line = 5;     // number of nodes to print per line in the
display

class node {
 public:
// stuff for node building
  int indx;
  float pop;      // Number of people in a node
  float a;        // area of the node, square feet
  int exit;       // 0 = not an exit 1 = exit, infinite IF
  node *d[4];     // Destinations of the node (NULL by default)
  node *s[4];     // Nodes having this node as a destination (NULL by default)
  float w[4];     // some expression of bandwidth to the nodes in d
  // Stuff only the model uses
  int iffound;    // in flow found for this node
  int isdone;
  int fffound[4]; // final flow found for this direction
  float ff[4];    // final flow, in people
};

const int MAX_NODES = 100; // maximum number of nodes
const double movement_base = 58.678;    // feet per minute
const double movement_modifier = 58.669;       // feet per minute
const double min_movement = 2.5; // feet per second
float bandwidth; // bandwidth of edges, persons / foot
float exitwidth; // bandwidth of edges leading to exit, persons / foot
float fill_const = 1;            // fill rate constant, persons/second
const float terminal = .3333;    // terminal capacity, persons/square foot
const float large_number = 9999; // input capacity of an exit
void movepeople(node * n); // moves people, modifies final flow if necessary
float S(float P, float A); // area movement, ft/sec
float FR(node * n, float t);      // fill rate, persons
float OF(node * i, node * j, float t);  // desired output flow from i to j
float IF(node * i, float t);      // maximum input flow, in people
float FF(node * i, node * j, float t);  // final flow i to j, in people
```

```
void calctree(node * n, float t);// recursion function for an entire graph

int numnodes;
int nodesperline;

void init_nodes();         // initialize all nodes before a step
void init_print(float t);  //print initial info to the report file
void print_nodes(float t); // display current node populations to the screen

node nodes[MAX_NODES];
float speed[MAX_NODES];
float spop[MAX_NODES];
float sof[MAX_NODES];
float sff[MAX_NODES];
ofstream greport;

struct vertex {
 int north;
 int south;
 int east;
 int west;
 float pop;
};

struct vertex *makegraph(int, char **, int *);
void check(int);

int main(int argc, char **argv) {
 int i, j, k, ii, jj, vcount, done;

// cout << "Enter grid size: "; cin >> GRIDSIZE;
 GRIDSIZE = 5;

 struct vertex *ar;

 ar = makegraph(argc, argv, &vcount);
 for(i =0;i<vcount;i++)
 for(j =0;j<4;j++) {
  nodes[i].d[j] = NULL;
  nodes[i].s[j] = NULL;
 }
 for (i = 0; i < vcount; i++) {
  nodes[i].indx = i;
  nodes[i].pop = (float) ar[i].pop;
  nodes[i].a = (float) GRIDSIZE *GRIDSIZE;
  j = 0;
  if(ar[i].north != -1) {
   nodes[i].d[NORTH] = &nodes[ar[i].north];
   nodes[ar[i].north].s[NORTH] = &nodes[i];
  }
  if(ar[i].south != -1) {
   nodes[i].d[SOUTH] = &nodes[ar[i].south];
   nodes[ar[i].south].s[SOUTH] = &nodes[i];
  }
  if(ar[i].east != -1) {
   nodes[i].d[EAST] = &nodes[ar[i].east];
   nodes[ar[i].east].s[EAST]= &nodes[i];
  }
  if(ar[i].west != -1) {
   nodes[i].d[WEST] = &nodes[ar[i].west];
   nodes[ar[i].west].s[WEST] =&nodes[i];
  }
  nodes[i].exit = 0;
 }
 nodes[vcount - 1].exit = 1;

  numnodes = vcount;
cout << "free" << endl;
cout << "numn: " << numnodes << endl;
for(ii = 0; ii < numnodes; ii++) {
 cout << nodes[ii].indx << " d ";
 for(jj = 0; jj < 4; jj++) {
  if(nodes[ii].d[jj] != NULL) {
   cout << nodes[ii].d[jj]->indx;
  }
 }
 cout << " s ";
 for(jj = 0; jj < 4; jj++) {
  if(nodes[ii].s[jj] != NULL) {
   cout << nodes[ii].s[jj]->indx;
  }
 }
 cout << " a " << nodes[ii].a;
 cout << " e " << nodes[ii].exit;
 cout << endl;
} //getchar();
/* for(ii = 0; ii < numnodes; ii++) {
  if(nodes[ii].exit != 1) {
   cout << ii << " Enter population: "; cin >> nodes[ii].pop;
  }
 }*/
 free(ar);
 float t = 1;
 int nt;
// cout << "Enter node to node bandwidth: ";
// cin >> bandwidth;
 bandwidth = .541;
// cout << "Enter node to exit bandwidth: ";
// cin >> exitwidth;
 exitwidth = .325;
// cout << "Enter fill rate constant: ";
// cin >> fill_const;
 fill_const = 4.33;
// cout << "Enter time step: ";
// cin >> t;
 t = 0.1;
 char name[40];
 cout << "Enter report file name: "; cin >> name;
 greport.open(name);
// cout << "Enter number of time steps to execute: ";
// cin >> nt;
// bandwidth = .5;
// nodesperline = 2;
// numnodes = 5;
 init_print(t);
 ii = 0;
 done = 0;
 while(done == 0) {
  init_nodes();
  calctree(&nodes[numnodes-1], t);
  print_nodes((ii + 1) * t);
  done = 1;
  for(jj = 0; jj < numnodes; jj++) {
   if(nodes[jj].exit == 0) {
    if(nodes[jj].pop > 0) done = 0;
   }
  }
  ii++;
  cout << t * ii << endl;
//  getchar();
 }
 greport.close();
 cout << "done";
}

void init_print(float t) { //print initial info to the report file
 int ii;
 greport << "GRID SIZE" << "\t" << GRIDSIZE << endl;
 greport << "node bandwidth" << "\t" << bandwidth << endl;
 greport << "exit bandwidth" << "\t" << exitwidth << endl;
 greport << "fill rate constant" << "\t" << fill_const << endl;
 greport << "time step" << "\t" << t << endl;
 greport << "time\tavg speed";
 for(ii = 0; ii < numnodes; ii++) {
  greport << "\tP" << ii << "old\tP" << ii << "new";
  if(nodes[ii].exit == 1) greport << "ex";
 }
 for(ii = 0; ii < numnodes; ii++) {
  greport << "\tAct.Speed" << ii;
 }
 greport << endl;
}

void print_nodes(float t) { // display current node populations to the screen
 int nump = 0;
/* cout << "Time: " << t << endl;
 for (int ii = 0; ii < numnodes; ii++) {
  nump++;
  cout << nodes[ii].pop << "\t";
  if (nump % nodesperline == 0) cout << endl;
 }*/
 float avespeed = 0;
 float temp = 0;
 for (int ii = 0; ii < numnodes; ii++) {
  if (nodes[ii].exit != 1) {
   if (sof[ii] > 0) {
    temp += (spop[ii] * speed[ii] * sff[ii] / sof[ii]);
    avespeed += spop[ii];
   }
  }
 }
 if (avespeed > 0) avespeed = temp / avespeed;
 greport << t << "\t" << avespeed;
 for (int ii = 0; ii < numnodes; ii++) {
  greport << "\t" << spop[ii] << "\t" << nodes[ii].pop;
 }
 for (int ii = 0; ii < numnodes; ii++) {
  greport << "\t" << speed[ii];
 }
 greport << endl;
}

void init_nodes() { // initialize all nodes before a step
 for (int ii = 0; ii < numnodes; ii++) {
  nodes[ii].iffound = 0;   // in flow found for this node
  nodes[ii].isdone = 0;
  spop[ii] = nodes[ii].pop;
  sof[ii] = 0;
```

```
   sff[ii] = 0;
   for (int jj = 0; jj < 4; jj++) {
    nodes[ii].fffound[jj] = 0; // final flow found for this direction
    nodes[ii].ff[jj] = 0; // final flow, in people
   }
  }
 }

void movepeople(node * n) { // moves people, modifies final flow if necessary
// cout << "MP";
 int ii;
 float tff;
 tff = 0;
 for (ii = 0; ii < 4; ii++) {
  if (n->fffound[ii] == 1) {
   tff += n->ff[ii];
   if (n->d[ii] == NULL) {
    cout << "Moved people into NULL space. They froze and died. Bad programmer.
No biscuit. Try Again." << endl;
    exit(0);
   }
  }
 }
 if (tff > n->pop) {
  for (ii = 0; ii < 4; ii++) {
   if (n->fffound[ii] == 1) {
    n->ff[ii] = n->ff[ii] * n->pop / tff;
   }
  }
 }
 for (ii = 0; ii < 4; ii++) {
  if (n->ff[ii] > 0) {
   n->pop -= n->ff[ii];
   n->d[ii]->pop += n->ff[ii];
  }
 }
 if (n->pop < 0) {
  cout << "Warning, population below zero: " << n->pop << endl;
  n->pop = 0;
 }
 if (n->pop < -1) {
  cout << "population below -1, maybe somebody is actually that worthless, but
they're not in this simluation" << endl;
  exit(0);
 }
// cout << "mp";
}

void calctree(node * n, float t) { // recursive step calculation function call
on a node with IF known!
// cout << "c" << n->indx << " ";
 int ii, jj;
 int ready;
 n->isdone = 1;
 for (ii = 0; ii < 4; ii++) {
  if (n->s[ii] != NULL) {
   n->s[ii]->fffound[ii] = 1;
   n->s[ii]->ff[ii] = FF(n->s[ii], n, t);
//   cout << "n";
   if(n->s[ii]->ff[ii] < 0) {
//    cout << "fixing ff";
    n->s[ii]->ff[ii] = 0;
   }
   }
  }
// cout << "1";
 for (ii = 0; ii < 4; ii++) {
  if (n->s[ii] != NULL) {
   ready = 1;
   for (jj = 0; jj < 4; jj++) {
    if (n->s[ii]->d[jj] != NULL) {
     if (n->s[ii]->fffound[jj] != 1) {
      ready = 0;
     }
    }
   }
   if (ready == 1) {
    n->s[ii]->iffound = 1;
    movepeople(n->s[ii]);
   }
  }
 }
// cout << "2";
 for (ii = 0; ii < 4; ii++) {
  if ((n->s[ii] != NULL)) {
   if ((n->s[ii]->iffound == 1) && (n->s[ii]->isdone != 1)) {
    calctree(n->s[ii], t);
   }
  }
 }
// cout << "u";
}

float S(float P, float A) { // area movement, ft/sec
// cout << "S";
 float movement;
 if (P <= 0) return 0;
// cout << "SC";
 float spp = A / P;
 if (spp > 1) {
  movement = movement_modifier / 60.0 * log(spp);
 } else {
  movement = 0;
 }
 movement += movement_modifier / 60.0;
 if (movement < min_movement) movement = min_movement;
// cout << "SO";
 return movement;
}

float FR(node *n, float t) { // fill rate, persons
// cout << "FR";
 float fr = t * fill_const;
 fr = fr * (n->a * terminal - n->pop) / (n->a * terminal);
 return fr;
}

float OF(node *i, node *j, float t) { // desired output flow from i to j
// cout << "OF";
 float s = S(i->pop, i->a);
 float of;
 if(j->exit == 1) {
  of = t * s * exitwidth;
 } else {
  of = t * s * bandwidth;
 }
 speed[i->indx] = s;
// cout << "of";
 return of;
}

float IF(node *i, float t) { // maximum input flow, in people
// cout << "IF";
 float flow = FR(i, t);
 if (i->exit == 1) {
  return large_number;
 } else {
  for (int ii = 0; ii < 4; ii++) {
   if (i->fffound[ii] == 1) {
    flow += i->ff[ii];
   } else if (i->d[ii] != NULL) {
    cout << "calculating if for a node that hasn't had all it's FF's yet. And 
now I die." << endl;
    exit(0);
   }
  }
  return flow;
 }
}

float FF(node *i, node *j, float t) { // final flow i to j, in people
// cout << "FF";
 float ff;
 float tof;
// cout << 
 float of = OF(i, j, t);
 float iff = IF(j, t);
 sof[i->indx] += of;

 tof = 0;
 for (int ii = 0; ii < 4; ii++) {
  if (j->s[ii] != NULL) {
   tof += OF(j->s[ii], j, t);
  }
 }
// cout << "ff";
 if (j->exit == 1) {
  ff = of;
//  speed[j->index] = 0; // Maybe use this for error checking?
  return ff;
 } else {
  if (iff > tof) {
   ff = of;
   return ff;
  } else {
   if (tof > 0) {
    ff = of * of / tof;
   } else {
    ff = of;
   }
   sff[i->indx] += ff;
   return ff;
  }
 }
}

struct vertex *makegraph(int argc, char **argv, int *count)
{
    FILE *in, *out;
    int i, j, k, l, m, n, dex, vcount, length, width, startx, starty,
     stopx, stopy, ustdin = 0, ustdout = 0;
    char line[LEN];
    struct vertex *ar;
    float pop;

    if (argc > 1) {
       if (strcmp(argv[1], "-") == 0) {
#ifdef DEBUG
          fprintf(stderr, "input from stdin\n");
#endif
          in = stdin;
          ustdin = 1;
       } else {
#ifdef DEBUG
          fprintf(stderr, "input from %s\n", argv[1]);
#endif
          in = fopen(argv[1], "r");
       }
    } else {
#ifdef DEBUG
       fprintf(stderr, "input from stdin\n");
#endif
       in = stdin;
       ustdin = 1;
    }
    if (!in) {
       fprintf(stderr, "error opening file %s: %s\n",
              argv[1], strerror(errno));
       exit(1);
    }
    if (argc == 3) {
#ifdef DEBUG
       fprintf(stderr, "output to %s\n", argv[2]);
#endif
       out = fopen(argv[2], "w");
    } else {
#ifdef DEBUG
       fprintf(stderr, "output to stdout\n");
#endif
       out = stdout;
       ustdout = 1;
    }

    if (!out) {
       fprintf(stderr, "error opening file %s: %s\n",
              argv[2], strerror(errno));
       fclose(in);
       exit(1);
    }
    printf("x y Enter Room Dimensions.\n");
    if (NULL == fgets(line, LEN, in)) {
       fprintf(stderr, "error reading from: %s\n", strerror(errno));
       CLEANUP;
       exit(1);
    }
    line[strlen(line) - 1] = '\0';
#ifdef DEBUG
    fprintf(stderr, "read line %s\n", line);
#endif
    i = sscanf(line, "%d %d", &length, &width);
#ifdef DEBUG
    fprintf(stderr, "length %d width %d\n", length, width);
```

```
#endif
    if (i != 2) {
       fprintf(stderr, "invalid length, width input\n");
       CLEANUP;
       exit(1);
    }
    check(length);
    check(width);

    printf("x y x y Enter endpoints of the exit\n");
    if (NULL == fgets(line, LEN, in)) {
       fprintf(stderr, "error reading from: %s\n", strerror(errno));
       CLEANUP;
       exit(1);
    }
    line[strlen(line) - 1] = '\0';
#ifdef DEBUG
    fprintf(stderr, "read line %s\n", line);
#endif
    i = sscanf(line, "%d%d%d%d", &startx, &starty, &stopx, &stopy);
#ifdef DEBUG
    fprintf(stderr, "startx %d starty %d\nstopx %d stopy %d\n",
           startx, starty, stopx, stopy);
#endif
    if (i != 4) {
       fprintf(stderr, "invalid startx, starty, stopx, stopy input\n");
       CLEANUP;
       exit(1);
    }
    check(startx);
    check(starty);
    check(stopx);
    check(stopy);
    printf("Enter population per node\n");
    if (fgets(line, LEN, in) == NULL) {
       fprintf(stderr, "error reading from: %s\n", strerror(errno));
       CLEANUP;
       exit(1);
    }
    i = sscanf(line, "%f", &pop);
    if (i != 1) {
       fprintf(stderr, "invalid pop input\n");
       CLEANUP;
       exit(1);
    }
    if (pop <= 0) {
       fprintf(stderr, "invalid pop value %d\n");
       CLEANUP;
       exit(1);
    }
    vcount = length * width / (GRIDSIZE * GRIDSIZE) + 1;
#ifdef DEBUG
    printf("vertex count %d\n", vcount);
#endif
    ar = (struct vertex *) malloc(vcount * sizeof(struct vertex));
    m = xtoi(length - .5 * GRIDSIZE);
    n = ytoj(2.5);
#ifdef DEBUG
    printf("m %d n %d\n", m, n);
#endif

    for (i = 0; i < vcount; i++) {
       ar[i].north = ar[i].south = ar[i].east = ar[i].west = (-1);
```

```
        ar[i].pop = pop;
    }

    ar[vcount - 1].pop = 0;

    k = xtoi(startx == 0 ? .5 * GRIDSIZE : length - .5 * GRIDSIZE);
    l = ytoj(.5 * (starty + stopy));

    for (i = 0; i <= m; i++)
        for (j = 0; j <= n; j++) {
            dex = j * length / GRIDSIZE + i;
            if (i < k) {
                ar[dex].east = dex + 1;
    }
            else if (i > k) {
                ar[dex].west = dex - 1;
         }
            if (j < l) {
                ar[dex].south = (j + 1) * length / GRIDSIZE + i;
        }
            else if (j > l) {
                ar[dex].north = (j - 1) * length / GRIDSIZE + i;
        }
         }

    if (startx == starty) {
        if (startx == length)
            ar[l * length / GRIDSIZE + k].east = vcount - 1;
        else
            ar[l * length / GRIDSIZE + k].west = vcount - 1;
    } else {
        if (starty == width)
            ar[l * length / GRIDSIZE + k].north = vcount - 1;
        else
            ar[l * length / GRIDSIZE + k].south = vcount - 1;
    }
    for (i = 0; i < vcount - 1; i++) {
    }
    CLEANUP;

    *count = vcount;
    return ar;
}

void check(int n)
{
    if (n < 0 || n % GRIDSIZE) {
        fprintf(stderr, "invalid input %d\n", n);
        exit(1);
    }
}
```

Appendix: Ballroom Point Simulator Input File

```
;             x1         y1        x2       y2
;North Wall
boundary      6          6         65       6
;East Wall
boundary      65         6         65       7
boundary      65         7         66       7
exit          66         7         66       10
boundary      66         10        65       10
boundary      65         10        65       62
exitsign      63         8.5       -1.1
;South Wall
boundary      65         62        44       62
boundary      44         62        44       65
exit          44         65        41       65
boundary      41         62        41       65
exit          41         65        38       65
boundary      38         62        38       65
exit          38         65        35       65
boundary      35         62        35       65
exit          35         65        32       65
boundary      32         62        32       65
exit          32         65        29       65
boundary      29         62        29       65
exit          29         65        26       65
boundary      26         62        26       65
exit          26         65        23       65
boundary      23         62        23       65
exit          23         65        20       65
boundary      20         62        20       65
exit          20         65        14       65
boundary      17         65        17       62
exit          17         65        14       65
boundary      14         65        14       62
boundary      14         62        6        62
exitsign      39         60        -1
;West Wall
boundary      6          62        6        6
;             x          y         radius
;Furniture – a set of tables
fcircle       13         13        3
fcircle       21         21        3
fcircle       29         13        3
fcircle       37         21        3
fcircle       45         13        3
fcircle       53         21        3
fcircle       13         29        3
fcircle       21         37        3
fcircle       29         29        3
fcircle       37         37        3
fcircle       45         29        3
fcircle       53         37        3
fcircle       13         45        3
fcircle       21         53        3
fcircle       29         45        3
fcircle      37         53        3
fcircle      45         45        3
fcircle      53         53        3
;People—they sit in circles around the tables
person       9          13        0.9306
person       17         13        0.9306
person       13         9         0.9306
person       13         17        0.9306
person       25         13        0.9306
person       33         13        0.9306
person       13         25        0.9306
person       13         33        0.9306
person       29         9         0.9306
person       9          29        0.9306
person       29         17        0.9306
person       17         29        0.9306
person       21         17        0.9306
person       21         25        0.9306
person       17         21        0.9306
person       25         21        0.9306
person       37         17        0.9306
person       37         25        0.9306
person       33         21        0.9306
person       41         21        0.9306
person       29         25        0.9306
person       29         33        0.9306
person       25         29        0.9306
person       33         29        0.9306
person       53         49        0.9306
person       53         57        0.9306
person       49         53        0.9306
person       57         53        0.9306
person       45         41        0.9306
person       45         49        0.9306
person       41         45        0.9306
person       49         45        0.9306
person       37         49        0.9306
...
person       53         25        0.9306
person       49         21        0.9306
person       57         21        0.9306
```

Appendix—Particle Simulator Code

//global.h: includes all objects related to the simulator itself

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "resources/menu_ids.h"

class disaster;//Forward-declare this, so we can use it later
class person;//Forward-declare this, too, so we can use it later
class gridCell;

//******************** Utility Classes/ Typedefs ********************
typedef double coord;//A coordinate in simulator space
typedef double real;//A real number (might use "float" for this)

//Point: a 2D coordinate pair.  Can also be thought of as a vector
class point {
public:
   coord x,y;//Location of the point
   point() {x=y=0;}
   point(coord Nx,coord Ny)   {x=Nx;y=Ny;}
   //(Default copy constructor)
   void add(point &p)  {x+=p.x;y+=p.y;}
   void sub(point &p)  {x-=p.x;y-=p.y;}
   void scale(real r)  {x*=r;y*=r;}
   void div(real r) {scale(1.0/r);}
   real dot(point &p)  {return p.x*x+p.y*y;}
   real mag(void) {return sqrt(dot(*this));}
   void normalize(void) {scale(1.0/mag());}
   friend real ptDist(point &a,point &b)
      {point c=a;c.sub(b);return c.mag();}
   friend real moreDist(point &a,point &b)//Is at least ptDist
      {real dx=a.x-b.x,dy=a.y-b.y;
      if (dx<0)dx=-dx;if (dy<0)dy=-dy;return dx+dy;}
   friend point blend(point a,real strengthA,point b)
   {point ret=a,bScale=b;ret.scale(strengthA);
   bScale.scale(1.0-strengthA);ret.add(bScale);return ret;}
};
//Point utility routines
bool intersects(point a1,point a2,point b1,point b2,point &intersection);
coord lineDist(person &p,point t1,point t2,coord width,point &hitPt);
//Return the distance between the given person and the line t1-t2,
// with given line width.

//Win32 Pens used for coloring various objects
extern HPEN blackPen,redPen,bluePen,greenPen;

//Random number utilities
real realRand(void);//Return a random number on [0,1)
point pointRand(real max);//Return random point on
                    //[-max/2,max/2)x[-max/2,max/2)
bool probability(int percent);//Return true every "probability" out of 100 tries


//Screen: Map from simulator "coord"s to screen locations; draw to screen
class screen {
public:
    HDC dc;
    double scaleX,offX,scaleY,offY;
    screen(HDC Ndc,
            double NscaleX,double NoffX,
            double NscaleY,double NoffY)
        {dc=Ndc;
        scaleX=NscaleX;offX=NoffX;
        scaleY=NscaleY;offY=NoffY;}
    void printFloat(point p,double f);
    void printStr(point p,const char *str);
    void map(point p,int &x,int &y)
{x=(int)(p.x*scaleX+offX);y=(int)(p.y*scaleY+offY);}
};


//****************************************************************************
*********
//**************** Simulation Classes ****************************

//Drawable: Abstract Superclass of all objects that can be drawn to a window
class drawable {
public:
        virtual void draw(screen &s)=0;
        virtual HPEN pen(void) {return blackPen;}
};

//Hit: These are things people can trip over/run into (including other people)
class hit: public drawable {
public:
    virtual coord hits(person &p,point &hitPt)=0;
    //Return nearest hit point, and the distance
    //between person's edge and this object.
    //(if negative, person is penetrating object)
    virtual bool inWay(person *you,point dest,coord &dist) {return false;}
    //Are we in the way between you and dest?
    //If so, how far away are we from you?
    virtual point reRoute(person *you,point dest) {return dest;}
    //Give me a better route between you and dest,
    //so I don't hit you.
};

//Wall: One straight line segment, comprising a barrier to travel and sight.
class wall: public hit {
public:
    point t1,t2;//Endpoints of wall
    real radius;//Size of "Repulsion Field" around wall
    virtual void init(point Nt1,point Nt2) {t1=Nt1;t2=Nt2;radius=0.0;}
    virtual void draw(screen &s);
    virtual coord hits(person &p,point &hitPt);//Return nearest hit point, and
distance to there
};

//Anexit: An exit is a wall that eats people
class anexit: public wall {
public:
    point myCenter;
    int nFolks;//Number of people surrounding exit
    double congestion;
    double myStrength;
    coord diameter;//Search diameter for congestion
    virtual bool iamBetter(anexit *other,point vantage);//Is this exit better
than other?
```

```
   virtual void calcCongestion(disaster &d);//Find out how many people are
nearby
   virtual double strength(point vantage);
   virtual void init(point Nt1,point Nt2,double Nstrength,double Ndiameter);
   virtual void draw(screen &s);
   virtual HPEN pen(void) {return greenPen;}
   bool canLeave(person &p);//Can person p leave via this exit now?
   point center(void) {return myCenter;}//Return the coordinates of the exit's
center
};

//SquareFurniture: A piece of square furniture, like a desk
class squareFurniture: public hit {
public:
   point v[5];//Vertices (v[0]==v[4])
   point c;//Center
   virtual void init(coord len,coord ht,coord x,coord y,coord theta);
   virtual void draw(screen &s);
   virtual coord hits(person &p,point &hitPt);
   virtual bool inWay(person *you,point dest,coord &dist);
   virtual point reRoute(person *you,point dest);
};

//RoundFurniture: A piece of round furniture
class roundFurniture: public hit {
public:
   point c;//Center of furniture
   coord radius;
   virtual void init(point Nc,coord Nradius) {c=Nc;radius=Nradius;}
   virtual void draw(screen &s);
   virtual coord hits(person &p,point &hitPt);
   virtual bool inWay(person *you,point dest,coord &dist);
   virtual point reRoute(person *you,point dest);
};

//Person: A person is basically a piece of round furniture that moves
//This way, the hit detection code only need be written once.
class person: public roundFurniture {
public:
   typedef enum {
      state_walking=0,//Walking toward exit
      state_noexit=1,//Can't see an exit
      state_atexit=2//Am sitting in exit
   } stateType;
   gridCell *cell;
   coord totalProgress,lastProgress;
   anexit *destExit;
   int nStuck;point lastStuck;
   point lastDest;
   point walkDir;
   person() {destExit=NULL;cell=NULL;totalProgress=lastProgress=0;}
   virtual coord walkSpeed(disaster &d);//Return walking speed during given
disaster
   point getDestination(disaster &d);//Find nearest exit from given disaster
   bool repelledAt(hit *h,point newPos,point &repulsion);
   bool hitTest(hit *h);//Do we run into h walking in this direction?
   virtual HPEN pen(void);

   void advance_begin(disaster &d);//Flee the given disaster
   bool hitPeople(person *p);//Do we run into p walking in this direction?
   void advance_end(disaster &d);//Done fleeing
   bool at_exit(disaster &d);//Out of the given exit
   virtual void draw(screen &s);//Draw self on screen
```

```
};

class gridBag;//A set of grid cells, forward-declared for efficiency

//Disaster: contains the state of the evacuation of a single room.
#define max_objects 100 //The
#define max_people 5000 //The maximum # of occupants (for static alloc)
class disaster {
public:
   real dt;/*Time step, in seconds*/
   int time;/*Total number of frames so far*/
   gridBag *grid;
   //Lists of objects in the room
   wall **walls;int nwalls;
   anexit **exits;int nexits;
   hit **hits;int nhits;
   person **people;int npeople;

   void init(const char *fName);
   ~disaster();
   bool isVisible(point dest,point source);//Is dest visible from source?
   bool parse_input(const char *fName);//Add objects in given input file to
disaster
   void advance(void);//Move objects around by one simulation timestep
   void draw(screen &s);//Draw all objects (may be much slower than
"advance()")
   int totalSaved;//Total number of people that have exited room
   FILE *log,*log2;//Log file pointers
   void saveLog(void);//Write important data to log file
};

#include "app_main.h"

#include "grid.h"
```

//grid.h: contains grid-related definitions

```
//GridCell: a 5x5 foot patch of ground.
//Keeps track of which people are in this cell.
class gridCell {
protected:
   int cur;
   person **people;int npeople;
public:
   gridCell();
   ~gridCell();
   void init(int max);
   int num(void) {return npeople;}//Return number of people in list

   void add(person *p);//Add given person to list
   void clear(void){npeople=0;}//Set people-list to empty
   void reset(void){cur=0;}//Start listing people from people-list
   person *next(void);//Get next person in list
};


#define gridRes 5.0 /*Resolution of grid, in feet*/
class gridBag {
protected:
   int wid,ht;
   gridCell *cells;//Row-major list of grid cells, widxht.
   gridCell **curCells;//9-element list of nearby cells
   int curCell;//Current cell number
public:
   gridBag(int Nwid,int Nht,int max);
   ~gridBag();

   void clear(void);//Set all people-lists to empty
   coord gridArea(void);//Return area of a grid cell (sq. feet)
   gridCell &getCell(point p);//Return grid cell for given location

   gridCell &setCur(point p);//Sets the current cell to that around p.
   person *next(void);//Get the next person near the current cell.

   int nCenter(void) {return wid*ht+1;}//Return number of grid cells
   point center(int num);//Return center of grid cell num
};
```

//app_main.h:

```
class disaster;

class CApp
{
protected:
   char inFile[255];
   HWND window;
   disaster *d;
   double zoom;
   int runFor,draws;
   bool running;
   bool leaveTrails;
public:
   virtual void init(HWND Nwindow);
   virtual bool paint(HDC dc);
   virtual void menu(int menuId);
   virtual void close(void);
};
```

//std_main.h: Contains standard Win32 definitions

```
#ifndef WinCE
#define WinCE 0
#endif

extern const TCHAR g_szAppName[];
extern HINSTANCE g_hInst;

typedef struct {
   UINT Code;
   LRESULT (*Fxn)(HWND,UINT,WPARAM,LPARAM);
//Prototype: LRESULT func(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam) {
}  decodeUINT;

//These variables must be filled out by the client.
extern const decodeUINT messageArr[];
extern const int messageLen;

extern const char *winName;
extern const int winWid,winHt;
```

//disaster.cpp: Implements all disaster methods except parse_input

```
#include "global.h"

void disaster::init(const char *fName)
{
    srand(1);//Seed the random number generator, so we get the same sequence
    dt=0.20;//Timestep in seconds
    time=0;
    log=fopen("exit_log.txt","w");
    log2=fopen("progress_log.txt","w");
    totalSaved=0;

    //Create the grid
    grid=new gridBag(30,30,50);

    //Create lists of objects
    walls=new wall*[max_objects];nwalls=0;
    exits=new anexit*[max_objects];nexits=0;
    hits=new hit*[max_objects];nhits=0;
    people=new person*[max_people];npeople=0;

    if (!parse_input(fName)&&!parse_input("test.txt"))
    {//If we couldn't open any input files
        //Hardcode objects to fill the lists
        (walls[nwalls++]=new wall)->init(point(10,0),point(10,15));
        (walls[nwalls++]=new wall)->init(point(10,15),point(5,15));
        (exits[nexits++]=new anexit)->init(point(14,17),point(14,18),-1.0,20.0);
        (exits[nexits++]=new anexit)->init(point(5,15),point(5,20),1.0,20.0);
        (walls[nwalls++]=new wall)->init(point(5,20),point(10,20));
        (walls[nwalls++]=new wall)->init(point(10,20),point(10,100));
        //(exits[nexits++]=new anexit)->init(point(5,40),point(5,45),1.0,50.0);
        //((roundFurniture *)(hits[nhits++]=new roundFurniture))->
        //  init(point(40,35),6);
        ((squareFurniture *)(hits[nhits++]=new squareFurniture))->
           init(3,8,40,25,0);
        //Allocate a bunch of people
        int x,y;
        for (x=55;x<85;x+=4)
        for (y=20;y<45;y+=4)
           ((roundFurniture *)(people[npeople++]=new person))->
              init(point(x,y),1);
    }
}

disaster::~disaster()
{//Destructor
    if (log!=NULL) fclose(log);
    delete grid;
    int i;
    for (i=0;i<nwalls;i++) delete walls[i]; delete [] walls;
    for (i=0;i<nexits;i++) delete exits[i]; delete [] exits;
    for (i=0;i<nhits;i++) delete hits[i]; delete [] hits;
    for (i=0;i<npeople;i++) delete people[i]; delete [] people;
}

void disaster::advance(void)//Move objects around
{
    if (npeople>0)
       time++;//Increment the frame counter (time*dt=simulation time in seconds)

    int i,cell;
    //Clear each grid cell
    grid->clear();

    //Sift people into their grid cells
    for (i=0;i<npeople;i++)
    {
       person *p=people[i];
```

```
        gridCell *g=&(grid->getCell(p->c));
        g->add(p);
        p->cell=g;
    }

    //Update the exit congestion
    for (i=0;i<nexits;i++)
        exits[i]->calcCongestion(*this);

    //Perform human interactions: For each grid cell...
    for (cell=0;cell<grid->nCenter();cell++)
    {
        gridCell &center=grid->setCur(grid->center(cell));
        //Break if nobody's here
        if (center.num()==0) continue;

        //Make a list of people near the current cell
        person *nearPeople[200];int nNear=0;
        while (NULL!=(nearPeople[nNear]=grid->next()))
            nNear++;

        //For each person in this cell...
        center.reset();
        person *cur;//The person inside the current cell
        while (NULL!=(cur=center.next()))
        {
            cur->advance_begin(*this);
            //Hit-test them against everybody nearby
            for (i=0;i<nNear;i++)
                if (cur!=nearPeople[i])
                    cur->hitPeople(nearPeople[i]);
            cur->advance_end(*this);
        }
    }

    //Check to see if they're at an exit
    for (i=0;i<npeople;i++)
        if (people[i]->at_exit(*this))
        {//This person has reached an exit and is ready to leave
            totalSaved++;//Increment saved-counter
             delete people[i];
            people[i]=people[--npeople];
            i--;
        }

    if ((time%5==0)||(npeople==0))
        saveLog();//Log every second
}

bool disaster::isVisible(point dest,point source)//Is dest visible from source?
{
    point cross;
    int i;
    for (i=0;i<nwalls;i++)
        if (intersects(walls[i]->t1,walls[i]->t2,source,dest,cross))
            return false;//Line from source to dest hits wall
    return true;//If no wall obscures our vision, we can see dest.
}
void disaster::saveLog(void)
{//Write useful information to log files
    if (log==NULL) return;
    fprintf(log,"%.2f\t",time*dt);//Write time to log file
    fprintf(log,"%d\t",exits[0]->nFolks);//Write # of people around exit 0
    fprintf(log,"%d\t",totalSaved);//Write # of people saved

    fprintf(log,"\n");//Write end-of-line to log file
    fflush(log);

    if (log2==NULL) return;
    //Loop on the number of people in this cell
    int nPeople;
    for (nPeople=0;nPeople<10;nPeople++)
    {
        double ave=0;//Average amount of progress for these people
        int num=0;
        int i;
        for (i=0;i<npeople;i++)
            if (people[i]->cell->num()==nPeople)
            {
                ave+=people[i]->totalProgress-people[i]->lastProgress;
                people[i]->lastProgress=people[i]->totalProgress;
                num++;
            }
        fprintf(log2,"%d\t%.2f\t",num,ave/num);
    }
    fprintf(log2,"\n");
    fflush(log2);
}


void disaster::draw(screen &s)//Draw all objects (may be much slower than
"advance()")
{
    //Draw frame counter
    char timeStr[100];
    RECT r={3,3,1000,1000};
    sprintf(timeStr,"%d.%02d s",(int)(time*dt),(int)(time*dt*100)%100);
    ::DrawText(s.dc,timeStr,-1,&r,DT_NOCLIP);

    //Loop over each object type, calling its "draw" function
    int i;
    for (i=0;i<nwalls;i++) walls[i]->draw(s);
    for (i=0;i<nexits;i++) exits[i]->draw(s);
    for (i=0;i<nhits;i++) hits[i]->draw(s);
    for (i=0;i<npeople;i++) people[i]->draw(s);
}
```

<u>//exit.cpp: implements all Anexit class methods</u>

```
#include "global.h"
void anexit::init(point Nt1,point Nt2,double Nstrength,double Ndiameter)
{
    t1=Nt1;t2=Nt2;
    myStrength=Nstrength;diameter=Ndiameter;
    myCenter=t1;
    myCenter.add(t2);
    myCenter.scale(0.5);
    radius=0;
}
void anexit::calcCongestion(disaster &d)
{//Calculate the congestion near the exit
    int dx,dy;
    int dist=(int)(diameter/gridRes);
    nFolks=0;

    for (dy=-dist;dy<=dist;dy++)
    for (dx=-dist;dx<=dist;dx++)
    {
        point p=point(myCenter.x+dx*gridRes,myCenter.y+dy*gridRes);
        nFolks+=d.grid->getCell(p).num();
    }
    nFolks;
    double floatFolks=nFolks/50;
    congestion=(floatFolks/(1+floatFolks));
}
double anexit::strength(point vantage)
{//Return the strength of this exit, viewed from vantage.
    double dist=ptDist(vantage,myCenter)/20.0;
    return myStrength-5*dist/(1+dist)-5*congestion;
}

bool anexit::iamBetter(anexit *other,point vantage)//Is this exit better than
other?
{
    double disStrength=strength(vantage);
    double hisStrength=other->strength(vantage);
    return disStrength>hisStrength;
}

bool anexit::canLeave(person &p)//Can person p leave via this exit now?
{
    if (myStrength<0)
        return false;//This is just a warning sign, not an exit!
    point ignored;
    if (hits(p,ignored)<0)
        return true;//Person is penetrating exit-- let them leave
    else
        return false;//Person is not penetrating exit-- they stay.
}


void anexit::draw(screen &s)
{
    wall::draw(s);
    //s.printFloat(myCenter,congestion);
}
```

//grid.cpp: implements all grid-related classes

```
#include "global.h"

/***************** GridCell Implementation*****************/
gridCell::gridCell()
{
    people=NULL;cur=0;npeople=0;
}
gridCell::~gridCell()
{
    if (people!=NULL)
        delete [] people;
}
void gridCell::init(int max)
{
    people=new person*[max];
    npeople=0;
}

void gridCell::add(person *p)//Add given person to list
{people[npeople++]=p;}
```

```
person *gridCell::next(void)//Get next person in list
{
   if (cur<npeople)
      return people[cur++];
   else return NULL;
}

/******************* GridBag Implementation ************************/
gridBag::gridBag(int Nwid,int Nht,int max)
{
   wid=Nwid;ht=Nht;
   curCells=new gridCell*[9];
   cells=new gridCell[wid*ht+1];
   int x,y;
   for (y=0;y<ht;y++)
      for(x=0;x<wid;x++)
         cells[y*wid+x].init(max);
   cells[wid*ht].init(max);//Out-of-bounds cell
}
gridBag::~gridBag()
{
   delete [] cells;
   delete [] curCells;
}

void gridBag::clear(void)//Set all people-lists to empty
{
   int x,y;
   for (y=0;y<ht;y++)
      for(x=0;x<wid;x++)
         cells[y*wid+x].clear();
   cells[wid*ht].clear();//Out-of-bounds cell
}
coord gridBag::gridArea(void)//Return area of a grid cell (sq. feet)
{
   return gridRes*gridRes;
}
gridCell &gridBag::getCell(point p)//Return grid cell for given location
{
   if ((p.x<0)||(p.x>=gridRes*wid)||
      (p.y<0)||(p.y>=gridRes*ht))
      return cells[wid*ht];//Out-of-bounds!
   int x,y;
   x=(int)(p.x/gridRes);
   y=(int)(p.y/gridRes);
   return cells[wid*x+y];//In-bounds
}

gridCell &gridBag::setCur(point p)//Sets the current cell to that around p.
{
   int cellNo=0;
   int dx,dy;
   for (dy=-1;dy<=1;dy++)
   for (dx=-1;dx<=1;dx++)
   {
      point pNew(p.x+dx*gridRes,p.y+dy*gridRes);
      gridCell *g=&getCell(pNew);
      g->reset();
      curCells[cellNo++]=g;
   }
   curCell=0;
   return *curCells[4];//Return middle cell.
}
```

```
person *gridBag::next(void)//Get the next person near the current cell.
{
   person *p=NULL;
   do
   {
      if (curCell>=9)
         return NULL;
      p=curCells[curCell]->next();
      if (p!=NULL)
         return p;
      else
         curCell++;//Advance to next cell-- this one's empty
   } while (p==NULL);
   return NULL;//If we couldn't find one by now, none exists
}

point gridBag::center(int num)//Return center of grid cell num
{
   if (num==wid*ht)
      return point(-10,-10);//Out-of-bounds
   int x=num%wid,y=num/wid;
   return point(x*gridRes+gridRes/2,y*gridRes+gridRes/2);
}
```

//person.cpp: implements methods for person class

```
#include "global.h"

//A person is basically a piece of round furniture that moves
//This way, the hit detection code only need be written
//once.

HPEN person::pen(void)
{
   return redPen;
}
coord person::walkSpeed(disaster &d)
{
   return 4*d.dt;//People walk at 4.0 feet per second (always)
}

point person::getDestination(disaster &d)//Find nearest exit from given disaster
{
//If we were last stuck, flee the person we stuck to
   if (nStuck>0)
      return lastStuck;

//Check to see if we need a new exit
   if ((destExit!=NULL) //And we had an old destination
      &&((ptDist(c,lastDest)>4.0))//If we're still a ways from our old
destination
      && probability(98) //And this is most of the time
     )
         return lastDest;//Keep old target (don't re-target)

//Find the nearest visible exit
   destExit=NULL;
   int exitNo;
   for (exitNo=0;exitNo<d.nexits;exitNo++)
   {
      anexit *e=d.exits[exitNo];
      if (d.isVisible(e->center(),c))
      //This exit is visible
         if ((destExit==NULL)||(e->iamBetter(destExit,c)))
         //This exit is better than the current best
            destExit=e;
   }

   if (destExit==NULL) //We didn't find any exits!
   //Head in a random direction (wander)
   {
      point dest=c;
      dest.add(pointRand(10.0));
      return dest;
   }
   else//We found a good exit
   {
      point dest=destExit->center();//First guess: head towards nearest exit
   //Make sure no furniture is in the way of the exit:
      coord nearest=10000000;
      hit *inWay=NULL;
      int i;
      //Find nearest furniture blocking our path (from c to dest)
      for (i=0;i<d.nhits;i++)
      {
         coord thisDist;
         if (d.hits[i]->inWay(this,dest,thisDist))
         {
            if (thisDist<nearest)
            {
               nearest=thisDist;
               inWay=d.hits[i];
            }
         }
      }
      //Find a way around this furniture:
      if (inWay!=NULL)
         dest=inWay->reRoute(this,dest);
      return dest;
   }
}

void person::advance_begin(disaster &d)//Flee the given disaster
{
   lastDest=getDestination(d);

   nStuck=0;
//We want to walk towards our destination
   walkDir=lastDest;
   walkDir.sub(c);//Walkdir points from us to our destination
   walkDir.scale(walkSpeed(d)/walkDir.mag());//Scale to correct speed

   int i;
   for (i=0;i<d.nhits;i++) hitTest(d.hits[i]);
   for (i=0;i<d.nwalls;i++) hitTest(d.walls[i]);
}

bool person::repelledAt(hit *h,point newPos,point &repulsion)
//Return the repulsion vector h delivers to us, if we are at newPos
{
   point hitPt;
   coord hitDist;//Distance between us and object h
   point oldC=c;

   //Pretend we've walked in this direction
   c=newPos;
   //See if we hit anything
```

```
    hitDist=h->hits(*this,hitPt);
    //Stop pretending
    c=oldC;

    if (hitDist>0) return false;//No hit would happen.

    repulsion=newPos;//Points from hitPt to c+walkDir, which should be
            //away from the intersecting object's surface
    repulsion.sub(hitPt);
    repulsion.scale(-hitDist/repulsion.mag());
       //Make repulsion have length -hitDist

    return true;

}
bool person::hitTest(hit *h)
//Do we run into object h walking in this direction?
{
    point repulsion;
    point newPos=c;
    newPos.add(walkDir);//Imagine new position
    if (!repelledAt(h,newPos,repulsion)) return false;//No hit would happen.

//Otherwise, a hit would happen.  We need to scale the walk direction
//so we don't slam into this object
    repulsion.scale(1.0+realRand()*0.4);
    walkDir.add(repulsion);
    return true;
}
bool person::hitPeople(person *p)
//Do we run into person p walking in this direction?
{
    point newPos=c;
    newPos.add(walkDir);//Imagine new position

    //Find the distance between centers
    if (ptDist(newPos,p->c)-radius-p->radius>0)
       return false;//No hit would happen.

//Otherwise, we'd run into this person
    walkDir=point(0,0);//Stop
    //Note that we're stuck
    nStuck++;
    //Head in a random direction
    lastStuck=pointRand(1.0);
    lastStuck.add(c);
    return true;//We hit them.
}

void person::advance_end(disaster &d)//Done advancing
{
    if (destExit!=NULL)
    {//Check to see how much progress we've made towards this exit
       point a=destExit->center();a.sub(c);
       point b=walkDir;
       totalProgress+=a.dot(b)/a.mag();
    }
    //Now that we know the right direction in which to walk, walk there!
    c.add(walkDir);//Teleport in this direction (no inertia)
}

bool person::at_exit(disaster &d)//Can we leave yet?
{
    if (destExit==NULL)
        return false;//No exit, even!
    if (destExit->canLeave(*this))
        return true;//We're saved!
    return false;//We can't leave yet...
}


void person::draw(screen &s)//Draw self on screen
{//Just call superclass
    roundFurniture::draw(s);
}
```

//roundFurniture.cpp: implements roundFurniture object methods

```
#include "global.h"

void roundFurniture::draw(screen &s)
{
    int x,y;
    s.map(c,x,y);
    int del=(int)(radius*s.scaleX);
    HPEN oldPen=(HPEN)::SelectObject(s.dc,pen());
    ::Ellipse(s.dc,x-del,y-del,x+del,y+del);
    ::SelectObject(s.dc,oldPen);
}

coord roundFurniture::hits(person &p,point &hitPt)
{
    hitPt=c;
    //Intersecting disks is SO easy!
    //Distance between their edges is just
    //distance between centers minus radii
    return ptDist(p.c,c)-p.radius-radius;
}

bool roundFurniture::inWay(person *p,point dest,coord &dist)
//Does this object lie between the given person and their destination?
{
    point a=dest;a.sub(p->c);//a points from p to target
    point b=c;b.sub(p->c);//b points from p to our center
    point nearest=a;//Nearest point to our center on line p-dest
    nearest.scale(a.dot(b)/a.dot(a));
    if (a.dot(nearest)<0)
        return false;//The nearest point is behind us already
    dist=nearest.mag();
    nearest.add(p->c);
    coord r=ptDist(c,nearest);

    if (r-radius-p->radius<0)//If person's path intersects us,
        return true;//we're in the way
    else
        return false;
}
point roundFurniture::reRoute(person *p,point dest)
//How should this person walk to get around this object to their destination?
{
    point a=dest;a.sub(p->c);//a points from p to target
    point b=c;b.sub(p->c);//b points from p to our center
    point nearest=a;//Nearest point to our center on line p-dest
    nearest.scale(a.dot(b)/a.dot(a));
    nearest.add(p->c);
    nearest.sub(c);//Nearest now points from c to closest approach
    nearest.scale((radius+p->radius)/nearest.mag());
```

```
   nearest.add(c);
   return nearest;
}
```

//SquareFurniture: implements square furniture object methods

```
#include "global.h"

void squareFurniture::init(coord len,coord ht,coord x,coord y,coord theta)
{//Ignores theta (for now)
   len/=2;ht/=2;
   v[0].x=x-len;v[0].y=y-ht;
   v[1].x=x-len;v[1].y=y+ht;
   v[2].x=x+len;v[2].y=y+ht;
   v[3].x=x+len;v[3].y=y-ht;

   v[4]=v[0];
   c=v[0];
   c.add(v[2]);
   c.scale(0.5);
}

void squareFurniture::draw(screen &s)
{
   HPEN oldPen=(HPEN)::SelectObject(s.dc,pen());
   int x,y;
   s.map(v[0],x,y);
   ::MoveToEx(s.dc,x,y,NULL);
   int i;
   for (i=1;i<5;i++)
   {
      s.map(v[i],x,y);
      ::LineTo(s.dc,x,y);
   }
   ::SelectObject(s.dc,oldPen);
}

coord squareFurniture::hits(person &p,point &hitPt)
{
   real minDist=1000000;
   int i;
   for (i=1;i<5;i++)
   {
      point thisHit;
      real thisDist=lineDist(p,v[i-1],v[i],0.0,thisHit);
      if (thisDist<minDist)
      {
         minDist=thisDist;
         hitPt=thisHit;
      }
   }
   return minDist;
}

bool squareFurniture::inWay(person *p,point dest,coord &dist)
//Does this object lie between the given person and their destination?
{
   point cross;
   if (intersects(p->c,dest,v[0],v[2],cross)||
      intersects(p->c,dest,v[1],v[3],cross))
   {
      dist=ptDist(p->c,cross);
      return true;
   } else
```

```
      return false;
}
point squareFurniture::reRoute(person *p,point dest)
//How should this person walk to get around this object to their destination?
{
   point cross;//Intersection nearest to p->c
   point cross1,cross2;
   bool c1=intersects(p->c,dest,v[0],v[2],cross1);
   bool c2=intersects(p->c,dest,v[1],v[3],cross2);
   if (c1&&c2)
   {//If they both intersected, we need to find the closer intersection.
      if (ptDist(p->c,cross1)<ptDist(p->c,cross2))
         cross=cross1;//Cross1 is closer to person
      else
         cross=cross2;//Cross2 is closer
   } else if (c1)
      cross=cross1;
   else if (c2)
      cross=cross2;

   point a=cross;a.sub(c);//a points from our center to intersection
   //Make "a" point from center to just outside us
   a.scale((ptDist(v[0],c)+p->radius)/a.mag());
   //Translate "a" back to global coordinates
   a.add(c);
   return a;
}

//wall.cpp: implements wall class methods
#include "global.h"
void wall::draw(screen &s)
{
   int x1,y1,x2,y2;
   s.map(t1,x1,y1);
   s.map(t2,x2,y2);
   HPEN oldPen=(HPEN)::SelectObject(s.dc,pen());
   ::MoveToEx(s.dc,x1,y1,NULL);
   ::LineTo(s.dc,x2,y2);
   ::SelectObject(s.dc,oldPen);
}
coord wall::hits(person &p,point &hitPt)//Return nearest hit point, and
distance to there
{
   return lineDist(p,t1,t2,radius,hitPt);
}
```

//util.cpp: Implements utiltiy routines used throughout code

```
#include "global.h"

HPEN blackPen=::CreatePen(PS_SOLID,1,RGB(0x00,0x00,0x00));
HPEN redPen=::CreatePen(PS_SOLID,1,RGB(0xff,0x00,0x00));
HPEN greenPen=::CreatePen(PS_SOLID,1,RGB(0x00,0xff,0x00));
HPEN bluePen=::CreatePen(PS_SOLID,1,RGB(0x00,0x00,0xff));

real realRand(void)//Return a random number on [0,1)
{
   return ((real)(rand()&0x7fFF))/0x7fFF;
}

point pointRand(real max)//Return random point on [-max/2,max/2)x[-max/2,max/2)
{
   return point(realRand()*max-max/2,realRand()*max-max/2);
}
```

```
bool probability(int percent)//Return true probability out of 100 tries
{
   return ((rand()&0x7fff)%100)<percent;
}

void screen::printFloat(point p,double f)
{
   char buf[100];
   sprintf(buf,"%.3f",f);
   printStr(p,buf);
}
void screen::printStr(point p,const char *str)
{
   int x,y;
   map(p,x,y);
   RECT r={0,0,10000,10000};
   r.left=x;
   r.top=y;
   ::DrawText(dc,str,-1,&r,DT_NOCLIP);
}

coord lineDist(person &p,point t1,point t2,coord radius,point &hitPt)
{
   point a=t2;a.sub(t1);
   point b=p.c;b.sub(t1);
   coord c_len=a.dot(b)/a.mag();
   hitPt=a;
   hitPt.scale(c_len/a.mag());
   hitPt.add(t1);
   if (c_len<0.0)
      //Off t1 edge-- return distance
      return ptDist(p.c,t1)-p.radius-radius;
   else if (c_len>a.mag())
      //Off t2 edge-- return distance
      return ptDist(p.c,t2)-p.radius-radius;
   else//Bounce off middle
      return ptDist(p.c,hitPt)-p.radius-radius;
}


//Return if line segments [A1,A2] and [B1,B2] intersect, and where.
bool intersects(point A1,point A2,point B1,point B2,point &intersection)
{
   real m;//Slope y=mx+b
   real a1,b1=-1.0,c1;//Numbers a1*x+b1*y+c1=0 for line A1-A2
   if (A1.x==A2.x)
   {//Vertical line
      a1=1;b1=0;c1=-A1.x;}
   else
   {//Not vertical
      m=(A1.y-A2.y)/(A1.x-A2.x);
      a1=m;
      c1=A1.y-m*A1.x;
   }

   real a2,b2=-1.0,c2;//Numbers for line B1-B2
   if (B1.x==B2.x)
   {//Vertical line
      a2=1;b2=0;c2=-B1.x;}
   else
   {//Not vertical
      m=(B1.y-B2.y)/(B1.x-B2.x);
```

```
        a2=m;
        c2=B1.y-m*B1.x;
    }

    real det=a1*b2-a2*b1;
    if (det==0)
        return false;//Lines are parallel
    det=1.0/det;
    //Compute intersection location
    real ix=(b1*c2-b2*c1)*det;
    real iy=(c1*a2-c2*a1)*det;
    intersection.x=ix;
    intersection.y=iy;

    real tmp;
#define swap(a,b) {tmp=a;a=b;b=tmp;}
    //Re-Order vertices for easier bounds-testing
    if (A1.x>A2.x) swap(A1.x,A2.x);
    if (B1.x>B2.x) swap(B1.x,B2.x);
    if (A1.y>A2.y) swap(A1.y,A2.y);
    if (B1.y>B2.y) swap(B1.y,B2.y);

    //Check bounds on intersection location
    if ((A1.x<=ix)&&(ix<=A2.x)&&
        (A1.y<=iy)&&(iy<=A2.y)&&
        (B1.x<=ix)&&(ix<=B2.x)&&
        (B1.y<=iy)&&(iy<=B2.y))
        return true;
    return false;
}
```

//parser.cpp: Implements disaster::parse_input file

```
#include "global.h"


bool disaster::parse_input(const char *fName)
{
     double p1,p2,p3,p4,p5;
    int peoplePercent=100;
    double peopleRand=0.0;//Maximum random displacement of people

    if (fName==NULL||fName[0]==0) return false;
    FILE *out=fopen(fName,"r");
    if (out==NULL)
    {printf("Couldn't open input file '%s'!\n",fName);return false;}

    char buf[1024];
    int lineNo=1;
    while (NULL!=fgets(buf,1024,out))
    {
        char keyword[1024];
        strtok(buf,";");//Cut off comments
        keyword[0]=0;
        sscanf(buf,"%s",keyword);

        if (0==strcmp(keyword,"boundary"))
        {
            int dex=0,dist;
            int i = 0;
            double vals[200];
            sscanf(&buf[dex],"%s%n",keyword,&dist);
            dex+=dist;
            while (1==sscanf(&buf[dex],"%lf%n",&p1,&dist))
```

```
          {
             dex+=dist;
             vals[i++]=p1;
             if ((i>2)&&(i%2==0))
                (walls[nwalls++]=new wall)->
                   init(point(vals[i-4],vals[i-3]),
                      point(vals[i-2],vals[i-1]));
          }
       } else if (0==strcmp(keyword,"frectangle"))
       {
          if (5>sscanf(buf,"%s %lf %lf %lf %lf
%lf",keyword,&p1,&p2,&p3,&p4,&p5))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          squareFurniture *f=new squareFurniture;
          f->init(p1,p2,p3,p4,p5);
          hits[nhits++]=f;
       }else if (0==strcmp(keyword,"fcircle"))
       {
          if (4!=sscanf(buf,"%s %lf %lf %lf",keyword,&p1,&p2,&p3))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          ((roundFurniture *)(hits[nhits++]=new roundFurniture))
             ->init(point(p1,p2),p3);
       }else if (0==strcmp(keyword,"peoplepercent"))
       {
          if (2!=sscanf(buf,"%s %lf",keyword,&p1))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          peoplePercent=(int)p1;

       }else if (0==strcmp(keyword,"peoplerand"))
       {
          if (2!=sscanf(buf,"%s %lf",keyword,&p1))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          peopleRand=p1;

       }else if (0==strcmp(keyword,"person"))
       {
          if (4!=sscanf(buf,"%s %lf %lf %lf",keyword,&p1,&p2,&p3))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          if (probability(peoplePercent))
             (people[npeople++]=new person)->

    init(point(p1+peopleRand*realRand(),p2+peopleRand*realRand()),p3);
       }else if (0==strcmp(keyword,"exitsign"))
       {
          p4=20.0;
          if (4>sscanf(buf,"%s %lf %lf %lf %lf",keyword,&p1,&p2,&p3,&p4))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          (exits[nexits++]=new anexit)-
>init(point(p1,p2),point(p1+1,p2+1),p3,p4);
       }  else if (0==strcmp(keyword,"exit"))
       {
          p5=20.0;
          if (5>sscanf(buf,"%s %lf %lf %lf %lf
%lf",keyword,&p1,&p2,&p3,&p4,&p5))
          {printf("Syntax error on line %d!\n",lineNo);exit(1);}
          (exits[nexits++]=new anexit)->init(point(p1,p2),point(p3,p4),1.0,p5);
       } else if (buf[0]!=0)//Null strings OK
       {
          printf("Unrecognized string '%s' on line %d\n",buf,lineNo);
       }
       lineNo++;
     }
    return true;
```

```
}
```

//std_main.cpp: Win32 main routine

```
#include <windows.h>
// Standardized Windows Main Routine
// mostly stolen from Doug Boling's "Programming Microsoft Windows CE"
//
// Orion Lawlor, 12/1998
#include "std_main.h"

const TCHAR g_szAppName[]=TEXT("LawlorStd");
HINSTANCE g_hInst;

LRESULT CALLBACK MainWndProc(HWND hWnd,UINT wMsg,WPARAM wParam,LPARAM lParam);
LRESULT CALLBACK MainWndProc(HWND hWnd,UINT wMsg,WPARAM wParam,LPARAM lParam)
{
    int i;
    for (i=0;i<messageLen;i++)
        if (wMsg==messageArr[i].Code)
            return messageArr[i].Fxn(hWnd,wMsg,wParam,lParam);
    return DefWindowProc(hWnd,wMsg,wParam,lParam);
}

void InitApp(HINSTANCE hInstance);
void InitApp(HINSTANCE hInstance)
{
    WNDCLASS wc;

    wc.style=0;
    wc.lpfnWndProc=MainWndProc;
    wc.cbClsExtra=0;
    wc.cbWndExtra=0;
    wc.hInstance=hInstance;
    wc.hIcon=LoadIcon(hInstance,"MAIN_ICON");
    wc.hCursor=LoadCursor(hInstance,"MAIN_CURSOR");
    wc.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName=TEXT("LawlorMenu");
    wc.lpszClassName=g_szAppName;

    RegisterClass(&wc);
}

void InitInstance(HINSTANCE hInstance,int nCmdShow);
void InitInstance(HINSTANCE hInstance,int nCmdShow)
{
    HWND hWnd;
    g_hInst=hInstance;
#if WinCE
    int windowStyle=WS_VISIBLE;
#else
    int windowStyle=WS_BORDER+WS_OVERLAPPED+WS_VISIBLE+
              WS_MAXIMIZEBOX+WS_MINIMIZEBOX+WS_SYSMENU+
              WS_THICKFRAME;
#endif
    hWnd=CreateWindow(g_szAppName,winName,windowStyle,
            CW_USEDEFAULT,CW_USEDEFAULT,winWid,winHt,
            NULL,NULL,g_hInst,NULL);
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
}

LPSTR myCommandHack;
```

```
#if WinCE
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPWSTR
lpCmdLine,int nCmdShow)
#else
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
#endif
{
   MSG msg;
   myCommandHack=lpCmdLine;
   if (hPrevInstance==NULL)
      InitApp(hInstance);
   InitInstance(hInstance,nCmdShow);
   while (GetMessage(&msg,NULL,0,0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return msg.wParam;
}
```

//app_main.cpp: maps Windows Message callbacks to CApp methods

```
#include <windows.h>
#include "menu_ids.h"
#include "app_main.h"

#include "std_main.h"
LRESULT DoCreateMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam);
LRESULT DoPaintMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam);
LRESULT DoDestroyMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam);
LRESULT DoMenu(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam);

const decodeUINT messageArr[]={
   WM_CREATE,DoCreateMain,
   WM_PAINT,DoPaintMain,
   WM_DESTROY,DoDestroyMain,
   WM_COMMAND,DoMenu
};
const int messageLen=sizeof(messageArr)/sizeof(messageArr[0]);

const TCHAR *winName=TEXT("MCM Project");
const int winWid=503,winHt=443;

CApp *app;

LRESULT DoCreateMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
   app=new CApp;
   app->init(hWnd);

   return 0;
}

LRESULT DoPaintMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
   PAINTSTRUCT paint;
   HDC crapDC=::BeginPaint(hWnd,&paint);
   ::EndPaint(hWnd,&paint);

   HDC dc=::GetDC(hWnd);

   if (app->paint(dc))
      ::InvalidateRect(hWnd,NULL,false);//for animation et. al
```

```
    ::ReleaseDC(hWnd,dc);
    return 0;
}

LRESULT DoDestroyMain(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
    app->close();
    delete app;
    PostQuitMessage(0);
    return 0;
}


LRESULT DoMenu(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
    int wmId    = LOWORD(wParam); // Remember, these are...
    int wmEvent = HIWORD(wParam); // ...different for Win32!

    //Parse the menu selections:
    switch (wmId) {
    case MENU_EXIT:
        DestroyWindow (hWnd);
        PostQuitMessage(0);
        break;
    default:
        app->menu(wmId);
        break;
    }
    return 0;
}
```

<u>//main.cpp: controls user interface, maintains disaster object</u>

```
#include "global.h"

extern LPSTR myCommandHack;
const static double zoomTable[6]={40.0,20.0,10.0,5.0,2.5};
void CApp::init(HWND Nwindow)
{
    strcpy(inFile,myCommandHack);
    window=Nwindow;
    d=new disaster;
    d->init(inFile);
    runFor=0;
    draws=1;
    zoom=zoomTable[2];
    running=false;
    leaveTrails=false;
}

bool CApp::paint(HDC dc)
{
    RECT r={0,0,10000,10000};
    screen s(dc,zoom,0,zoom,25);
    int i;
    for (i=0;i<draws;i++)
    {//Step draws number of times
        if (running)
            d->advance();
        if (runFor>0)
        {d->advance();runFor--;}
    }

    if (!leaveTrails)
```

```
        FillRect(dc,&r,(HBRUSH)GetStockObject(WHITE_BRUSH));
    d->draw(s);
    return (runFor>0)||(running);
}
void CApp::menu(int id)
{
    switch(id)
    {
    case MENU_STOP:
        runFor=0;
        running=false;
        break;
    case MENU_START:
        running=true;
        break;
    case  MENU_STEP:
        draws=1;
        runFor++;
        break;
    case  MENU_STEP10:
        draws=1;
        runFor+=10;
        break;
    case  MENU_STEP100:
        draws=10;
        runFor+=100;
        break;
    case  MENU_TRAILS:
        leaveTrails=true;
        break;
    case  MENU_NOTRAILS:
        leaveTrails=false;
        break;
    case MENU_RESET:
        delete d;
        d=new disaster;
        d->init(inFile);
        break;
    case MENU_ZOOM4:
    case MENU_ZOOM2:
    case MENU_ZOOM1:
    case MENU_ZOOMHALF:
    case MENU_ZOOMQUARTER:
        zoom=zoomTable[id-MENU_ZOOM];
        break;
    case MENU_1DRAW: draws=1;break;
    case MENU_3DRAW: draws=3;break;
    case MENU_10DRAW:   draws=10;break;
    case MENU_100DRAW:  draws=100;break;
    case MENU_1000DRAW: draws=1000;break;
        break;
    }
    ::InvalidateRect(window,NULL,false);//Redraw
}

void CApp::close(void)
{
    delete d;
}
//That's all, folks!
```

References

[EGAN] M. David Egan, 1978
Concepts in Building FireSafety
John Wiley & Sons, Inc. A Wiley-InterScience Publication
ISBN 0-471-02229-2

[SMITH] Penelope Smith and Patricia C. Kenschaft, Department of Mathematics and Computer Science, Montclair State College, NJ
Evacuating an Elementary School Building
UMAP Journal 9.2

[LSC] National Fire Protection Agency, 1997
NFPA 101 Life Safety Code, 1997 Edition

[BENZ] Gregory P. Benz, 1986
Application of the Time-Space Concept to a Transportation Terminal Waiting and Circulation Area.
Transportation Research Record 1054
Transportation Research Board, National research Council
ISBN 0-309-03970-3

[EXODUS1] E R Galea, J M P Galparsoro and J Pearce. 1993
EXODUS: A simulation Model for the Evacuation of Large Populations from Mass Transport Vehicles and Buildings under Hazardous Conditions.
CIB W14, Int. Symp. and Workshop Engineering Fire Safety in the Process of Design. Univ. of Ulster Sept. 1993, Part 3 pp 11-25.

[EXODUS2] M. Owen, E R Galea and P Lawrence, 1996
The EXODUS Evacuation Model Applied to Building Evacuation Scenarios
Journal of Fire Protection Engineering 1996, Vol.8(2), pp.65-86

[BENZ et.al.] Gregory P. Benz, John S. Chow, Jerome M. Lutin, 1986
PC-Based Pedestrian Flow Simulation Model for Grand Central Terminal

	Transportation Research Record 1054 Transportation Research Board, National research Council ISBN 0-309-03970-3
[PLESSIS]	Matt Du Plessis, 1986 BART Patron Egress/Ingress Study Transportation Research Record 1054 Transportation Research Board, National research Council ISBN 0-309-03970-3
[UFC]	International Conference of Building Officials, Western Fire Chiefs Association, 1991 The Uniform Fire Code, 1991 edition International Fire Code Institute ISSN 0896-9736
[TPFH]	State Dept of Health and Social Services, Office of the Commisioner, March 1978 Transition Plan for Handicapped Implementation of Section 504, Congressional Rehabilitation Act of 1973 and 45 CFR part 84 No ISBN, can be found in state records.
[KR]	Brian W. Kernighan, Dennis M. Ritchie The C Programming Language Prentice-Hall ISBN 0-13-110362-8
[DSNA]	Robert Endre Tarjan Data Structures and Network Algorithms Society for Industrial and Applied Mathematics ISBN 0-89871-187-8
[DM]	Richard Josenbaugh Discrete Mathematics, Fourth Edition Prentice Hall, 1997 ISBN 0-13-518242-5