Pedestrian Evacuation Model in Second Life - Technical Details

Joel Dearden

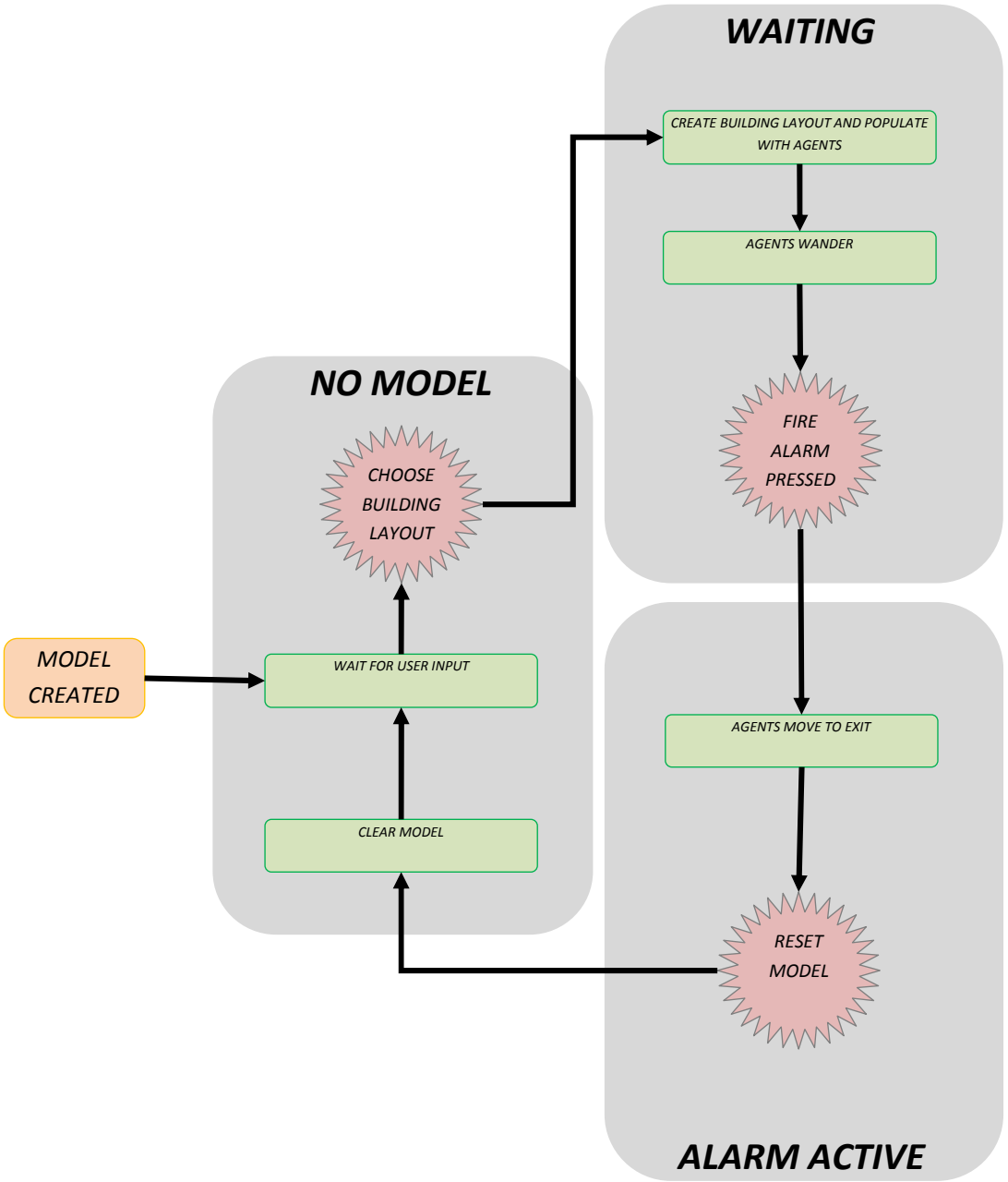
Centre for Advanced Spatial Analysis, University College London

Overview

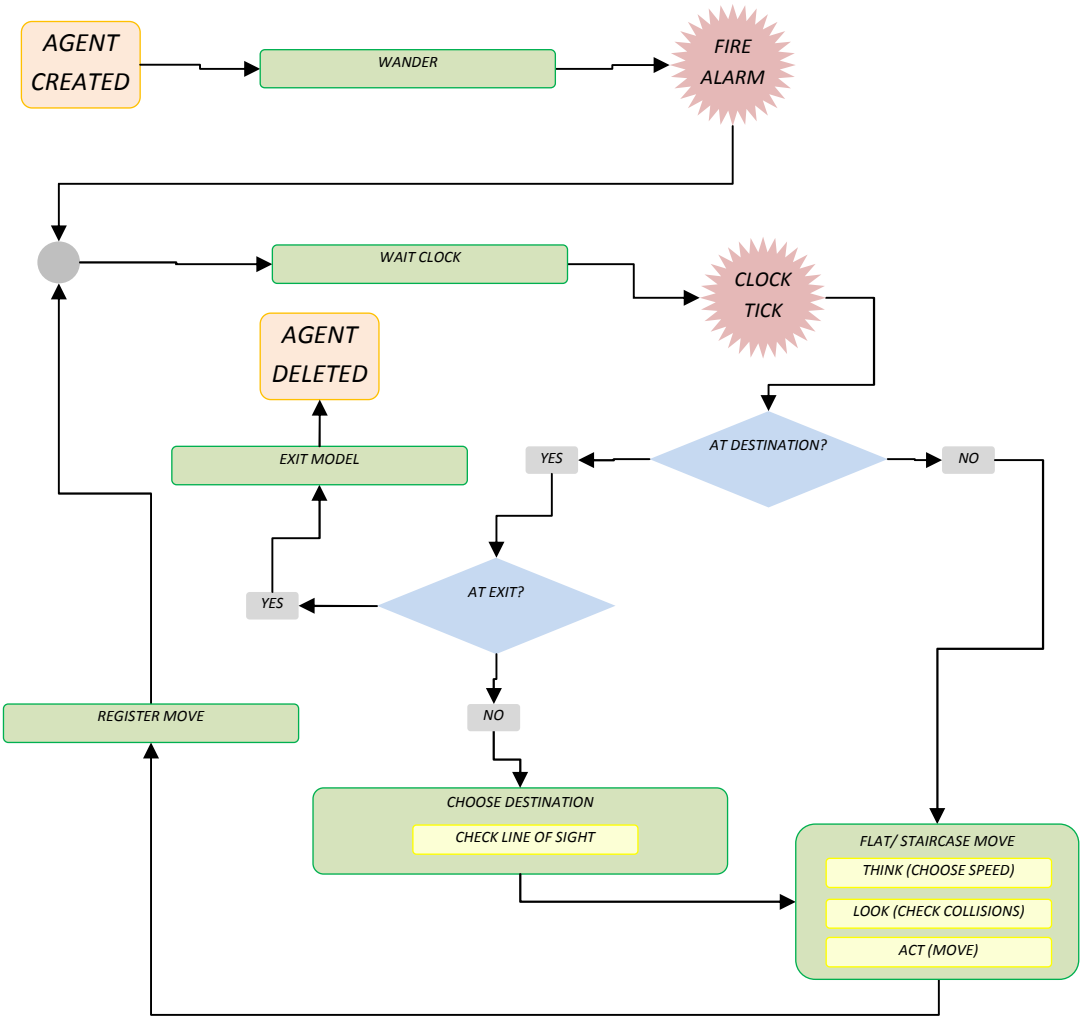
This document is a collection of rough notes about how the CASA's Second Life Pedestrian Evacuation Model works.

The notes were made as part of the development process and details were inevitably changed during implementation and bug fixing so there may be some differences between this document and the real code.

Model Behavior State Diagram



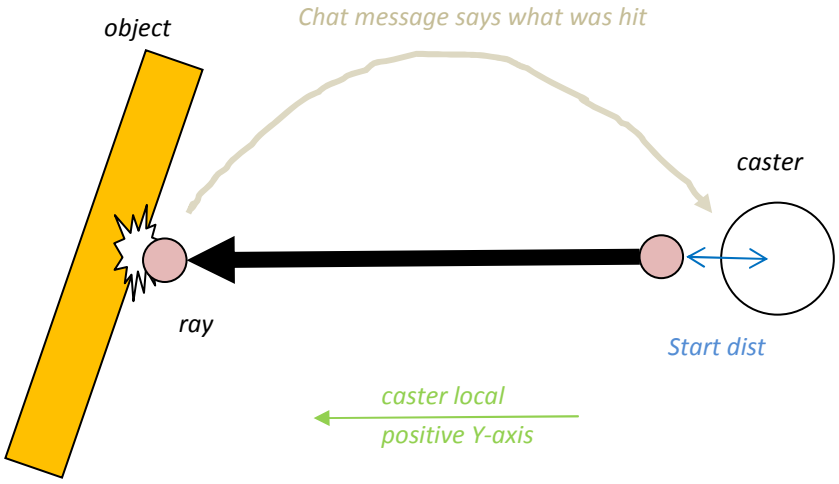
Agent Behavior State diagram



Line of Sight vision system in Second Life

The pedestrian agents need some kind of vision system to determine what they can see at any one time. Linden Scripting Language has a sensor system but this can only detect up to 16 objects.

Instead we use a ray casting method to determine whether an agent has a direct line of sight to a point in space. We basically fire a ray object and see what it hits first:



1. *The caster creates a ray object just in front of itself (local positive Y-axis is front)*
2. *The caster establishes a unique chat channel with the ray using REZ start_param*
3. *The caster listens on that chat channel for any messages from the ray*
4. *The ray moves along the required vector until either:*
    a. *It collides with something **interesting*** – it uses llRegionSay to tell caster what it hit*
    b. *It reaches the maximum vision distance from the caster – it uses llRegionSay to tell the caster nothing was hit – the model must be far enough from the edge of the world so rays don't go off the edge otherwise pedestrians will be waiting around forever for their rays to reply.*
5. *The ray deletes itself*

***interesting** objects are defined as things in the model (walls, doors, etc) – we don't want to know if someone's avatar is wandering around in front of an agent.*

***Reliability:** The ray casting system has some problems due to the unreliable nature of the collision detection system in second life. When a LOS ray is fired it might miss a collision with an object. For this reason the ray object needs to move quite slowly to have a high chance of registering a collision with an object.*

Destination Priority

Pedestrians look for the features of a building in the order given:

1. *exitDoor (it gets you out of the building)*
2. *staircaseEntry (it gets you closer to the ground floor)*
3. *new routeDoor (it helps you progress along the route)*
4. *old routeDoor (it helps you retrace your previous wrong move)*

Pedestrian Velocity Calculation

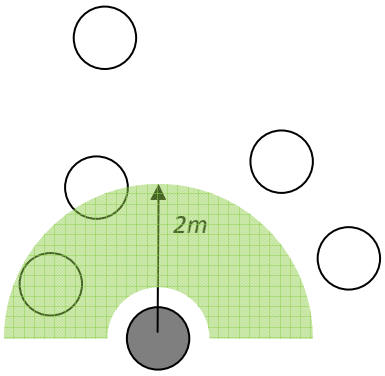
On flat surfaces:

*The velocity of pedestrians will range from **0.4** m/s to **1.5** m/s.*

On staircases:

*The velocity of pedestrians will range from **0.25** m/s to **0.8** m/s.*

Pedestrians will use a 180 arc two metres in front of them to determine how much space there is to move.*



*The area taken up by an average pedestrian is 0.25m * 0.4m = **0.1** m^2*

*The area in the arc is (assuming it is half a circle) = $\pi r^2/2$ = **6.28** m^2*

To calculate the movement speed we:

- *Count the pedestrians within the arc*
- *Calculate the total free space (6.28 - (n*0.1))*
- *Work it out on a scale from MAX = 6.28 to MIN = 0.3*

So if we end up with 4.5 m^2 of free space we calculate the speed multiplier:

(currentSpace – minSpace) / maxSpace

4.5 – 0.3 / 6.28 = 0.668

We then calculate the speed using:

*minSpeed + (speedMultiplier * (maxSpeed - minSpeed))*

so for a pedestrian on a flat plane we get:

*0.4 + (0.668 * (1.5-0.4)) = 1.13 m/s*

Psuedo Code is below:

Before CHOOSING a direction

Perform sensor sweep for pedestrians 180 degrees to front with range of two metres

Count the number of pedestrians

Calculate remaining free space to front

Calculate speed multiplier = (currentSpace – minSpace) / maxSpace

*Calculate speed = minSpeed + (speedMultiplier * (maxSpeed - minSpeed))*

*Now use calculated speed to set **PMV depth** and choose a safe direction to move in.*

*Use calculated speed as **distance to move** when safe direction has been chosen.*

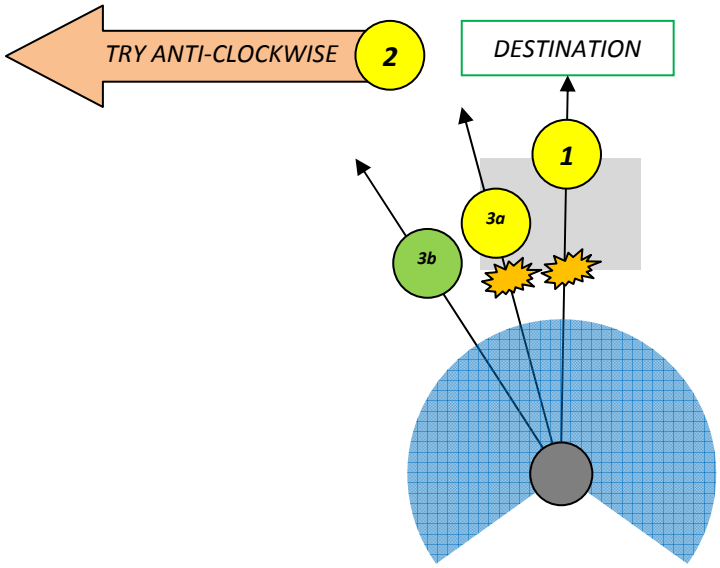
Movement Direction

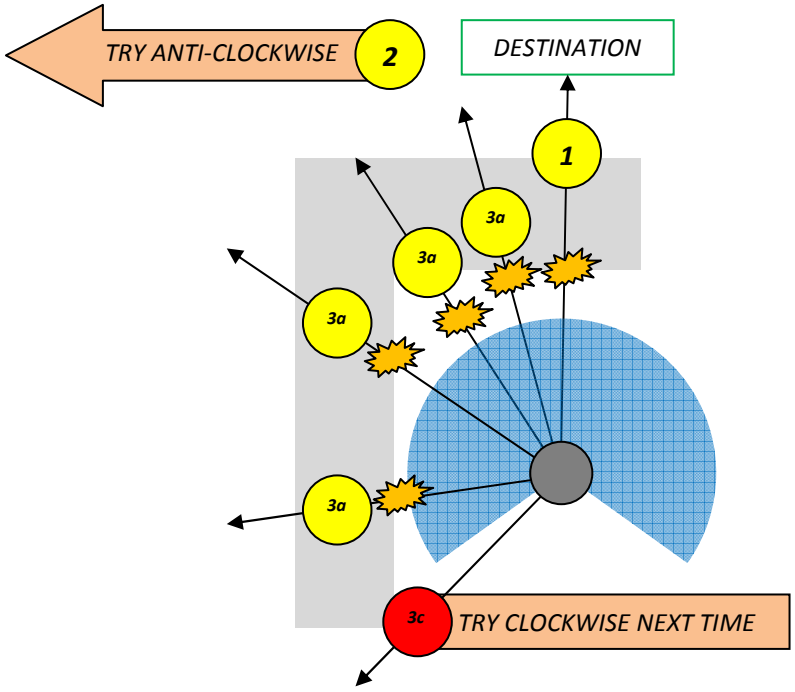
Once a pedestrian has chosen a destination it now has to make a series of moves to reach that destination while avoiding obstacles.

<u>Pedestrians on a flat surface use the following rules:</u>

1. *Try to go straight towards the destination*
2. *If that is not safe (see **collision detection system**) then choose a rotation-direction randomly, either CLOCKWISE or ANTI-CLOCKWISE.*
3. *LOOP:*
    a. *Modify the previous direction a small random amount in your chosen rotation-direction and check if that is safe*
    b. *If you find a safe direction move there – remember this rotation-direction – we will use it for the next model iteration.*
    c. *If the total modification angle exceeds 135* we can't move this iteration of the model - we MUST try the other rotation-direction next iteration (e.g. if CLOCKWISE this time then ANTICLOCKWISE next time)*

The two diagrams below show examples of this process.





<u>Pedestrians on a sloping surface use the following rules:</u>

Behave in the same way as on a flat surface but set the maximum angle at 90.*

Collision Detection System

The collision detection system is the logic that gets a pedestrian to its objective without crashing into any barriers.

*The whole collision detection system is based around the **predicted movement volume** (PMV) of each pedestrian. These help each pedestrian decide if it can move safely in its chosen direction.*

The barriers we are interested in are listed below:

- *walls*
- *other PMVs*
- *other pedestrians*
- *furniture*

The PMV is a temporary object that is placed in front of the pedestrian to represent the volume it plans to move through during the next second. A PMV is created each time a pedestrian wants to check whether a move is safe.

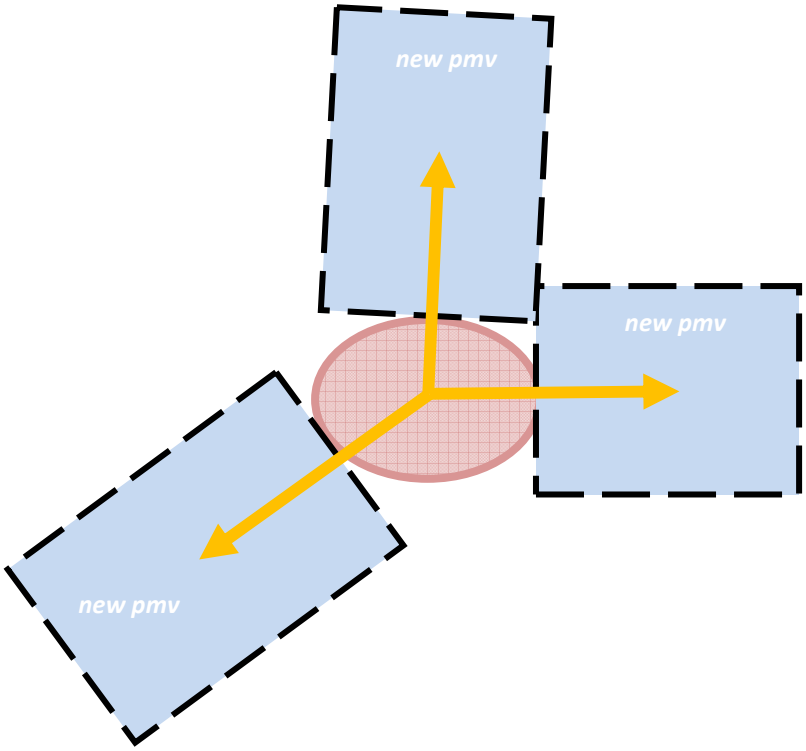
So if a pedestrian wants to check the safety of a move at 2 m/s straight ahead the PMV will be placed in this direction with a length of ~2m and a width and height to match the pedestrian.

As mentioned above we are interested in preventing collisions with walls, other PMVs, other pedestrians and furniture. A PMVs job is to detect all of these objects for its pedestrian owner.

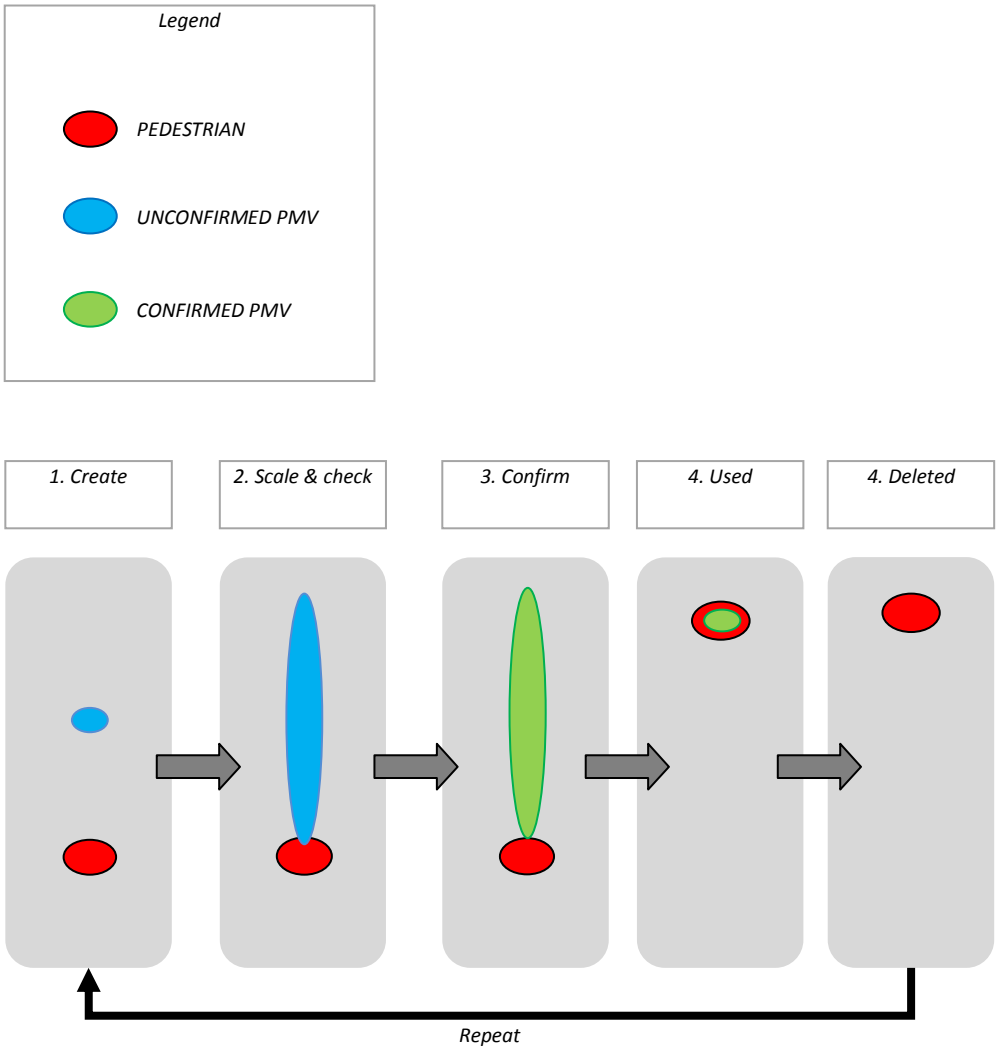
When choosing which direction to move a pedestrian goes through the following steps:

1. *Calculate movement speed (see section above)*
2. *LOOP*
    a. *Create a new **PMV** where you want to go*
    b. *If any [**wall, PMV, pedestrian or furniture**] within our PMV volume*
        i. *Direction is **not safe**: we're either going to simultaneously walk into the same space as another pedestrian or walk into someone or something already there.*
        ii. *Delete the newly placed PMV*
        iii. *Adjust our direction intelligently*
        iv. *REPEAT LOOP*
    c. *Else*
        i. *Direction is safe*
        ii. *Send "confirm" message on PMV channel*
        iii. *END LOOP*

The diagram below shows a pedestrian with three PMVs demonstrating moves in various directions.



Once a PMV is confirmed it moves forward with the pedestrian as it moves and shinks to match the dimensions of the pedestrian. The life cycle of a PMV is shown below:


Legend
PEDESTRIAN
UNCONFIRMED PMV
CONFIRMED PMV
1. Create
2. Scale & check
3. Confirm
4. Used
4. Deleted
Repeat

Every pedestrian is assigned two unique PMV chat channels which it alternates between.

Pedestrian is created

When it wants to move

Delete old PMV

While we haven't found a safe move

Create a new PMV showing the volume we want to move through and into

PMV decides if it is safe and sends a chat message back "safe" or "notsafe"

If safe

Confirm new PMV

Make move

Break loop

Otherwise

Delete new PMV

Adjust direction

Repeat loop

Through testing this system the following improvements have been made:

- *Predicted Movement Volumes now have two states:*
  - *Time-stamped Movement Volume (TMV): the PMV has its creation time in its name so other PMVs can decide if it takes priority over them*
  - *Confirmed Movement Volume (CMV): a PMV that has finished moving and checking collisions and always takes precedence over other PMVs*

Ray channels

Each pedestrian needs to communicate with a number of helper objects:

- *LOS rays*
- *PMV*
- *SPMV*

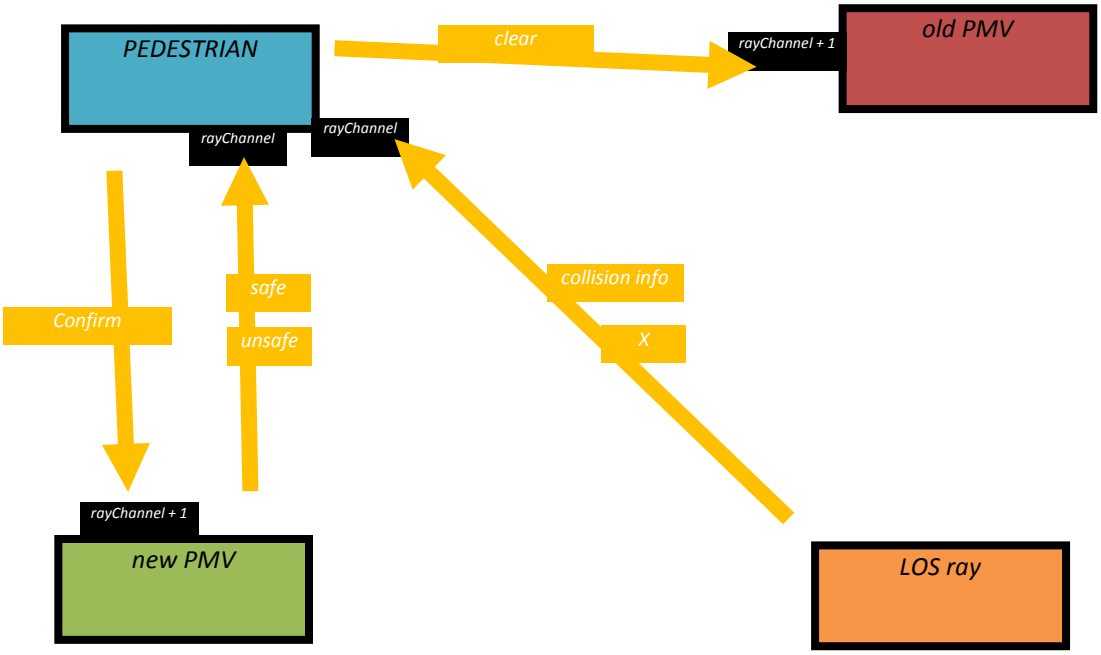
To communicate properly we need two channels:

1. *talk_to_pedestrian channel*
2. *talk_to_<helper> channel*

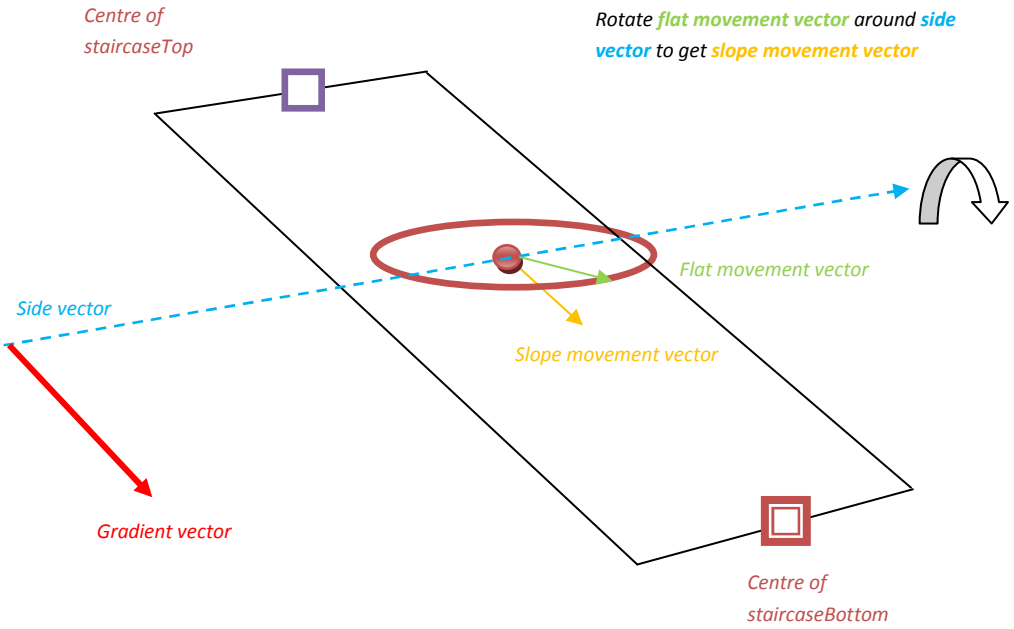
The controller allocates each pedestrian a pair of channels:

1. *talk_to_pedestrian = ray_channel*
2. *talk_to_PMV = ray_channel + 1*

How the channels are used is shown in the diagram below:



Staircase movement



Collision detection is the same as for a flat surface

Movement is different only in that you need to move up and down the Z-axis when moving relative to the bottom of the staircase.

Calculate movement vector as normal and then rotate it around the vector perpendicular to the gradient vector.

Pedestrian visible states

To help with understanding the model and debugging it pedestrians, PMVs and SPMV objects set the text above their head to say what state they are currently in.

RE-CHECK LOS

To prevent pedestrians from getting stuck in corners I have given pedestrians a maximum of ten moves to get to their destination. After this they are forced to choose their destination again (checking LOS) to ensure they are moving towards the most suitable destination.

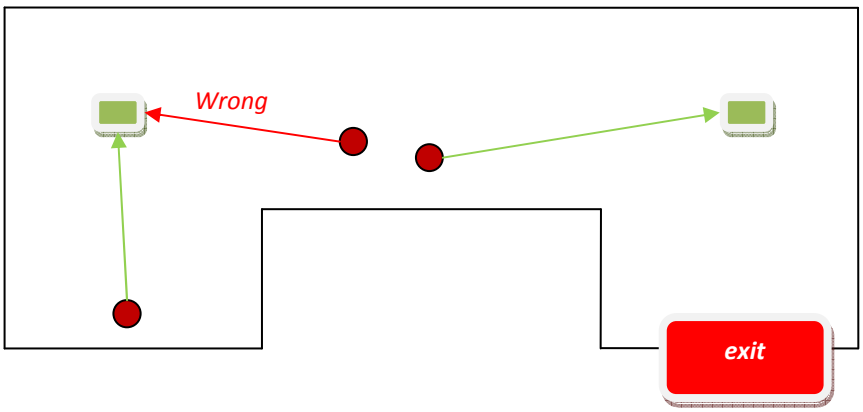
ROUTING PROBLEM

The line-of-sight routing system currently works in the following way:

1. *move towards exit if you can see it*
2. *If not, move towards staircase entry if you can see it*
3. *If not, move towards NEW exitSign if you can see one*
4. *If not, move towards OLD exitSign if you can see one*
5. *If not, move towards nearest exitSign whether you can see it or not.*

This works well for flows of pedestrians all starting in the same place but not for pedestrians that are scattered randomly because they don't know if a route sign takes them away from the exit or towards it.

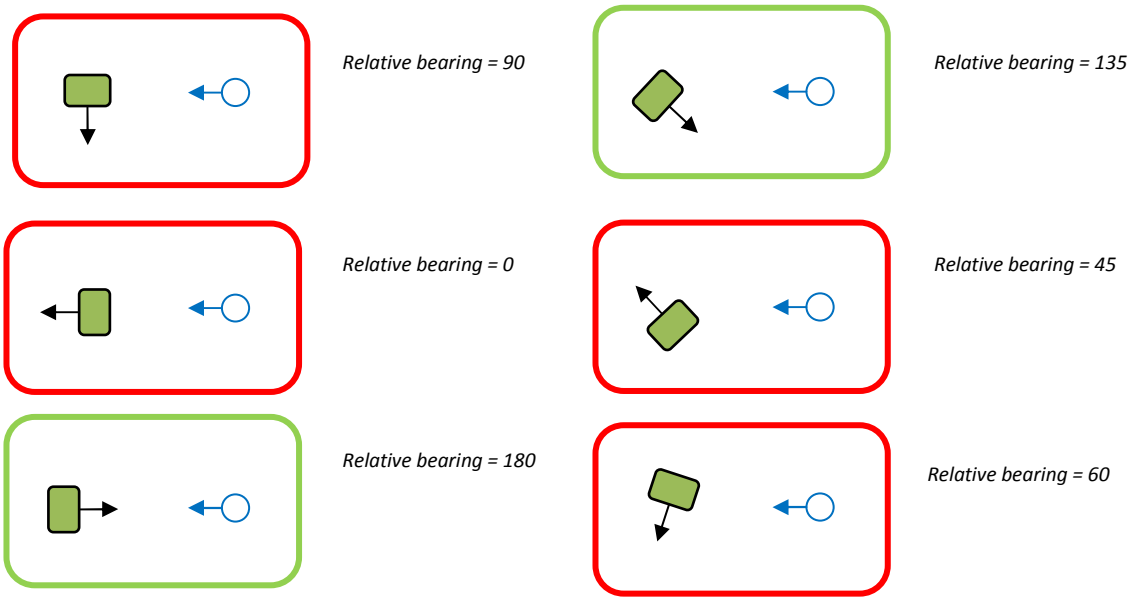
Left agent chooses the "wrong" route sign because it happens to be closest to him and is NEW.



Possible solutions:

1. *Route signs are numbered with a priority manually*
2. *Route signs have a direction associated with them*
3. *There are route signs of different types*

Option 2 is the most realistic and easy to implement because we can use the llDetectedRot to allow a pedestrian to see if a sign is facing towards it or not. Some examples are shown below:



From this we can see the rule should be:

Pedestrians should not go towards route signs that have a relative bearing (i.e. relative z rotation) of less than 135 degrees.

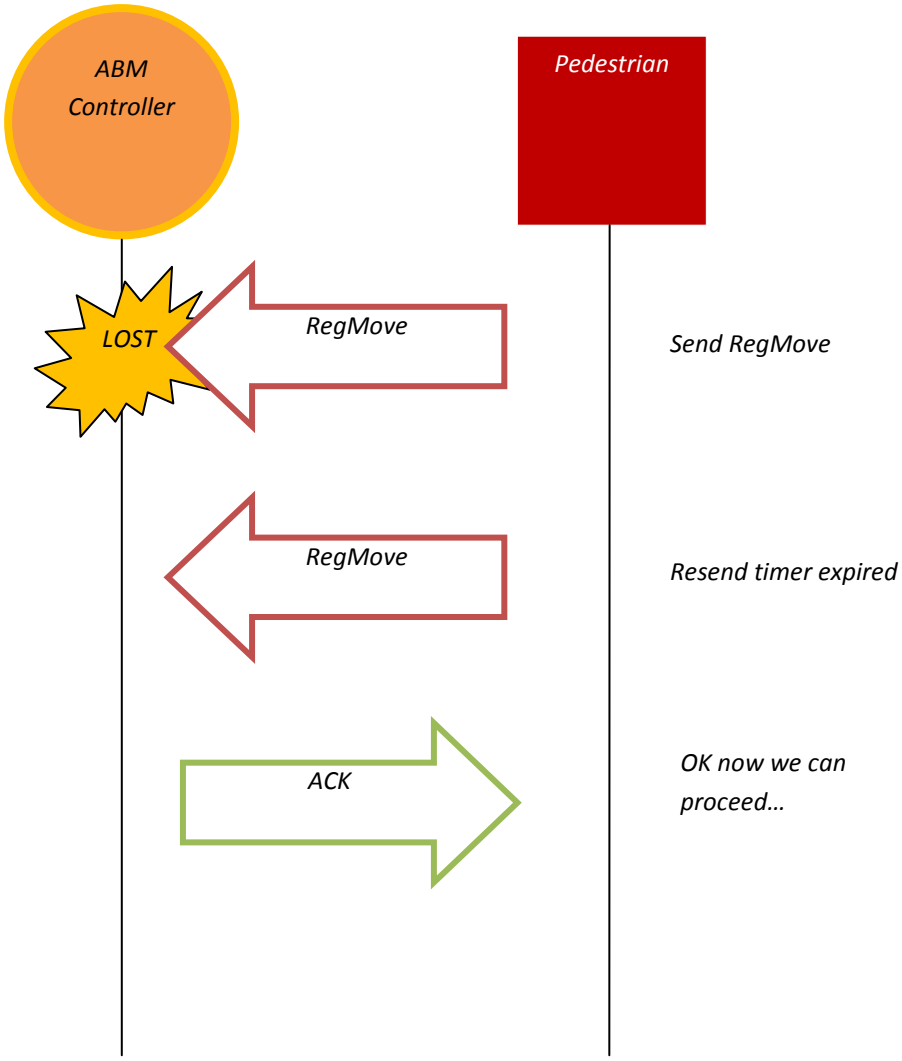
LOST CHAT MSGS

Facts about second life:

- *The event queue (including listen events) is 64 items long – anything else gets lost.*
- *The event queue is cleared on state change.*

The ABM controller object will be receiving a lot of messages when there are large numbers of agents so it might lose a "regmove" message due to its event queue being full. To resolve this, the ABM controller sends an acknowledgement (ACK) message back to a pedestrian to confirm it received the message. The pedestrian receiving the message shouldn't lose the ACK message because it is only receiving a few messages and so its event queue is unlikely to be anywhere near full.

The diagram below shows the message exchange:


ABM Controller
Pedestrian
LOST
RegMove
Send RegMove
RegMove
Resend timer expired
ACK
OK now we can proceed…