Project Journal

5th November 2016

Plans:

1. ~~Research peer-reviewed literature on tile-based game creation, management games, and AI in gaming.~~
2. ~~Begin initial Unity project set up, and create initial basic scripts such as WorldController.~~

List of documents and journals viewed:

1. Tile-Based Game Design – Springer Link
   http://link.springer.com/chapter/10.1007/978-1-4302-2740-3_8
2. Github project – Tile-Based Base building game
   https://github.com/TeamPorcupine/ProjectPorcupine/tree/5027adcc472a3489c21176924cbf8973f4cb2fc2

Journal Details:

1. Advantages of tile-based games –

Array Storage: Creation of new levels is simple due to every level simply being stored as an array.

Collision Detection: Collisions will only happen when the two objects colliding are next to each other, so only neighbouring tiles need to be checked for collisions.

Simplified AI: In a tile-based world, it is very easy to see for the AI to check what is in the surrounding tiles, and so the AI decision making can be dramatically reduced due to it only being able to move to maybe 4 or 8 tiles.

Efficient use of graphics: Since every object is made up of a fixed number of tiles, making a graphic to fit those tiles is very easy, compared to a normal world where you have non-standard sizes and decimal numbers.

Making Tiles –

All the tiles in a game are the same size, and they all share a standard pixel amount too. 64 by 64 is popular, as well as other multiples of 2 since computers can handle these number more efficiently.

Tile sheets are used to allow one large image to replace lots of smaller images for multiply things by putting them all onto one image. Due to all the tiles, and thus the sprites, being the same size, you can easily assign different sprites to different coordinates on the tile sheet.

Similarly to the tile sheet, the game world can be easily split into coordinates, making it easy to match the location needed for the sprite to be, and where it is on the tile sheet.

The tile model –

The tile model is a class that represents every tile, this can be done because every tile has the same base characteristics such as location, and type etc.

Putting the map into the game –

Creating a world full of tiles is easy since every row and column needs to be filled with the same tile. A for loop is good for this since it can go along each row and then up each column and create the world of tiles, it needs to know how tall and wide the world will be first though. Once all the tiles are in the world, and their location is stored as an array, it is easy to manipulate individual tiles based on the position.

Adding Game Characters –

The character occupies a single tiles, or a few tiles, just like the walls and floors do. So they also have a position and therefore can be manipulated easily.

Layering Maps –

Foreground images and background images are different, for example, the walls and floor is different from the character since they cannot move. The background images get loaded first, then the foreground map is checked and if there is supposed to be a character in a tile, the foreground image is then rendered on top of the background image. If the foreground image has transparency, then some of the background should be visible also, which is good.

Project Porcupine code analysis

Code linked to a tutorial series demonstrates a great baseline for what is needed for this project. A lot of the code is not required for our project, but the baseline is what will be looked at and used as inspiration for the backbone of this project.

The code's general set-up is that the visual aspects of the game are separate from the hidden game logic. It uses controllers to link the two together. The WorldController is used to link Unity with the general C# scripts. The WorldController is the center point of the code, and is the only singleton class in the code, all the other scripts, can be reached via this central class, either directly or through other classes, such as the World class. The furniture and character sprite controllers are used to link the characters and furniture logic to the Unity's Monobehaviour gameobject logic. These class will collect the data from the standard C# classes, and use it to put the correct gameobjects in their correct places. The mouse controller contains a simple input system which has access to the game's logic via the WorldController.

The rest of the classes are not derived from monobehaviour and therefore cannot access Unity's gameobject functions and methods, hence the need for the controllers. These general C# classes are used to represent the game logic. There are a number of models which represent the different parts of the game logic. The World class contains the baseline game logic. It contains all the tile data, the character data, the furniture data, the inventory data, as well as other important data such as the game speed, and general global logic.

The tile model represent a single tile in the world, and contains a location, as well as data about the furniture placed on it, and its movement cost.

The furniture model acts as a template for all the furniture in the game. It contains data such as the furniture's tile, its name, any jobs it has, its movement cost etc.

The character model acts as a template for all characters in the game. It contains data such as the current tile the character is on, the destination tile, the speed, the i nventory, the jobs etc.

The inventory model is similar but will not be used in our project and so will not be looked at or used as a reference for development.

The job model is something that will be used in our project, however the logic will be different from the logic in this project as this project uses furniture as its job creation point, but in our project it will be unique to characters, and not a global job request.

Other models such as Room and JobQueue won't be used in our project and so will be skipped.

This project contains a very good, and successful version of the A* pathfinding algorithm, and if A* is the algorithm that will be used in our project, after research is completed, then thi s version will likely be used. The code contains another project's code for a priority-queue system, and if this pathfinding system is used, then the priority system will also be used, but will be properly referenced and the correct acknowledgements will be made for others' work.

A few of the general UI scripts have some useful things in them, and will be used in consideration when the UI is being developed.

<u>Coding</u>

No visible changes on the game screen yet, all background processing.

<u>List of created scripts:</u>

1. World Controller
2. World
3. Tile

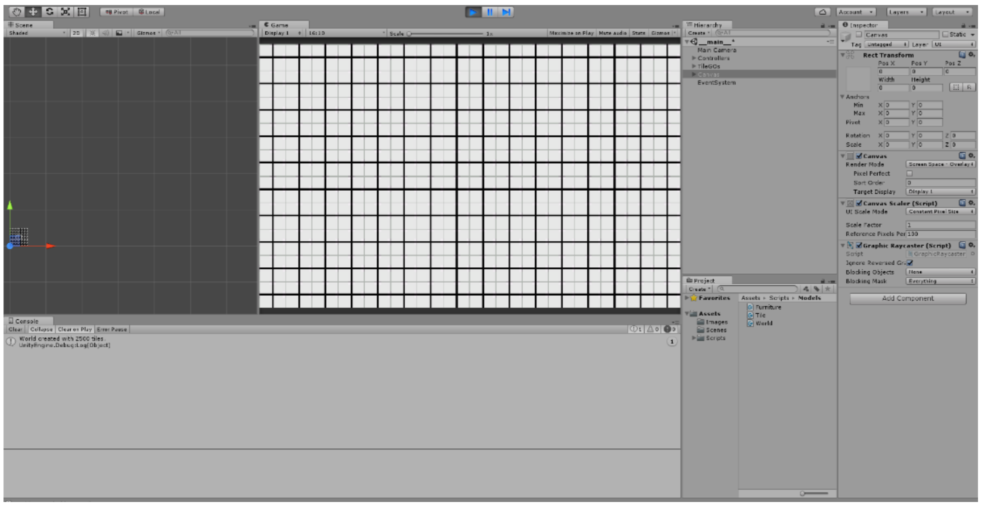
<u>Purposes of created scripts:</u>

1. World Controller – This will contain all other elements of the game. It is where tiles and characters get created and destroyed, and their variables get stored such as position. It will be a singleton due to it being the main controller for the game.

2. World - This is a class not derived from monobehaviour. It will be controlled by the World Controller. It will contain all the elements of the current game such as tiles, characters, furniture etc. It currently knows information about all the tiles and the height and width of the world.

3. Tile – This is a class not derived from monobehaviour. The world is filled with tiles and they are sorted into a 2D array. It will contain only functions that affect itself, such as what character or furniture is on it, or what its neighbours are. It also contains information such as

its position in the World and its movement cost, which is dependent upon what is in the tile such as characters or furniture.



```
World created with 2500 tiles.
UnityEngine.Debug:Log(Object)
```

7th November 2016

<u>Plans</u>

1. ~~Carry on coding and creating the basics for the world.~~
2. ~~Add mouse interactions such as moving/scrolling around the screen.~~
3. Add basics for furniture placement.

<u>Coding</u>

Tile game objects now visible in game. They have a basic sprite.

Camera movement has been implemented. Moving the camera around and zooming in and out works.

Did not get chance to implement basics for furniture placement, will begin with that next session.

<u>Created Scripts</u>

1. Mouse Controller

<u>Purposes of created scripts</u>

1. Mouse Controller – In charge of all mouse movements, clicks, and drags. Interacts with the camera for camera movements. Will eventually deal with furniture previews once they are implemented.

9th November 2016

<u>Plans</u>

1. ~~Add basics for furniture placement.~~
2. ~~Implement furniture~~
3. ~~Add walls to list of furniture~~

<u>Coding</u>

Can now add lots of different furniture with different base types, and sizes.

<u>Created Scripts</u>

1. Furniture
2. Furniture Sprite Controller

<u>Purposes of created scripts</u>

1. Furniture – This is a model which does not inherit from monobehaviour. It is the template for all furniture in the game, including walls and doors.
2. Furniture Sprite Controller – This is a controller in charge of all the sprites used for the furniture. If sprites get changed, or added during gameplay, this class sets all the correct settings for the game objects.

12th November 2016

<u>Plans</u>

1. ~~Finish furniture implementation~~
2. ~~Add UI for furniture placement~~
3. ~~Test furniture placement~~

<u>Coding</u>

Finished adding UI to Unity. Adding more furniture into the game is now very easy.

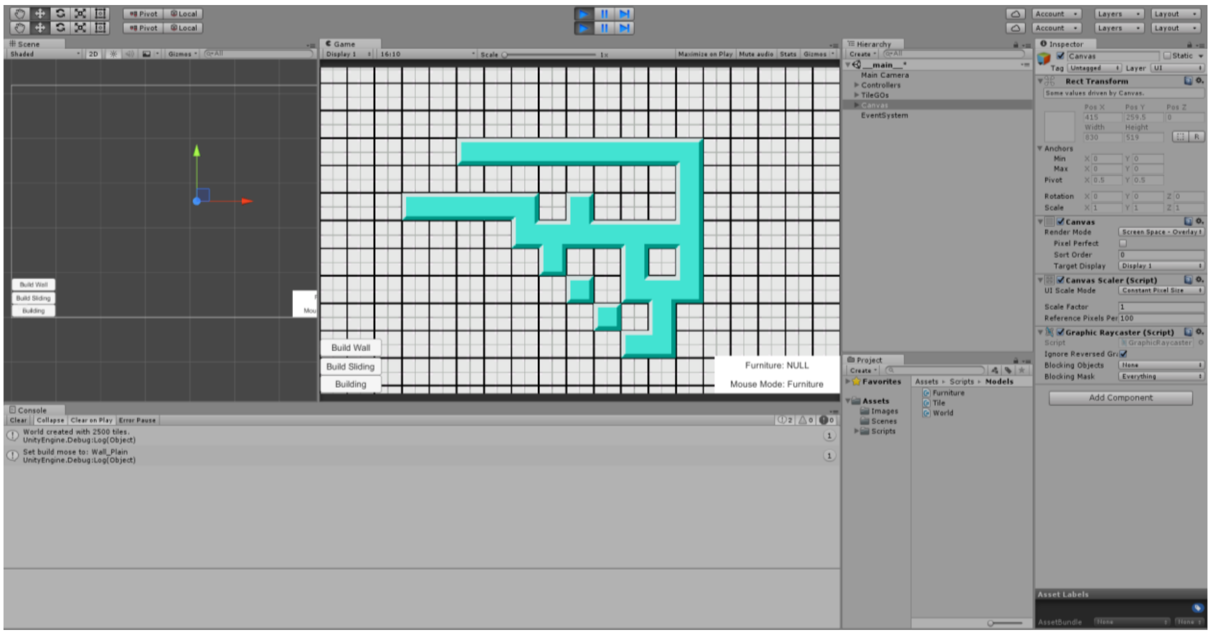
All that that is required is to add the prototype to the world class, and its parameters. Then add a button and the furniture will be created in the world when the button is clicked and a location i s set.

<u>Created Scripts</u>

1. 5 'Helper' scripts that are used to easily change UI menus.

<u>Purposes of created scripts</u>

1. 5 scripts were made to allow a button to be active in the editor so at the click of a button the menu will be re-sized for when new furniture buttons are added.



```
World created with 2500 tiles.
UnityEngine.Debug:Log(Object)
Set build mose to: Wall_Plain
UnityEngine.Debug:Log(Object)
```

13th November 2016

<u>Plans</u>

~~1. Add additional furniture such as doors.~~

<u>Coding</u>

Doors implemented. If a piece of furniture has the 'Door' base type, it can only be placed between two pieces of furniture with the 'Wall' base type, and will rotate if required. Additional furniture will be added when appropriate, for example, to test or demonstrate another system such as pathfinding.

<u>Created Scripts</u>

1. Furniture Actions

<u>Purposes of created scripts</u>

1. Furniture Actions – This is a static script which is used to allow furniture to have some kind of parameters attached to them which update at certain points. For example, the furniture door needs to be open before it can be walked through, and it cannot go from closed to open instantly, it needs some time in between where it is opening. This is used in conjunction with the furniture update function.

15th November 2016

Plans

~~Find examples of NPCs in video games and analysis their game mechanics and UIs.~~

Examples

Rimworld - http://rimworldwiki.com/wiki/Colonist

Each character has different properties that define them and make them unique. These include –

Skills – A character's skills denote how effective they are at relevant tasks. Depending on their backstory, a character may have some skills permanently disabled.

Backstories – Each character has two backstory elements; a childhood and an adulthood. These elements affect the colonist's starting skills and may prevent the use of some skills entirely.

Traits – A character's traits are permanent modifiers that affect their stats like walk speed, work speed, base mood, and mental break threshold.

Mood – A character's mood is the total value of the effects of their thoughts and traits.

Thoughts – Thoughts are a summary of a character's experiences over the last day or so. Thoughts are either positive or negative, and can be generated in huge number of ways.

Health – Each character has an anatomy, with each part having a separate health value. Depending on what gets damaged, the character will be affected in different ways.

Links to Project

Rimworld –

A lot of the properties are useful for the project including skills, traits, mood and thoughts. But health for example if not a factor due to there not being any combat in the game.

22nd November 2016

Plans

1. ~~Code initial character scripts.~~

Coding

Character script is ready to be added to when pathfinding, jobs and stock are added.

Created Scripts

1. Character
2. CharacterSpriteController

Purpose of Scripts

1. Character – This class is not derived from Monobehaviour. It is the model for all characters. It is used when a character spawns. It will deal with that character's attributes such as name, speed, current tile, stock they are carrying etc.
2. CharacterSpriteController - This is a controller in charge of all the sprites used for the characters. If sprites get changed, or added during gameplay, this class sets all the correct settings for the game objects.

28th November 2016

Plans

1. Find literature focused on different pathfinding techniques with advantages and disadvantages.

List of literature

1. Hybrid Pathfinding in StarCraft http://ieeexplore.ieee.org/document/7063238/
2. Direction Based Heuristic for Pathfinding in Video Games http://ieeexplore.ieee.org/document/7124867/

Literature Details

1. Navigation of units in RTS games is typically handled with pathfinding algorithms such as A*. A* always finds the best possible path between two positions in a reasonably short time, but does not handle dynamic worlds very well.
   If the path is suddenly blocked by a mobile object it becomes obsolete and the agent has to recalculate all or parts of it. Extensive work has been done to modify A* to work better in highly dynamic worlds. Silver (D. Silver, 2006) [1] proposes an addition of an extra time dimension to the pathfinding graph to allow units to reserve a node at a certain time. The work of Olsson (P. M. Olsson, 2008) [2] address the issue of changes in the pathfinding graph due to the construction of destruction of buildings. Koenig and Likachev (S. Koenig, 2004) [3] (S. Koenig and M. Likachev, 2006) [4] have made contributions to the field with their work on real-time A*. This paper proposes a hybrid approach for navigating where A* is used when no enemy units or buildings are within sigh radius, and potential fields when unit(s) are engaged in combat. The hybrid approach avoids the problem of local optima when using potential fields (units get stuck on complex terrain) by using A* while at the same time getting the benefits of potential fields for positioning of units in combat situations. The purpose of the paper is to evaluate if the potential fields based part of the hybrid navigation system can be replaced with a system based on flocking algorithms.
   The paper then goes on to explain Boids and its uses, and compares it to potential fields, however this will be skipped due to uncertainty about whether either of these methods will be used for our project.

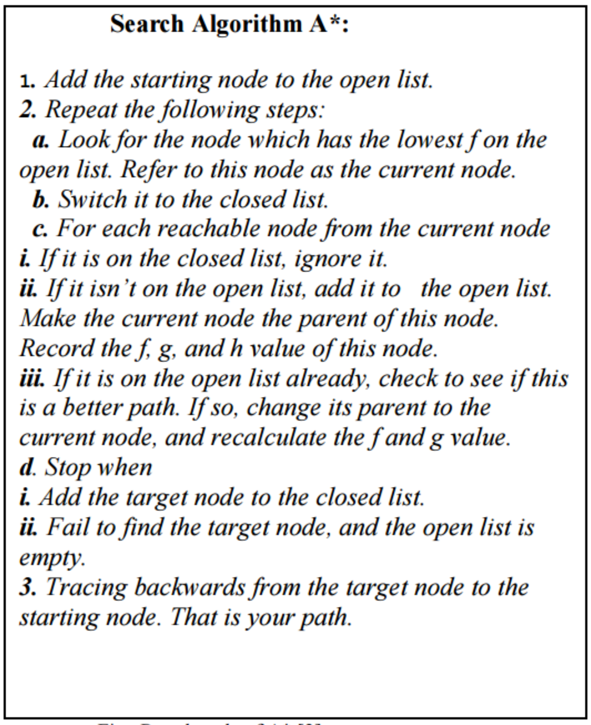
2. Pathfinding is plotting, by a computer application, to find the shortest distance between two points. It starts at a start node and reaches the goal node by repeating searching for the same, for finding a path between these points. Two primary problems of pathfinding are to find a path between two nodes in a graph and to find the optimal shortest path [5]. Pathfinding in the context of video games concerns the way in which an object finds a path

around obstacles; the best explained context is real-tile strategy games in which the player leads units around a play area containing obstacles, but the variations of this approach are found in many of the games.

A* Pathfinding –

A* is a generic search algorithm that can be used to find solutions to many problems, pathfinding is just one of them. Many problems in engineering are related to pathfinding problems. The lookahead effort in searching trees are found to provide improved results in pathfinding. A* is the most popular and widely used AI pathfinding algorithm proposed by Hart, Nilsson and Raphael in 1967. Due to the simplicity it guarantees, A* is almost always the search method of choice. This is because A* is guaranteed to find the shortest path on a graph.

The problem with A* is that a shortest path on a graph is not equivalent to the shortest path in the continuous environment. Another issue related to A* is that, when the map size is significantly larger, A* algorithm cannot find a minimum path to goal state in limited amount of time. Also for larger maps A* uses memory extensively. A* uses this heuristic to improve on the behaviour relative to Dijkstra's algorithm. When the heuristic evaluates to zero, A* is equivalent to Dijkstra's algorithm. As the heuristic estimate increases and gets closer to the true distance, A* continues to find optimal paths, but runs faster. When the value of the heuristic is exactly the true distance, A* examines to find the optimal nodes. However, it is generally impractical to write a heuristic function that always computes the true distance.



Search Algorithm A*:

```
1. Add the starting node to the open list.
2. Repeat the following steps:
  a. Look for the node which has the lowest f on the
open list. Refer to this node as the current node.
  b. Switch it to the closed list.
  c. For each reachable node from the current node
i. If it is on the closed list, ignore it.
ii. If it isn't on the open list, add it to   the open list.
Make the current node the parent of this node.
Record the f, g, and h value of this node.
iii. If it is on the open list already, check to see if this
is a better path. If so, change its parent to the
current node, and recalculate the f and g value.
d. Stop when
i. Add the target node to the closed list.
ii. Fail to find the target node, and the open list is
empty.
3. Tracing backwards from the target node to the
starting node. That is your path.
```

[6]

Fig. Pseudocode of A* [3].

Heuristics –

Heuristics is a method used for experience based problem solving, which may or may not end up with an optimal solution. Algorithm's behaviour based upon the heuristic and cost functions can be very useful in a game. The trade-off between speed and accuracy can be exploited to make your game faster. One way to construct an exact heuristic is to precompute the length of the shortest path between every pair of nodes. This is not feasible for most game maps. However, there are ways to approximate this heuristic:

a. Fit a coarse grid on top of the fine grid. Precompute the shortest path between any pair of coarse grid locations.
b. Precompute the shortest path between any pair of waypoints. This is a generalization of the coarse grid approach.

In a special circumstance, the heuristic can be exact without precomputing anything. If there is a map with no obstacles and no slow terrain, then the shortest path should be a straight line.

On a grid, there are well-known heuristic functions to use:

a. On a square grid that allows 4 directions of movement, use Manhattan distance
b. On a square grid that allows 8 directions of movement, use diagonal distance
c. On a square grid that allows any direction of movement, might or might not want Euclidean distance.
d. On a hexagon grid that allows 6 directions of movement, uses Manhattan distance adapted to hexagonal grids.

<u>Literature Comments</u>

1. This paper is useful in thinking about not just using one algorithm for our pathfinding, but a mixture of different ones to end up with a realistic result. The actual comparison between boids and potential fields is skipped because those particular types of pathfinding are not relevant for this project as boids are most useful when large groups are moving together, and potential fields are useful when certain areas need to be avoided or want to be driven towards. Thinking about a possible hybrid pathfinding system may be useful due to the final game having a dynamic map. The actual layout of the store will not be changed during gameplay where characters will need to be moving around, however, doors will be opening and closing and so a dynamic pathfinding system may be required, but at this stage it is impossible to tell.
2. This paper explains the best algorithm which is A*. A* guarantees to find the shortest path if one if available. This of course is great for the project because this is all we want. However, there are disadvantages such as large maps cause a lot of memory to be needed every time the pathfinding is required. Another disadvantage is that it does not work with dynamic maps, once the cost of a node has been declared, it cannot be changed without the whole algorithm restarting. Both these problems do not apply to our project due to the map never being very large, and the walls and furniture in the game will not change while the characters will be trying to move around, except moveable objects such as stock cages, and trolleys. From the list of heuristic function, the one that will most likely be used for this project will be the 2nd one: On a square grid that allows 8 directions of movement, use diagonal distance.

<u>References Used in the works</u>

[1] D. Silver, “Cooperative pathfinding,” in AI Game Programming Wisdom 3. Newton Center, MA, USA: Charles River Media, 2006.

[2] P.-M. Olsson, “Practical pathfinding in dynamic environments,” in AI Game Programming Wisdom 4. Newton Center, MA, USA: Charles River Media, 2008.

[3] S. Koenig, “A comparison of fast search real-time situated agents,” in Proc. Autonom. Agents Multi-Agent Syst. (AAMAS), 2004.

[4] S. Koenig and M. Likhachev, “Real-time adaptive A*,” in Proc. Autonom. Agents Multi-Agent Syst. (AAMAS), 2006.

[5] Björnsson, Yngvi;Vadim Bulitko ; Nathan Sturtevant. TBA*: Time-Bounded A*. Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09);2009' 431-436.

[6] Björnsson, Yngvi; Enzenberger, Markus; Holte, Robert C. Fringe Search: Beating A* at Pathfinding Game Maps; IEEE 2005 Symposium on Computational Intelligence and Games, 2005, 125-132.

1st December 2016

Plans

1. Find literature focused on different pathfinding techniques with advantages and disadvantages.

List of literature

1. Uninformed Multigoal Pathfinding on Grid Maps
   http://ieeexplore.ieee.org/document/6946181/

Literature Details

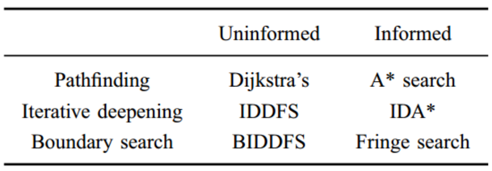
1. There are two classifications of pathfinding algorithms: Informed and Uninformed.

   Informed: Involves the use of a heuristic function [1] to estimate the location of the goal. The direction of pathfinding is guided towards the estimate, making informed searches typically faster than uninformed searches (but can be less optimal)

   Uninformed: Does not use heuristics for pathfinding, and are also known as blind searches. Typically, the search is done in all directions from the starting node and growing outward in a radial pattern.

   Pathfinding algorithms have been developed using different pathfinding methods, such as iterative-deepening searches [2], boundary searches [3], bidirectional searches [4] and multigoal searches [5][6].

TABLE I. EXAMPLES OF PATHFINDING ALGORITHMS



	Uninformed	Informed
Pathfinding	Dijkstra’s	A* search
Iterative deepening	IDDFS	IDA*
Boundary search	BIDDFS	Fringe search

   Pathfinding algorithms are typically applied on topographical maps or grid maps. Topographical maps segment regions based on their elevation while grid into regularly spaced regions. Each regularly spaced region can be represented as a node.

A node can be one of two types, expandable (visitable) or non-expandable (unvisitable). Non-expandable nodes include the starting, goal and obstacle nodes. Expandable nodes only include expanded (visited) and unexpanded (unvisited) nodes.

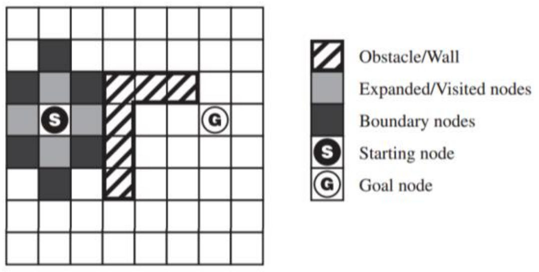
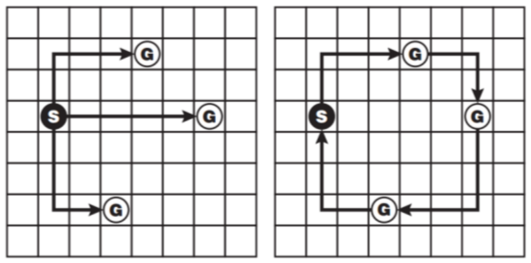


Fig. 1. Pathfinding node types.

There are two implementations of multigoal algorithms: (1) multigoal start-goal pairs, in which the algorithm finds multiple goals by calculating paths for each start-goal node pair [6], and (2) the travelling salesman problem (TSP), in which the algorithm expands from the starting node to a predefined number goals before reaching the ending node, resulting in a single route [5] [7].



(a) Expanding to multiple goals successively from the start node. (b) Expanding to multiple goals from one goal to another.

Fig. 2. Two different outcomes of multigoal algorithms.

To test the speed of the proposed multigoal algorithms, two uninformed search algorithms were used because they represent the worst case scenario in terms of time taken for the search. The algorithms are simplified versions of Dijkstra's Shortest Path (DSP) algorithm and the Boundary Iterative-Deepening Depth-First Search (BIDFFS) algorithm proposed in [8]. The algorithms are modified to continue searching beyond the first goal and will only stop after all the predefined number of goals have been exhausted.
A uniform cost of traversing from one node to another is used. All distances described are rectilinear distances (as opposed to Euclidean distances) in which inly right angle paths are considered.

A. Dijkstra's Shortest Path Multigoal Algorithm;
   This algorithm aims to find the optimal path of least cost (which is the same as the shortest path). During the process the starting node is marked as the current node and all other nodes are marked as unvisited. From the current node, the locations of all unvisited neighbouring nodes are stored in a First-In-First-Out(FIFO) buffer. The cost from all neighbouring nodes are calculated and saved in an array. The neighbouring node with the lowest cost will be the next node to be expanded. In the next iteration, it will become the new current node and the previous current node with be marked as visited. This is repeated until the goal is reached or until all nodes have been visited.
B. Multigoal Boundary Iterative-Deepening Depth-First Search
   The iterative-deepening depth-first search (IDDFS) searches each possible path in turn until it reaches a threshold or goal. The threshold determines how far(or deep) along

the path it will go before stopping on that path and moving on to the remaining paths. For pathfinding, this threshold is increased from one till it reaches the goal. Each time the threshold is increased, the search starts all over again as if there were no previous searches. This can help save memory but introduces redundancy especially along deeper paths on larger maps where the threshold can be a larger value.
The BIDDFS is based on the IDDFS but eliminates the need for repeating all previous operations in a search when it iterates in a new threshold. Instead, it starts the search from the boundary nodes from the previous iteration. The fringe search [3] achieves this by storing the boundary nodes of a previous iteration. The BIDDFS achieves this by scanning the map for boundary nodes at the start of every new iteration. Boundary nodes are identified by locating nodes that neighbour expanded nodes. The BIDDFS algorithm starts by locating the boundary nodes around the starting node and stores it into the buffer, and for every node expanded its location in the buffer is removed and that location marked as visited. When the buffer is exhausted, the threshold increases by one. Then, boundary node locating commences to find a new set of boundary nodes from which to expand. This expansion process repeats itself until the goal is reached or until all nodes have been visited.

The conclusion found from the tests conducted show that the multigoal algorithms show better time efficiencies on both maps with and without obstacles. There is an exponential increase in pathfinding time recorded by single-goal algorithms when searching for multiple goals on open maps. Results show that time taken by single-goal algorithms to search for nine goals can be reduced by up to 458% when multigoal algorithms are used. The BIDDFS is also slower than the Dijkstra's but on actual maps the difference in the time taken is less pronounced and within 5% of each other.

Literature Comments

1. Multigoal algorithms isn't something immediately useful for the project and therefore will not be implemented into the initial pathfinding system. However, it is useful to acknowledge the improvements between single goal and multigoal algorithms. I situation may arise within the game where a character needs to go to a number of location before finally finishing by going to a final location. For example, a customer has asked for a certain type of stock that is not on the shop floor; a worker may need to go and look in the warehouse, and maybe in other locations, to see if they stock is in store somewhere. Simple A* pathfinding may take the worker on a non-optimal path to find the item in a number of locations, but if multigoal pathfinding was implemented alongside the A* pathfinding using a hybrid algorithm, the most optimal path may be used which would of course be better and more realistic. Also, due to A* being a version of Dijkstra's algorithm, the Dijkstra's Shortest Path Multigoal Algorithm may be quite simple to implement into the A* algorithm.

References Used in the works

[1] S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards, Artificial Intelligence: A Modern Approach. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[2] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," Artif. Intell., vol. 27, no. 1, pp. 97–109, Sep. 1985. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(85)90084-0

[3] Y. Bjornsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, "Fringe ¨ search: beating A at pathfinding on game maps," in In Proceedings of IEEE Symposium on Computational Intelligence and Games, 2005, pp. 125–132.

[4] C. Moldenhauer, A. Felner, N. R. Sturtevant, and J. Schaeffer, "Singlefrontier bidirectional search," in Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010. AAAI Press, 2010.

[5] L. Hongyun, J. Xiao, and J. Hehua, "Multi-goal path planning algorithm for mobile robots in grid space," in The Proceedings of the 25th Chinese Control and Decision Conference (CCDC), May 2013, May 2013, pp. 2872–2876.

[6] A. M. Parodi, "Multi-goal real-time global path planning for an autonomous land vehicle using a high-speed graph search processor," in Proceedings of the 1985 IEEE International Conference on Robotics and Automation, vol. 2, Mar 1985, pp. 161–167.

[7] M. Werner, "Selection and ordering of points-of-interest in large-scale indoor navigation systems," in Proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC), 2011, July 2011, pp. 504–509.

[8] K. L. Lim, L. S. Yeong, K. P. Seng, and L.-M. Ang, "A simplified implementation of the boundary iterative-deepening depth-first search algorithm," in Proceedings of the 13th International Conference on Electronics, Information and Communication, ICEIC 2014, Jan 2014, pp. 173–174.

3rd December 2016

Plans

~~1. Find literature focused on different pathfinding techniques with advantages and disadvantages.~~

List of Literature

1. Pathfinding in Partially Explored Games Environments
   http://ieeexplore.ieee.org/document/6930151/

Literature Details

1. One of the distinct differences between human players and a Non-Playable Character (NPC) is that often the NPC is given information about the whole environment prior to solving the pathfinding process [1]. This gives the NPC an unnatural awareness of its environment which can result in an unrealistic behaviour when navigating. The proposed system aims to address this issue as the NPC has to plan the path to a given target through a fully unknown environment. As the NPC moves along the path it builds an internal map of the environment and recalculates the best path based on this internal map.
   The paper then goes on to explain A* pathfinding and Dijkstra's algorithms. These can be skipped as this is explained in a previously reviewed literature.
   The proposed system uses a hybrid approach [2] as the system will sense the environment, plan a path and move. If the NPC detects a change in the environment the system will re-evaluate the path.

The system was developed in Unity3D and uses the Raycast function to act as sensors. The NPC simulates three sensors as shown in figure 3. The sensors sweep through a radius of a 90-degree angle giving the NPC a viewable range of 270 degrees.

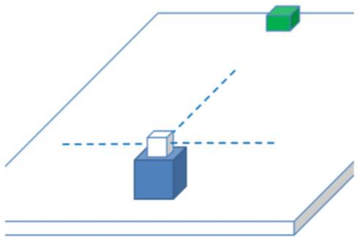


Fig. 3. Simulated sensor positioning

The Raycast function acts in a similar way to a Sonar or LIDAR based system in that a single point is projected outwards. If the Raycast collides with an object, the system records the collision point. The NPC is able to detect any object in the environment which is set as a physics collider.

When the simulation starts the only information the NPC is given about the environment is the position of the target destination.

Occupancy Grids

The Occupancy grid is stored as a multidimensional array that represents a grid of 100x100 double data types and represents the certainty of uncertainty of object in the environment. This gives the NPC a memory space of 100,000 doubles which requires 9.8 Kilobytes of storage. Each value represents a given area of the environment and the size of each area can be adjusted the through the use of resolution variable. The values stored represent the probability that the square is occupied and with the NPC not been given any prior information about the environment each square is defaulted to 0.5. This represents that each square is neither occupied not empty.

During very iteration of the simulation the NPC take sensor readings. If the sensors detect an object, the system recalculates the occupancy values for each square in relation to the sensor's reading position.

The paper then goes on to explain the different calculations used depending upon the distance from the source square the observed object is. These finer details are not relevant currently and so will be skipped, but may be returned to later if this system is implemented into the program.

Literature Comments

1. This is a very interested topic because it may become very relevant if advanced AI pathfinding is implemented. Later in the project, when the characters are being developed, line of sight is something that may be implemented. If this is the case, then this system of partially explored game environments may be something useful. Imagine the case of a customer entering the shop for the first time, they should not know the layout of the shop or where things are located. This system could then be used and it should create a realistic version of the A* pathfinding where the customer takes a non-optimal path because they simply don't know the best way to go. This would improve if they returned to the store because their previous encounter information could stay and then they will be able to make a better guess on the right path to use. A concept of 'shop familiarity' is something that is planned to be implemented in later stages and so this advanced pathfinding will hopefully be an important part of it.

This idea of advanced pathfinding will not be implemented until the 'line of sight' system is implemented and so a simple A* pathfinding algorithm will be used initially.

References Used in the works

[1] Bourg, D.M., (2004) AI for Game Developers 1 Edition. O'Reilly Media

[2] Ronny Hartanto, 2011. A Hybrid Deliberative Layer for Robotic Agents: Fusing DL Reasoning with HTN Planning in Autonomous Robots 2011 Edition. Springer.

General Thoughts on pathfinding

After reading up on different techniques and hybrid options, it has been decided that A* is the best algorithm for the project. A more advanced hybrid pathfinding may be used further into the project when line of sight or other more advanced features are also implemented, and those will be discussed when relevant.

5th December 2016

Plans

~~1. Implement A* Pathfinding~~

Coding

Pathfinding in theory implemented. Cannot be tested until UI is created to allow a move order to be given to the character in the game. This will be done next session. A set of scripts were used to implement Priority Queues which simplify the code and allows the openSet queue to be ordered by lowest node cost (available here: <https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp>).

The world now has a reference to the tileGraph so that every character can share the same tileGraph information, to save computing power. When a piece of furniture is placed down or deleted, the tileGraph invalidates and is re-created.

Every piece of furniture now needs a movement cost associated with it. The higher the movement cost, the longer it will take to travel through the furniture. A cost of 0 is special and means it cannot be travelled through at all, like a wall.

Each character now has an update function, which every frame checks to see if the character should be moving. The character then goes through the process of working out what is the next tile they need to move to, with the help of the pathfinding path, and then lerp to the next tile. Before entering the tile, it checks to see if it is valid and if it isn't the tileGraph must be incorrect and a new one if created. If the next tile is enterable soon, the function returns which will cause the character to stop moving. If the tile is enterable, the character checks to see if there is a piece of furniture there; if there is, the movement speed is slowed according to the movement cost of the furniture. A SetDestination function is also created which simply sets the m_destTile variable and will trigger movement on the next frame if needed and allowed.

Scripts Created

1. Path_AStar
2. Path_TileGraph

3. Path_Edge
4. Path_Node

<u>Purpose of Created Scripts</u>

1. Path_AStar – This is where the main pathfinding algorithm takes place. It uses the Path_TileGraph to find a path from and to two given tiles.
2. Path_TileGraph –This class creates a simple path-finding compatible graph of the world's tiles. Each tile is a node. Each walkable neighbour from a tile is linked via an edge connection.
3. Path_Edge – Every node is linked to another node, that it can access by movement, by an edge.
4. Path_Node – Every node has an array of edges leading out from it, and is used to determine if a neighbouring node can be reached.

9th December 2016

<u>Plans</u>

~~1. Test pathfinding code and fix if not working.~~

<u>Coding</u>

Added 'Go To' button which changes the mouse mode in the MouseController Class to CharacterWalk. While in CharacterWalk mode, clicking a tile will cause the first character's m_destTile to change to that tile. Then during the next update, the difference sh ould be recognised and the pathfinding should begin to find a route the destination tile.

On initial testing, the character was not moving. Debugging found that the character's position was moving correctly, but the GameObject was not updating.

Added OnCharacterMoved function to CharacterSpriteController which runs if a character's OnChanged callback is active, and is activated when a character is created. The function checks to see if the given character is in the GameObjectMap, and if it is, the Character's GameObject Coordinates are set to the correct values.

After testing again, pathfinding appears perfect. The calculation is instant and the movements are correct. The opening of doors has now been tested due to the character's ability to walk through them. The doors' opening 'animation' is correct as intended. Default movement speed of character and opening of doors have bene changed to a reasonable rate, however, these will of course be reviewed at a later time when more relevant.

The sprites' layering was incorrect. This was proven by the character's head being below the wall sprite. All the other sprites would have been incorrect too. This is now fixed, and the rendering order is correct for all current sprite types.

3rd January 2017

Thoughts about approaching General AI Research

Research will be done on Non-Playable Characters (NPCs) in video games. The player cannot directly control the workers, and of course the shoppers; this means that they are all NPCs and so this research I think will be the best. The characters not only have basic gameplay mechanics such as skills, but also more abstract mechanics such as moods, thoughts, and opinions of other characters. For example, a CSA having a bad relationship with a supervisor may reduce productivity and perhaps they do not obey the orders that you give them through the supervisor. Research will also need to be done on coding job assignments and optimization of this code, but this will most likely be done during the

Aims

1. Research and analyse literature focused on NPC development and coding.
2. Look for literature focused on NPC moods and thoughts.
3. Look for literature focused on NPC decision making based on player decisions.

List of Literature

1. Emotion-based Synthetic Characters in Games
   http://ieeexplore.ieee.org/document/4667798/
2. Towards the design of a human-like FPS NPC using pheromone maps
   http://ieeexplore.ieee.org/document/6659132/

Literature Details

1. Historically, Nonplayable Characters (NPCs) are generally given direct access to the game data, free to extract whatever they need, logically they may know everything and never forget whatever known, but it is unfair to Player Characters. Nonplayable Characters themselves may explore the same world as well as Player Characters while obtaining knowledge and achieving their goals.

   Synthetic Characters have a virtual body and they are subject to the constraints of their environment. Many game players, and even developers, would consider synthetic characters the “proper” way of dealing with AI Nonplayable Characters [1].

   Actually simulating Synthetic Characters enables developers to add biologically plausible errors to the iteration with the environment. Including such biologically plausible details allows Synthetic Characters to behave more realistically.

   To most AI developers, such research in biologically plausible emotion systems sounds extremely promising. In many ways, artificial emotions represent an ideal complement to classical AI. Therein reside our interests from a game developer’s perspective: emotions are a key factor of realism and believability. With emotions, all NPC behaviours would seem more realistic and generally increase the immersiveness of the game environment. Each of these features increases entertainment value.

   A model is proposed for customizing automatically NPCs according to the player’s temperament and players can enjoy characters with personalities that reflect human behaviour [2]. In some experiments, the NPC can change its facial expression according to its emotion like the human to attract game players [3]. Karim Sehaba explains an emotion model for Synthetic Characters with Personality [4].

Cathexis [5], a computational model of emotions, is presented, which addresses a number of issues and limitations of models proposed before including the need for models of different kinds of affective phenomena, such as emotions and moods, the need to consider different systems for emotion activation, and the need for flexible way of modelling the behaviour of agents. Another model, which is an extension of the Cathexis model, was built to integrate perception, attention, motivation, emotion, behaviour and motor into specific circuits.

<u>The Model</u>

There is agreement in that emotion includes an expressive or motor component. Some of the aspects involved in this expressive component include central nervous system efferent activity, prototypical facial expressions, body posture, head and eye movements, vocal expression, and muscle action potentials. Finally, most researchers would agree that once an emotion is generated, it registers in consciousness [5]. In response to an external stimulus, the emotion component and the memory component both extract a reduced set of essential features. In the emotion component, there are two direct maps. One is between stimulus and desirability and another is between emotions and moods. On the other hand, in the memory component, the stimulus is temporarily stored.

Ebbinghaus discovered that people forget 90% of what they learn in a class within thirty days, which has been confirmed by those who followed him [6]. We simulate this characteristic of human through assigning fuzzy values to what the memory component stored, and the fuzzy values decay with time elapsing

equation 1 where $\alpha \in [0,1]$ denotes the fuzzy value and $b>0$ denotes forgetting rate.

$\alpha = ae^{(-bt)} + c.$

When the fuzzy values become below a certain threshold, the contents relating to those fuzzy values will be deleted, as though a synthetic character totally forgot them. If the intention component decides to pay attention to the stimulus, it sends a signal to the memory component. As a result, the memory component creates a cognitive image which is rich enough to allow a fairly good reconstruction of the original stimulus and the original stimulus is reserved much long time. Once an emotion or mood is selected by the intention component with its intensity above a certain threshold which is not the same value as generating expressions, then the intention component sends a signal to the memory component and later selects a proper motor. Sometimes the intention component can generate an emotion to either replace or inhibit the current emotion, for example, a policeman searching a robber can generate an intension or an expectation to inhibit his fear.

<u>Basic Emotions</u>

The expression of basic emotions has been used in many different ways. In this thesis, the term basic is used to emphasize how evolution has played a significant role in forming the unique and common characteristics that emotions exhibit, as well as their current function [7]. However, as a first step towards addressing the complexity, the model deals with the following basic emotions: Happiness, Anger, Fear and Sadness. Expectation is also included in the model in order to simulate the comparisons of external events with goals. Ekman has maintained that emotions can be very brief, typically just a couple of seconds and at most minutes [7], i.e. once an emotion is generated, it does not remain active forever. After some period of time, unless there is some sort of sustaining activity, it disappears. In our model, we assume that emotions only last a few time cycles.

2. There are several mechanisms that are used to model the decision process of a NPC such as decision trees, hierarchical state machines, behaviour trees, fuzzy logic, etc. each with their advantages and disadvantages [8]. For example, decision trees in which every branch node is a condition and leaf node is an action, are simple to represent and use but does not scale well and is hard to modify.
   In Quake III, as a typical game AI implementation, the agents use multiple decision trees embedded inside a state machine [9]. The state machine represents states an agent can reach in the game such as Battle Fight or Seek Long Term Goal. At every think frame, the agent goes through the network of states and finds the most appropriate one, best suiting the agent's current situation. In each state, there exist a structure of if-then-else, representing the decision making process of that agent in that particular state. Moreover, inside the structure new states can be reached.
   An indicator is how players act when they are in a particular state, e.g., in low health. Players are expected to take more defensive instances when they are in low health and are vulnerable or go to location where health pack, armour, or similar items exist. Therefore, a difference map as discussed before but taking into account a particular state (e.g. health < threshold) can act as a measurement of their performance. Similarly, the frequency with which players pick up items and the percentage of picking up different items, can also se rve as a measure of performance.
   Strategic decision making of bots is fairly hard to measure without considering the specific situation and since a single best solution rarely exists. However, we can measure how on average NPCs performs. We do this by comparing locations that human and NPC are typically killed or are able to score a point. These places highlight the vantage points good for attacking others as well as vulnerable locations.
   Players are much more likely to move towards items such as health packs and armours. Particularly, NPCs on average where 10% less likely to pick up an item during the game in comparison to humans.
   Long term goals of NPCs in a FPS game, can be goals such as attacking an area in the game, defending the base, providing support for teammates, or maintaining strategic locations. The pheromone map can provide a summary of locations under attack as well as vulnerable positions (with regards to visibility and game mechanics) to help the NPC choose the target destinations. For example, when the player is in attack state it can help choose goals that are more strategically vulnerable. Note that while many options may exist on the map, their attractiveness is dependent on the amount of pheromone for that location as we ll as their Euclidean distance from the player
   An example of short term goals would be picking up an item while pursuing a long term goal. These decisions in particular are related to the current state of the player. For example, a player with low health tries to pick up a health pack on it's way to attack the enemy base. The pheromone map provides necessary information for choosing between items and selecting a suitable path to pick up the health pack while minimizing the threats.

<u>Literature Comments</u>

1. This paper goes into very high detail about emotionally states and also references facial expressions as well as emotions and moods. Facial expressions are not relevant for this

project however, the generally idea of linking emotions with moods is something that may be used in this project. How characters interact with each other is a very important part of the realism involved with the AI; the moods they are in and their likes and dislikes towards each other add character to the characters and brings them to life. Some kind of fuzzy logic system described in this paper could be used for the project in terms of their short-term emotional reactions to events in the environment and their interactions with other people, and their long-term moods throughout the day, which in turn will affect their work output and general disposition to their job and their colleagues. Similar to ideas in this paper, emotions should drain through time, depending on their initial affect, and similarly, their moods should also 'drain' or 'change' through time.

2. Although this is a FPS NPC paper, which is not relevant for the project, it does give some good ideas about learning AI. Pheromone maps may be useful in creating an AI which adapts to its environment and events that happen. For example, if the queue for the till is currently very low, and there are not a lot of customers in the shop, the employees may leave the tills where they were ready to serve more customers, and go do something less important. However, if the shop has been busier than normal, they could use that information to make a different choice about their next action and perhaps choose a task they wouldn't normally choose but it is done closer to the tills so that they can quickly move to serve more customers when they need to. Normally, the employees would ignore the tills since no one is in the shop and it isn't required, but due to the overly busy day a huge wave may come in at any time and would need to be reacted to. The fact that it has been a busy day could be something built into the NPCs by them remembering the amount of customers over a certain period of time, but their response to it would need to be adjusted depending on how familiar they are with working that particular shift so they can make an informed decision about whether the amount of customers is actually above average, or if they are told by a colleague that it is busier than normal.

<u>References Used in the works</u>

[1] A. J. Champandard, AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors, New Riders Publishing, Indianapolis, 2003.

[2] H. Gomez-Gauchia, and F. Peinado, Automatic customization of non-player characters using players temperament, Technologies for Interactive Digital Storytelling and Entertainment, Third International Conference, TIDSE 2006, Proceedings (Lecture Notes in Computer Science Vol. 4326), p 241-52, 2006.

[3] C. Kozasa, H. Fukutake, et al, Facial animation using emotional model. Proceedings, Computer Graphics, Imaging and Visualisation Techniques and Applications, p 428-433, 2006

[4] Karim Sehaba, Nicolas Sabouret, and Vincent Corruble, An Emotional Model for Synthetic Characters with Personality, the second International Conference on Affective Computing and Intelligent Interaction (ACII2007), Lisbon, Portugal, pp 749-750, 2007.

[5] J. Velasquez, Cathexis, A Computational Model for the Generation of Emotions and their Influence in the Behavior of Autonomous Agents, Master's thesis, MIT.1996

[6] Hermann Ebbinghaus, Memory: A Contribution to Experimental Psychology, Translated by A. R. Henry & C. E. Bussenius(1913) Originally published in New York by Teachers College, Columbia University.

[7] P. Ekman, An argument for basic emotions. Cognition and Emotion, 6, 169-200, 1992.

[8] l. Millington and J. Funge, Artificial Intelligence for Games, 2nd ed. Morgan Kaufmann, 2009.

[9] J. M. P. V. Waveren, "The Quake 111 Arena Bot," Master's thesis, University of Technology Delrt, Netherlands, 200 l.

8th January 2017

Aims

1. ~~Research and analyse literature focused on NPC development and coding.~~
2. ~~Look for literature focused on NPC moods and thoughts.~~
3. ~~Look for literature focused on NPC decision making based on player decisions.~~

List of Literature

1. Adaptive Behaviour Control Model of Non Player Character
   http://ieeexplore.ieee.org/document/6527386/
2. Component-based Hierarchical State Machine – A reusable and Flexible Game AI Technology
   http://ieeexplore.ieee.org/document/6030340/

Literature Details

1. The base of fuzzy rules describing the NPC behaviour is forming with the structural adaptation. The structural adaptation (change / add / remove the fuzzy rules) can be viewed as a process of learning of NPC, and the fuzzy rules base as memory of NPC. In order to ensure interactive interaction of player and NPC, learning is carried out in online mode (operational learning) taking into account not only internal, but also external factors.
   At each discrete moment it is not rational to carry out an assessment of videogame enjoyment and respectively the learning of NPS in on-line mode. It is connected with the limited resource of the PC hardware and it can break a continuous behaviour control of NPC. The fuzzy rule base (memory of NPC) is limited because of the limited available memory in computer. We assume that in the memory of NPC can be store not more than M rules. At each time-step kT is activated a deterministic number of rules.
2. Traditionally, computer games developers use a small set of techniques over and over again in the implementation of artificial intelligence (AI) functionalities in video games, especially Finite State Machine (FSM) [1] [2] is the most frequently used one. In video games, FSMs are typically used to model the behaviour of computer-controlled game characters, also called non-player characters (NPC), to make NPCs to react to game events seem as intelligent and natural as possible. FSMs consist of a set of states, which represent some kind of actions or behaviours, and a collection of transitions for each state, which specify NPC's reactions to game events.
   In this paper, we propose a new technique called Component-Based Hierarchical State Machine (CBHSM), which introduces software component technique to the implementation of hierarchical state machine. This technique overcomes the limitations of Object Oriented Hierarchical State Machine (OOHSM) and has three significant advantages:

–Compile Time Composability: At compile time, game programmers can create a new high-level and complex state by composing other prefabricated and simple states and transitions as building-blocks (whose source codes keep untouched).
–Design Time Configurability: CBHSMs are no longer completely fixed at compile time by programmers, and game designers have a chance to configure them at design time according to the game's high-level design. This feature decouples game designing from game AI programming, and a minor change of a game's design no longer needs a recompile.
–Run Time Flexibility: At run time, CBHSMs can be reconfigured as needed. This feature frees CBHSMs from fixed hierarchical structure and greatly improves their flexibility and adaptability to the changing game environment.
The software technique of Inheritance reuse utilized in OOHSM, is often called 'white-box reuse' and generally considered to be defective [3]. Over class inheritance, object composition, another common technique for reuse, is favoured by 'Gang of Four', who advocate that it's the best practice to reuse the functionalities of a class by assembling or composing its object. Figure 1 illustrates an ideal CBHSM making use of object composition technique.

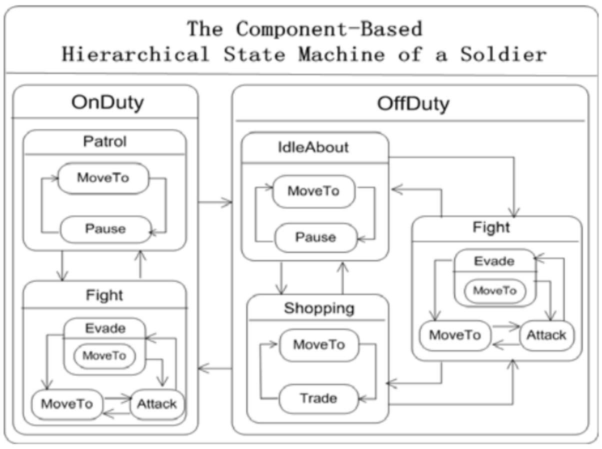


Fig. 1. An Example of CBHSM

In Figure 1, the composite states, such as OnDuty, Patrol, Fight, Evade, OffDuty, IdleAbout and Shopping etc., act as states containers. Some other states, such as MoveTo, Pause and Attack, can be composed as building-blocks into different contexts (up level composite states). The object composition technique obviates the need to create a new class by inheriting from a state to reuse it. A composite state can only 'employ' an instance of the state and delegate operations to the 'employee' when receiving a request. We call this ideal HSM 'Component-Based'.
In a game system, game objects, such as, avatar, NPCs, trees, missiles, etc., interact to each other by game events. A game object only acts when an event is produced and sent to it. When a game object receives an event, a transition rule may be satisfied and then be triggered to change its current behaviour (state). In most cases however, game events are simply synchronous events (method calls) which is most straightforward to use. But a synchronous event has a disadvantage that it couples the event senders and the event consumers. In the case of FSM, synchronous event causes a state to be dependent on its context and then tends to be un-reusable.
In FSMs, each state has a collection of transitions, which specify the NPC's reactions to various game events. It is a common practice to hardcode the logic of transitions into the corresponding states, often taking the form of a series of statements of 'if-else', which causes the coupling between a state and its transitions. However, a state interweaved with

hardcoded transitions tends to be un-reusable, because the same state in different contexts may have different reactions to the same event.
To decouple a state and its context, the state should receive asynchronous events. Since most of conventional game engines don't support asynchronous game events directly, we must establish an Asynchronous Event-Driven System (AEDS) based on a conventional game engine first.
The paper goes on to explain the process of establishing the AEDS and the implementation of CBHSM system. This is will be skipped for now as this more advanced version of the finite state machine system is not required for the project at the current time. As always, as the develop goes on the idea of implementing this will be thought about and will be returned to if required.

Literature Comments

1. The main amount of the paper is not relevant for the project, however, the idea of using fuzzy rules for the NPC behaviour and decision making is something that can be considered. Using an algorithm based upon memory for each character can be used to affect what jobs they choose to perform as well as their moods, emotions and thoughts. The memory of each character is closely linked with its thoughts, since your thoughts are what help you to remember certain things, for example, if a character is talking to someone and is engaged in the conversation because they find it interesting or important, they may not have their attention on something else that they need to be concentrating on, and so their memory of the important thing would be lacking than if they weren't having the conversation.
2. The main idea of this paper is that finite-state machines are very simple, which can be good, but it means that all the states need to be worked out and coded before run-time. This means that the whole system needs to be planned before any coding can begin since all the states should link with each other. This paper suggests a system that allows a more complicated implementation to be used to create a more intelligent way of making states on the fly. This is an interesting idea, and one that will be taken into account. However, due to the limited amount of jobs the employees will have in the game, the simpler version should be good enough to allow the employees to be as smart as they can be.

References Used in the works

[1] Thomas, Andy Hunt, 'State Machines,' IEEE Software, vol. 19, no. 6, pp. 10-12, Nov./Dec. 2002.

[2] Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P., 2006. Modelling software with Finite State Machine: A practical approach, United States: Taylor Francis Group.

[3] Alan Snyder. Encapsulation and inheritance in object-oriented languages. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 38-45, Portland, OR, November 1986. ACM Press.

16th January 2017

Aims

1. Look for literature on employee specific AI creation techniques.

List of Literature

1. Game Coding Complete. McShaffry, M. and Graham, D., 2013. *Game Coding Complete Fourth Edition*. 4. USA. Course Technology

<u>Literature Details</u>

1.

- In the early days of game programming. AI was often completely hard coded.
- Let's look at a trivial example: that of a light timer. The update function checks to see if the time passed is within the start and end times and turns on the lights if necessary. It also turns them off when outside of that time zone. Since time is cyclical, the function takes into account whether or not the end time has wrapped around back to the beginning.
- The next step is randomization. The easiest implementation would be to instantiate the LightTimer Class with random start times and end times and then do it again every 24 hours or so. This would certainly solve the problem of being deterministic, but it falls on the exact opposite end of the spectrum.
- A better solution is to create a random deviation from the start and end times.
- Weighted randoms are a close cousin to the distribution curve. While a distribution curve is essentially an analogue device, weighted randoms are more 'digital'. This idea is that for some number of possible decisions, each of those decisions is given a weight.
- The weights are all added up, and a random number is generated from zero up to the sum of all weights. This determines which action is chosen. This is a very easy way to create potentially complex decisions.
  <u>Finite State Machines</u>
- A finite state machine is a construct that can exist in any number of finite states.
- A video game itself is often managed as a state machine, where the title screen is one state, playing the game is another state, and the options menu may be a third state, and so on.
- In this example the enemies are all teapots. Every teapot is given a state machine instance, which contains a back-reference to the teapot itself, a current state, and a brain. The current state is the state the teapot is in right now. The brain is an object containing a Think() function that returns the best state for the teapot. the SetState() function checks to see if the current state is nil or if the new state is not the same as the current state. If either condition is true, it sets the new state. We need to check to make sure the state s are different because choosing the same state rally means choosing to continue doing what the teapot is doing.
- ChooseBestState() tells the state machine to find the best state for the given situation. This is the AI update function and is called periodically by a script process. If the teapot has a brain, it calls the Think() function on that brain to find the best state and attempts to set it. The Update() function runs the current state ad is called every frame by another script process. The _InternalSetState() function instantiates the state object and calls its Init() function.
- States are typically self-contained with rules defining how the state machine transitions from one state to another.
- One of the big advantages of state machines is that states can often be reused among many different creatures.
- Let's say we want to make a guard that patrols an area until the player gets within a certain radius and then attack. If the player gets too far away, he resumes his patrol. If his health gets too low during the fight, he runs away.

- To do this with a state machine, you need three states: one that defines the pacing behaviour, one for the attack behaviour, and the third for the running away behaviour. These states are connected by transitional logic.
- States can have any number of implementations but are typically implemented with an abstract base class that defines an update function. Each state implements this update function to provide the appropriate behaviour for that state.
  The Think() function subtracts the player's position from the teapot's position. If it gets below a point, the player is considered close. The hit points are then checked. If they are low, then the teapot runs away, if they aren't low he will attack. If the player isn't close he will patrol.
- We can make the transitional logic generic too. If there is a land mine that explodes if the player gets too close, then the same logic can be used.
- Each of these pieces of transitional logic can be encapsulated into generic functions, and each state can have a list of one or more of these functions paired with a target state.
- Platformer games tend to have reactive AI. They will only change state once a condition is met. Other AIs are active, meaning they will constantly seek the best possible action to maximise their happiness. A sim from The Sims is an example of active AI.
  <u>Decision Trees</u>
- A decision tree is a simple way of representing decision making. Each nonleaf node in the
- tree is called a decision node, and it represents a single decision with a binary yes/no answer.
- Each leaf node is called an action node, and it represents an action. In our case, this action is a new state.
- Decision nodes have a true node and a false node., which can be either another decision node or an action node.
- A decision is made by starting at the root node and recursively walking down the tree until an action node is reached.
- The diamonds represent decision nodes, while the rounded rectangles represent action nodes.
- Decision trees can easily be shared, and individual nodes can be shared across different trees.
- A decision node has a back reference to the brain, the true node, and the false node. Since this is an abstract class, the Decide() function is defined with the same error patterns as before. It will eventually return the action to perform, which it does by recursively calling the appropriate child. This class also defines functions for adding a true node and false node.
- The action node class inherits from DecisionNode and implements the Decide() function to simply return the action. This ends the recursive chain and causes the action to be sent all the way back up to the initial Decide() call. Not that SetTrueNode() and SetFalseNode() are redefined to kick out errors. Action nodes are leaf nodes by definition, so attempting to add a child is an error.
- After this is some code to explain how the nodes interact. The parameter for 'close' is defined. And then the distance between the two actors are taken away from each other. If the actor is 'close' then the TrueNode's Decide() function is called, and if not then the the falseNode's Decide() function is called.
- The only thing left is the brain itself. This class implements the TeapotBrain class. The Init() function calls a private _BuildDecidionTree() function.

- The Think() function of DecisionTreeBrain simply calls the root nodes Decide() function and returns the results. The function starts the chain of recursion to find the appropriate state to be in.
- Even if you are processing hundreds of nodes, the nature of the tree structure means you can easily make the decision across multiple frames. At each step, you check to see how much time has passed. If the decision is taking too long, you simple save the current node and return. The nest time, the decision-making process can be picked up at the last node. Just be careful with this; the decision already made may no longer be valid. As long as the decision doesn't take more than a couple of frames, this is rarely a problem.

  <u>Fuzzy Logic</u>
- This system works fairly well, but it's not exactly realistic. The value of 'close' is an absolute value and humans don't think in absolutes like that.
- The idea of fuzzy logic is that objects can belong to multiple fuzzy sets by different amounts. In order to assign degrees of membership within fuzzy sets, some translation needs to occur.
- What is the degree of membership in the close and far fuzzy sets? In order to find this out we need to translate the absolute value into these degrees of membership. This is called fuzzification.
- In order to process the date to make a decision, we need to go in reverse, which is called defuzzification.
- The simplest way to fuzzify these types of values is to provide a simple cutoff. These cutoffs means that beyond a certain value you are 100% in one set and 0% in the other, and vice versa for another value. If the case is somewhere in between, then both sets are used and will have a degree of membership for each one equal to a linearly interpolated value between those cutoffs. This can be represented as a percentage between the two values.
- There are other fuzzification methods, or course. You could apply a logarithmic curve or Gaussian curve (aka bell curve). Nothing says that your degree of membership values need to add up to 1, although typically best they do. It makes the math a bit easier.
- Defuzzification is a bit trickier. There is rarely a direct mapping from the degree of membership to a useful value. For example, if the player is behind cover by 0.6 and exposed by 0.4, what is the correct behaviour? We could just generate a random number and choose to throw the grenade 60% of the time. This works for extremely small fuzzy sets, but what happens when we're trying to take into account multiple fuzzy sets.
- If the result you're looking for is a number, a blended approach becomes very useful. You can blend the numbers together, normalize the results, and then apply that as a multiplier.
- For Boolean results, a cutoff is typically determined. If you belong to a set by more than the cutoff value, the Boolean value is true. Otherwise it is false.
- The real power of fuzzy logic comes from being able to write logical sentences:
- *IF (distance < 20 AND health > 1) THEN Attack() END*
- This is a simple logical sentence with an AND. You can also make OR or even NOT one of the values. You can apply these same logical operators to fuzzy logic systems:
- *IF player is close AND I am healthy THEN Attack() END*
- If you want to use this for fuzzy sets, it works within the attack function, which in itself is a fuzzy set. The AI can belong to this action set as well as others.
- *AttackSet = player is close AND I am healthy*
- *RunSet = player is close AND I am hurt*
- You need to redefine AND, OR and NOT for fuzzy sets

A	B	A AND B

0	0	0
0	1	0
1	0	0
1	1	1

- The most common definition of AND for fuzzy sets is: R = min(A,B)
- A and B in this case are the degrees of membership in those sets. Assuming that the degree of membership in both cases is absolute, then this truth table still holds true. With mixed values, the truth of the statement A AND B is essentially equal to the least true member.
- The reverse is said for OR:

A	B	A AND B
0	0	0
0	1	1
1	0	1
1	1	1

- In this case, the degree of truthfulness of the statement A OR B is equal to the truest member: R = max(A,B)
- For example: the player is 0.6 close and 0.4 far, and the enemy is 0.3 healthy and 0.7 hurt. That means that:
  AttackPercentage = 0.3/(0.3+0.6)
  RunPercentage = 0.6/(0.3+0.6)
- These are the chances he will stay and fight and the chance he will run. But you can also set it up so he will do both, he will run 66.7% of the time, and attack 33.3% of the time. So he will run while shooting. Eventually he may get hurt enough that the healthy is low enough that the AttackPercentage is so low that he ignores it and only runs.

<u>Literature Comments</u>

1. Clearly, there is a lot of information in this book. Most of it is appropriate and informative.
   <u>Finite State Machines</u>
   This is a very good way to program the characters in the game. When a worker is idle, it needs to work out what to do next, and these can all be different states. Lesser ranked workers, such as CSAs, will be told what to do by their superiors and so they will not have much choice in what to do next. This makes their state network simple because they do not need to do much thinking for themselves. Higher up employees such as supervisors and managers will need to make a choice on what to do next, and they might end their tasks prematurely if something else happens which causes them to change their priorities, that means all their states will need to link in some way to each other, even if this is a simple stop what you're doing and re-evaluate your task state. However, CSAs could have a passive order from their superiors of something like, make sure the queue on the till is not above this number. This means that they also need to keep checking on their environment and may need to stop their task early to do something else, but they will need to check less things.
   A lot of the states employees will use will be shared, such as manning a till, or facing up. However, some will be exclusive to different workers based on their job role and position. Supervisors will need to handle complaints and till counting, both of these things CSAs won't need to worry about.
   <u>Decision Trees</u>
   These are useful when the employees are choosing their next task, and will be used for the priority system. It is quite easy to see why these will work well with the priority system that is due to be implemented. Decision trees will might also be used when making a decision

involving a customer. For example, if a customer is being rude and demanding a refund the worker dealing with them needs to decide on their course of action; do they allow the refund, or do they deny it? This can also be linked with fuzzy logic due to the fact that lots of factors may influence the decision, for example has the customer complained before? Are they abusing the return policy? Are they being overly rude? Etc.

<u>Fuzzy Logic</u>

Fuzzy logic seems to be something that may not be taken into much use during the project, despite it creating very realistic acting AI. It will probably be used for smaller decisions rather than large task changing decisions. For example, a worker is stacking boxes of cereal on top of each other to create a display, fuzzy logic might work well for the decision of how high to place the boxes. It needs to not be too high or too low but also needs to be stable enough to not tip over.

Emotions and thoughts might be something fuzzy logic is useful for. Due to the very non-absolute thoughts and emotions are, you can feel or think lots of things at once but at different amount, so fuzzy logic might work well there in determining whether to take a break and get a cup of tea, or how fast to move, or to obey the order of a supervisor the worker doesn't like.

Decision trees and finite state machine will probably be more useful than fuzzy logic in creating an overly realistic character, and the fuzzy logic will be in place to add emotion and small amounts of subtle realism to the characters. On a small note, fuzzy logic will probably be used more in the customer AI than the employee AI due to the employee needing to follow orders and having specific tasks to perform at certain times.

31st January 2017

<u>Plans</u>

1. ~~Begin planning employee job list, priority system.~~
2. ~~Create an employee model that inherits from the character model.~~
3. ~~Create Jobs.~~
4. ~~Code the Finite State Machine and employee AI logic.~~

<u>Coding</u>

Changed Character class to be abstract. This is to prevent characters from being created as this needs to be done by creating employees or customers instead. Created Job and separate Job scripts and classes. Each job script inherits from job. This did not work. Backtracking was needed. The separate job scripts were deleted, and instead replaced by a finite state machine within the job class.

The job script has a enum for each job, and another enum for each task within that job. For example, one primaryState is ServeOnTill, and this contain three secondary states; GoTo, Idle, and Use. These secondary states can have been named and will be coded so that they can be used with other primary states. A switch statement was created within the job class with each primary state having a case, and another switch statement inside those for the secondary states. It was realised that this was very long and non-optimal, and also the job class would need access to the employee it is associated with, this was not correct and shouldn't be done. For example, the GoTo state would need to use the SetDestination function of the character class.

Backtracking was then done again, and now the finite state machine is within the employee class, and acts as the employee's 'brain'. The job script still exists but currently only contains basic variable about the job and the enums.

The job has a tile in which employees need to go to, to perform the job. Right now this only works for jobs that require only 1 piece of furniture, such as the checkout, this will need to change when job require more than one furniture, such as WorkStockCage where employees will need a stock cage and a trolley to perform a task for the job.

Added a m_jobTile variable in the furniture class. This is what the job class using for reference when it is assigned a piece of furniture to be a part of the job.

Each job state has different required furniture to perform the job. This is what the employee will use to work out where to go. This currently only works for single furniture jobs, not multi-furniture jobs.

All characters have a Update_Dothink() function which is called once a frame. In the character class, it is empty, but is overridden in the employee class, and in the future the customer class too. For the employees, this function is where the character thinks about what they need to do next. If they don't have a job, they create one, this works for now but in the future CSAs cannot create jobs so this logic will need to be adjusted to include different job titles, such as managers. The employee will then attempt to go to the job's tile, and the JobSecondaryState is set to GoTo. This is so that in the future logic can be added for while the employee is travelling somewhere, such as talking to other characters, or thinking about their mood or thoughts. When they decide to go to the job tile, they need to decide which furniture to go to. If the required piece of furniture is in the world, they will find it, and then decide if they can use it and if they can get to the job tile, if both of those are true, the job tile is then set. Once the employee is at the job tile, they will perform the job. What they actually do now will depend on the Job's Primary State. Another function called Update_DoJob is run once a frame if the Job's Secondary state is set to Use. This allows the employee to think about what to do depending upon the job Primary State.

<u>Created Scripts</u>

1. Employee
2. Job

<u>Purpose of Scripts</u>

1. Employee – This inherits from the Character abstract class. It has a job variable which its AI uses to walk around and perform jobs. It has update functions which contain the Finite State Machine for the decision making.
2. Job – This is a model script which its instances will be created by employees when wanting to create a new job to perform, either for themselves or for lower ranking employees. This contains variables such as til, maxtime to complete the job and a dictionary of all the furniture that may be required during the job. The most important part is that it contains states; primary and secondary states. The primary states represent entire job processes, such as facing up. The secondary states are interchangeable between primary states and represent that smaller tasks for each stage of the primary state, such as going to a job site, or moving stock from a trolley to a shelf. There is no AI logic in the job class, it is all processed in the Employee class. The job class simply keeps the job information in an ordered fashion and provides the employee class with that information when it asks for it.

3rd February 2017

Plans

~~1. Begin thinking about the final questionnaire/ survey that will be used in the Beta testing.~~

General Thoughts/Ideas

Currently it is hard to think about specific questions due to the unknown factor about how complex and intelligent that AI will be at the time of the Beta testing, however, thinking about general questions can be done now and then the questions can be altered later depending upon the state of the AI and the medium in which the AI will be tested.

Currently the plan for testing the AI is creating a scenario using code to create a shop, so the game will begin with a shop, and employees, and a few customers and will allow the testers to get right into the complexity of the AI and quickly observe it without needing to build up the shop and mess around with the game side of the project. This allows the game mechanics to not be worked on, which is fine due to that being outside the scope of the project, and instead more focus is set onto the AI. Lots of the Game mechanics will not be implemented by the time to testing commences, such as building the shop using realistic means, keeping up to date with costs of running the shop and hiring and firing employees etc. Things such as shop opening and closing times will not be able to be changed, as well as times to do certain tasks, for example a few hours before closing, the priority for facing up needs to increase, this exact time will not be able to be changed.

Main areas for testing:

- Job priority decisions – Are the decisions the employees making during certain situations realistic, the NSAs should keep working on their given tasks unless the queue gets very large, or a supervisor orders them to do something else. Supervisors should be creating jobs based upon the current situation and performing them themselves, or intelligently giving them to NSAs if it makes sense to.
- Communication – All the characters should be communicating to each other if they are in close proximity, like in real life. They should all have relationships with one another, and this will affect what they talk about, how long they talk for and whether they talk at all. A NSA that doesn't like a supervisor may not choose to talk to the supervisor at all, but if the supervisor engaging in conversation, the NSA should respond with an appropriate attitude due to them talking to a superior, the supervisor should also learn from the NSA's response and adjust its relationship understanding with the NSA, and act upon it based on its traits. If its traits mean that it ignores the relationship and barks demands anyway, then the supervisor should be all means ignore the negative relationship they share with the NSA. On the opposite side, if the supervisor wants to be liked, then they should use that trait to decide whether to give a NSA a job, or if a NSA is slacking, whether they should approach the NSA and comment on the lack of working. This is all realistic behaviour and should be implemented into the AI coding.
- Traits, Thoughts, and Moods – The characters should all have certain traits, and moods. They their interactions with the world, and other characters should create thoughts which then, based on their traits, affect their mood. This should be looked at and commented upon by the testers. If an employee has a trait which causing them to tire easily and affects their work rate greatly, then long periods of uninteresting work should cause their mood to become negative, and as a result their work rate should decrease. Another character's trait

may be that they are always trying to impress the boss and wants a promotion, and so their poor mood may not affect their work rate very much, which is realistic.

- **IF LINE OF SIGHT IS IMPLIMENTED** – Line of sight – Testers should look at the characters' lines of sight and comment on them. Are they long enough? Does it seem realistic? Do the characters react appropriately to seeing something they didn't previously see, or do they correctly not react to certain things because their line of sight blocks their view from the things that should have caused a reaction.

Things that shouldn't be a part of the testing and should be ignored by the testers:

- Game balancing – costs of certain things such as employee wages, cost of buying stock and selling stock etc.
  - However, things such as work rates, movement speeds, how much the environment affects characters' thoughts and moods are all a part of the AI and SHOULD be commented upon.
- Graphics and sound – No effort will be put into the graphics or the sound, UI may look boring, characters and furniture may look simple. The lack of sound, or the dullness of the sound is not a concern. This is all fine and should not affect the testing results, due to the AI and code being solely tested.

5th February 2017

Plans

1. ~~Continue coding the finite state machine and the employee AI logic.~~
2. ~~Add Stock to the game.~~

Coding

In world, a two new variables are StockPrototypes and StockInWorld. All the stock used in the game will have a default copy saved in the StockPrototype variable, these are all created at the start of the World creation. The StockInWorld represents all of the stock that the shop can currently sell, and each and every piece of stock in this dictionary will be somewhere in the shop, either on a piece of furniture or on a character.

The world can now place a piece of furniture like before, but with specified stock on it already. This function checks all weight restrictions are part of the placement, and will log an error if the stock exceeds the furniture's maximum weight. This should never occur in this project, except for coding errors, due to all added stock being hardcoded into the game as part of the AI testing.

The furniture class has two new functions: TryAddStock and TryGiveStock. TryAddStock checks the weight restrictions and if they are allowed, the stock is added to the m_stock variable, and if not it returns false. The TryGiveStock checks to see if the wanted stock is in the m_stock variable, if it is, it returns true and updates the m_usedWeight variable, if it's not in the list, it returns false.

Like furniture, characters now have TryTakeStock and TryGiveStock functions. The TryGiveStock function checks to see if the wanted stock, by name, is in the character's possession, if it isn't it returns null, if it is it returns the first stock of the stock's name and removes it from the list of that stock name, if the list is empty, it removes the list from the dictionary. The TryTakeStock function checks to see if the stock is too heavy to be taken, if it is, the function returns null, if it isn't too heavy, the furniture's TryGiveStock function is called. If that is true, the stock is added to the m_stock variable and the weight is updated, and the function returns true.

Added button to unity which creates the world. This was done so that all the Start() functions would run before any attempt to create the world was made. This is to avoid any potential issues in the future, and is closer to the final version of the menu system.

<u>Created Scripts</u>

1. Stock

<u>Purpose of Scripts</u>

1. Stock is a model class that represents all the stock in the game. When stock is created it either needs to be linked with a piece of furniture or a character. Like furniture, all the stock prototypes are added at the start of the game, so that their default values such as name, weight, cost etc. are set. The only things that may change for each individual piece of stock are things such as sell-by-date, and maybe quality if that is implemented.

<u>Found Problem</u>

A problem that was found was that the employee would pick up stock from the till, and then scan it, but would then put it right back down into the same till. This makes sense in real life, the employee would put it on the other side of the till, but for this game it doesn't make sense. There are two ways of solving this. The first way is to add a variable onto every piece of stock called m_sold, and this is a bool that flags if the stock has already been scanned out. Then the employee would check to see if the item has been scanned before picking up the item. If all the stock on the till has been scanned, then the transaction is over and the customer pays. Another way of solving the issue is to have a different piece of furniture next to the till which the employee puts the stock onto. This requires more unrealistic work however as in real life there is seldom such a piece of furniture.

9th February 2017

<u>Plans</u>

1. ~~Continue coding the finite state machine and the employee AI logic.~~
2. ~~Add selection to the game along with corresponding UI.~~

<u>Coding</u>

MouseController name was changed to InputController, this was done because it now processes keyboard inputs as well as mouse inputs. A new function was added which processing the keyboard inputs. At the moment the only button is does anything with is the Escape key. Pressing the escape key will set the mouse mode to null, which causing it to turn into the select mode. Clicking on a tile with a piece of furniture while the mouse mode is null will cause the tile to be selected with the correct information coming up on the screen. This will be used for testing so that a furniture's stock can be seen.

<u>Created Scripts</u>

1. SelectDisplay

<u>Purpose of Created Scripts</u>

1. This is very similar to the TileInspector Script. Every frame it checks to see if the select display is active. If it is active then it updates the text to show the name of the furniture that is selected and who is using it, if anyone. It also does the same for the stock display, if it is active then it displays all the stock on the furniture and whether it has been scanned or not. This has been implemented so that testing can be made easier as debug mode does not need to be used as often.

10th February 2017

Plans

1. ~~Add rest of furniture that will be required to successfully implement and test the employee AI.~~

Work Done

Created sprites for:

- Trolley
- Backshelf
- Frontshelf
- Fridge
- BigFridge
- Freezer
- BigFreezer

Coding

The above furniture has been added to the game with their own sizes and weight restrictions.

Instead of manually adding the furniture to the world, the input controller will now add furniture to different places in the world to create a standard shop. This will be used to test the AI so far and allows testing to be done without needing to re-create the shop environment each time. This environment will likely not be the one used in the final testing phase, but will do for now, until it can be refined to show off certain AI behaviour.

Created a 'first draft' of an environment. Cannot be fully completed until furniture rotation has been implemented.

Found a bug with the UpdateNeighbours code which caused the checkout to return to its main tile instead of moving 1 to the left, due to its increased width size. This was fixed by adding a flag, if the tile due to be moved doesn't link to neighbours, such as the checkout, then it doesn't need to be updated, so it does no long update, and thus its position doesn't reset to the incorrect starting location.

12th February 2017

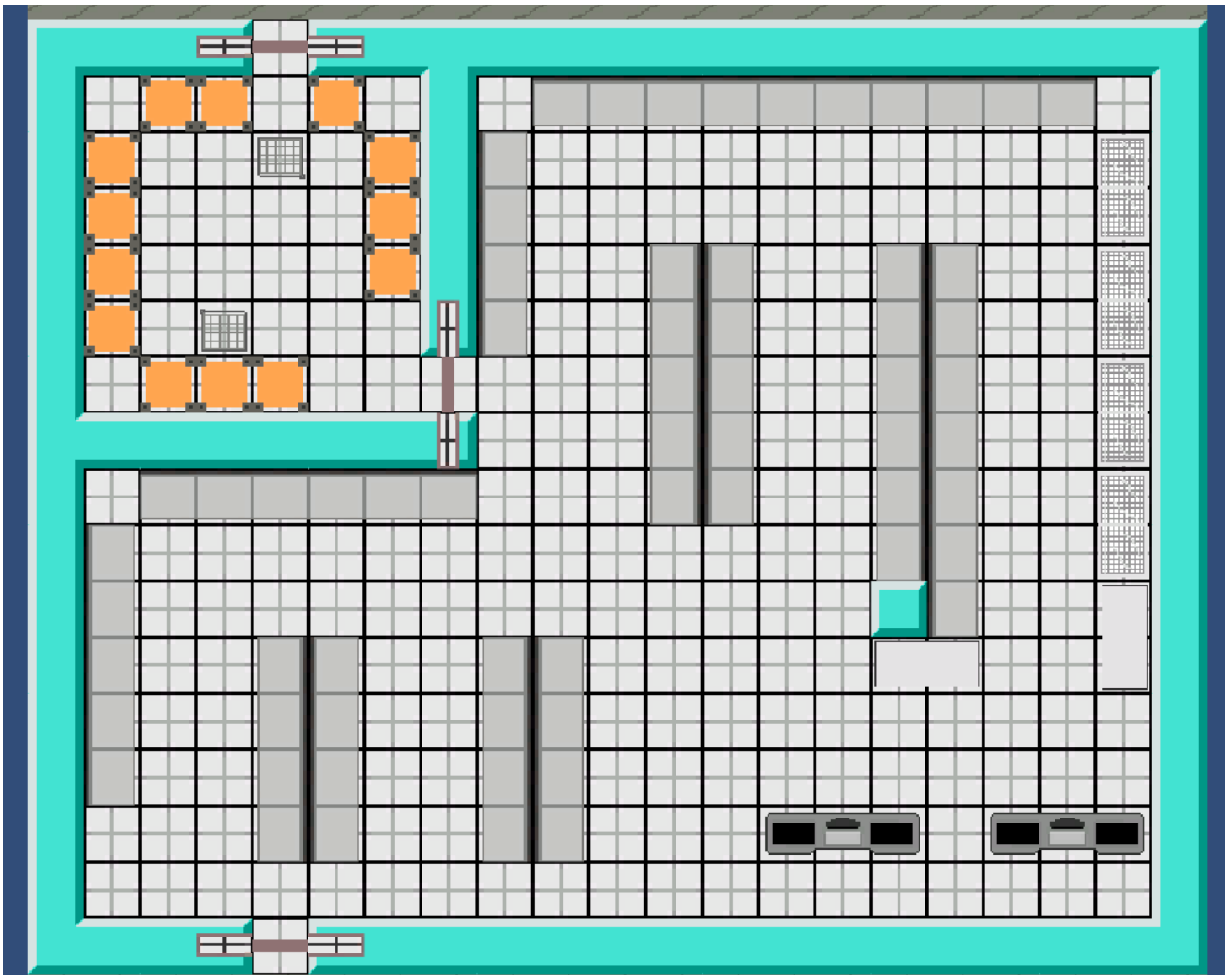
Plans

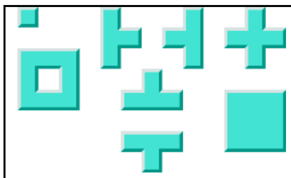
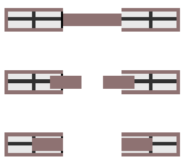
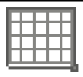
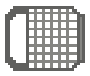
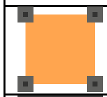
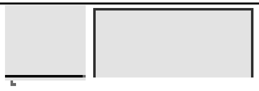
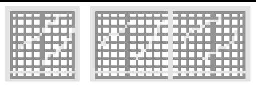
1. ~~Add rotation into the furniture code.~~

Coding

Added a function in Furniture to rotate the furniture. If the rotation is 2 or 4, east of west, then the furniture's height and width gets swapped. For example, the till is 3x1 (widthxheight), if its rotated to the east then its height is 1x3(widthxheight). Then the FurnitureSpriteController also checks the rotation when creating the GameObject, if the rotation is at 2, then it gets rotated by 90 degrees anti-clockwise, if its 3 then by 180, if its 4 then by 270. The final step is setting the furniture's job tile. This by default is always in the centre tile -1 in the y. So if the checkout is taking up tiles (20,21), (21,21), and (22,21), then the furniture's job tile will be at (21,20). If it is rotated then it simply adds 1 to the x or y, or takes away one from the x or y depending upon the rotation.

Completed the rest of the shop layout code. Work can now begin on the rest of the finite state machine employee AI coding. Below is a screenshot of the current shop setup. Also below are the different furniture that is currently in the game along with their sprites.



	Walls – the walls are special. They actually change depending upon if the tiles around them also contain walls. Here is a sprites that displays all of the possible wall sprites used for individual wall tiles.
	Door – The door is also unique in that it has three different sprites, and it is the only furniture that its sprite is larger than the furniture. The furniture in game is 1x1, but the sprite is 3x1 due to the ‘animation’ behaviour it has when a character walks through the tile.
	Stockcage – This will become one of the two movable pieces of furniture in the game.
	Checkout
	Trolley – This will become one of the two movable pieces of furniture in the game.
	BackShelf – Simple shelf for storing stock not on the shop floor.
	FrontShelf – Simple shelf for storing stock on the shop floor.
	Fridge and BigFridge – These two are the same except the BigFridge is double the width. These are used for storing stock that needs to be kept refrigerated.
	Freezer and BigFreezer – These two are the same except the BigFreezer is double the width. These are used for storing stock that needs to be kept frozen.

13th February 2017

Plans

1. ~~Add new stock to the game and add them to different furniture around the shop.~~

Coding

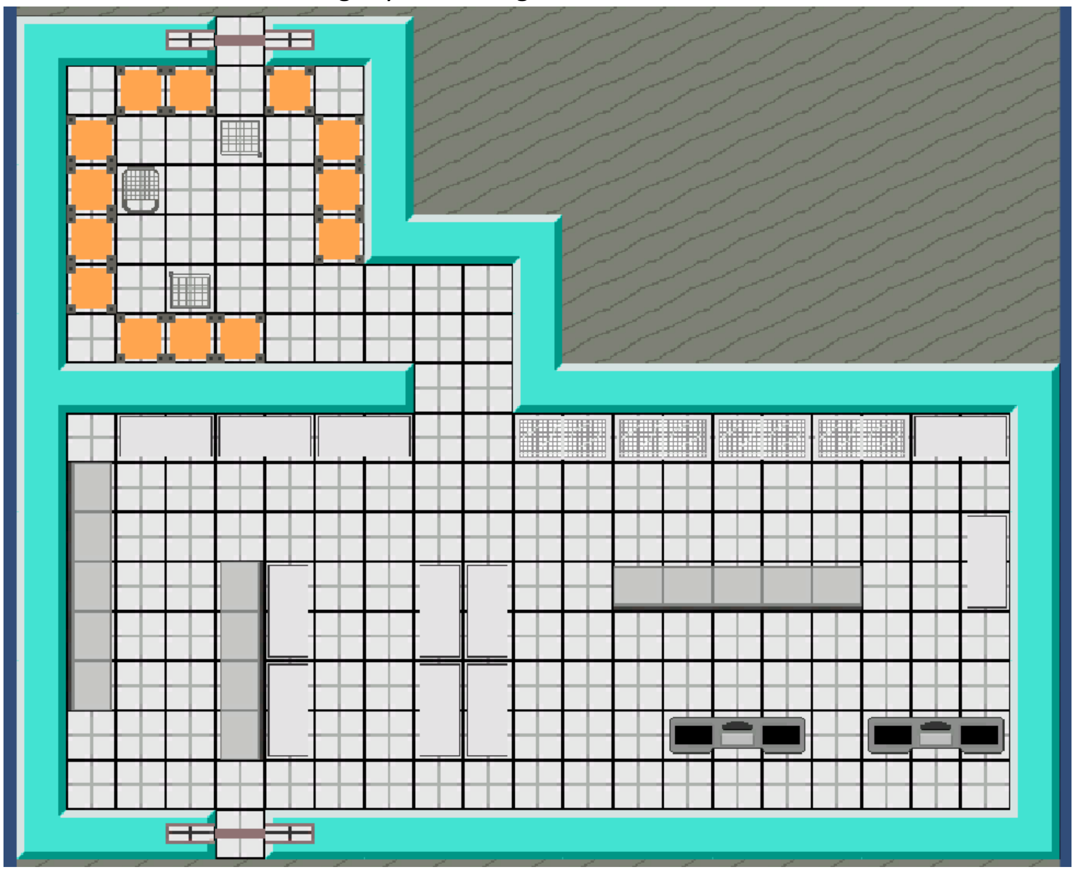
Adding the stock prototypes in the way presented in this project is not the best way. For the full game the stock prototypes would be read from a text file and added with a while loop. In this project, they have all been added one by one in the CreateStockPrototypes class.

Added all stock to the game that is needed throughout the project. A problem has been found that when spawning furniture with stock, the stock is added to the furniture prototype, not to the furniture instance, this causes all furniture in the game of that type to receive the stock. It is not currently clear on why this occurs. This can either be solved, or the stock can be added to the furniture after the furniture has been placed which should solve the problem. Work will be done on fixed the initial problem first, before finding a way around it without solving it.

The above problem was solved. The 'issue' still remains that the actual instance of the prototype is being passed, by this was intended at the beginning, the problem is that the values where not designed to be altered by grabbing the prototype's values first. The problem was solved by resetting the stock dictionary once the information was grabbed. This is kind of a cheat because if we ever want to grab a piece of furniture, with its stock, and put it somewhere else then the stock will reset. This can be solved by created yet another constructor that doesn't reset the stock, but this may not be needed. The only place that this may come into play is when movable furniture logic gets added to the game.

Added a scrolling list of stock to the UI, this has a scroll bar that can be used, and the information displayed is the stock's name, and price. In the future, similar stock items would like to be grouped and the number of the items is displayed, then clicking on the item will open another box with more information on such as weight, ideal, temperature, sell-by-date etc.

The shop layout has been altered. It was discovered that the initial shop layout was too big for the project and was unnecessary, and there are not currently enough stock items in the game to demand such a large store. The layout below is the new environment, not much has change d except there are now less shelves and slightly more fridges.



14th February 2017

Plans

1. Add movable furniture logic.

Coding

Trolleys and Stockcages can now be moved. There are still problems/bugs which need to be sorted. Furniture now has 2 more varibles; bool m_movable, and bool m_moving. These are used with the furniture parameters and furniture actions to move furniture in a similar way to character movement, except the character moving the furniture determines where it will go. More code will need to be added, right now the furniture will get moved only if it is in the way, but this will need to change so that characters can move furniture freely to certain tiles in an efficient way.

15th February 2017

Plans

~~1. Improve the movable furniture logic.~~

Coding

Right now a character can move a piece of movable furniture if it is in the way of their path. However, there are a number of problems. Firstly, the pull back code is not as it should be, and secondly, the characters cannot currently move a piece of movable furniture directly to a set tile. There is improved code that allows characters to move a piece of movable furniture if is in the way, then they can work out what is the closest tile that they do not need to pass through to get to their final destination, and then move it to that tile, then carry on to their destination. The move furniture to certain tile logic will be used in conjunction with the employee's, and later the customer's, thought logic.

More code has been added to allow more realistic furniture movement if it is blocking a character's path. A recursive function is called, which will check to see if where the furniture will end up will continue to block the character to their path. If it does continue to block the character it will keep simulating where the character and furniture will go after the furniture is in that spot, this will keep repeating until a tile has been found that doesn't block the character's movement to their final destination, or if the character ends up not being able to leave from their spot. If the function returns null, it means eventually, if the character moves the furniture to the initial tile it was going to, then they will end up trapped, if it doesn't return null, then it will return the tile in which the character was going to move the furniture to. A check is done and if the tile shouldn't be used, it is added to the list of invalid tiles that are used to determine if a tile can be moved to. This code creates a very realistic bit of AI and means that character's wont block themselves by moving furniture again and again to an eventual invalid location.

Another piece of code that was improved is the pulling and pushing code. The character will now correctly determine whether will they be pushing or pulling a piece of furniture, this is important because it will change where their final tile position is after moving a piece of furniture.

Code has now been added that allows a character to move a certain piece of furniture to a certain location. If they are not already at the side of the piece of furniture they will set the nearest side as their destination, then once they are there, they will move the furniture to where they want to take it.

Created Scripts

1. FindNearestFreeTile

Purpose of Created Scripts

1. FindNearestFreeTile – This class using the breadth first search algorithm to scan the tiles around the given tile and returns the first tile it finds that does not contain a furniture or is part of the path that the character uses to get to their destinati on. This was created for use with the furniture movement logic. If a piece of movable furniture is in the way, it needs to be moved to the nearest tile that will no longer block the character.

17th February 2017

Plans

1. ~~Continue coding finite state machine and employee AI logic~~

Coding

Found some problems with the furniture movement code, which was very difficult to debug, due to needing to go line by line through the program to see when certain variables get set. The problems included the character moving the required furniture due to it thinking it will be in the way, instead of it just moving it like it needs to. This was resolved with a variable which sets the required furniture, so that if that furniture is the one in the way, it gets ignored, becau se it's going to be moved by the character later on in the frame anyway. Another problem was that checking neighbouring tiles for the required furniture wasn't including diagonal ones, it now does which is correct.

Employees can now find a free trolley if they need it to do their job. If they are not next to it they will go to it, once there, they will move the trolley to the nearest free tile that is neighbouring the job's tile, and the job's furniture's tile. This means they can access the job's furniture and the trolley at the same time by being adjacent to both.

When a job's primary state is WorkStockcage, the employee will try and find a free trolley, if they find one, they will move to it, then take it to the stockcage they need to work. When they are in the job tile and the trolley is adjacent to them, they will attempt to move stock from the stockcage to the trolley. When the trolley is full the primary state is set to WorkBackStock with their trolley being the first task. They will search through the stock and then search the front shelves for stock of the same type. If they find a front shelf they can use they will move to it.

20th February 2017

Plans

1. ~~Continue coding finite state machine and employee AI logic~~

Coding

Logic has improved considerably. Previous untidy, hole-filled code has been tidied up and simplified. Improved code on destination designations. Bulk of the work was tidying up and fixing leaks in code and errors only just found. Added comments for some complicated functions. Improved function which searches for a furniture with the same stock as on the trolley we are using.

Added empty trolley code, which runs when we are next to a furniture we need to put stock into. Loops through all the stock in the furniture, and on the trolley, if they match, the stock is moved. If there is no stock in the furniture, the stock in the trolley gets added anyway; this is used when other furniture already containing the stock gets filled. In the future logic needs to be added that runs when all the empty furniture is also filled, in real life this is when employees would begin to create 'overflow' stock cages, these are for stock that has been worked, but the warehouse shelves are already full. It is impossible for the overflow stock cages to all get full, since the stock arrived on them in the first place.

Removed complicated tile neighbouring code when finding a suitable place for the character and furniture to go when using a piece of furniture, now the trolley simply 'follows' the character. At the moment this is too simple, since the character in real life would push the trolley, but for now it works and allows the rest of the AI logic to get completed.

22nd February 2017

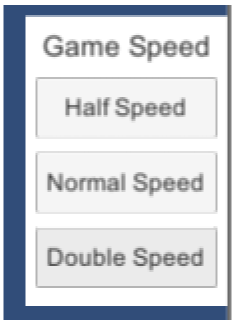
Plans

~~1. Continue coding finite state machine and employee AI logic~~

Coding

Added comments to all the code. Function and variable summary comments added. Fixed bug where if a job's furniture's job tile has furniture the job's tile doesn't get set, which is correct, but this would also be the case if the furniture occupying the space was a movable, such as the trolley being used for the job. Now if there is no furniture, or if the furniture is movable, that is fine.

Added 3 new buttons; half speed, normal speed, and double speed. These control a new variable, gameSpeed. The gameSpeed is multiplied by deltaTime and the result is put into the Update functions of furniture and characters. The gameSpeed will also need to be plugged into timers when they are implemented.



When trying to code the WorkBackStock code, a problem was found where the employee does not know whether it is meant to be filling up the trolley with back stock, or emptying it with front stock. A new state was added to Job which makes the destination and all the WorkBackStock and WorkStockcage code has been altered accordingly.

At this stage, employees will correctly try to empty all stockcages, and once they have all been worked, they will move on to stock on Back Shelves that haven't been worked. After this, the next step is for the employees to start facing up the shop.

A new variable has been added to furniture; m_facedUpPerc and this represents the percentage amount that that furniture has been faced up. As stock is taken from the furniture the percentage will decrease depending upon the stock's weight.

Two new lists have been created in world; m_frontFurniture and m_backFurniture. These represent all furniture in the world that employees should use to do their jobs. frontFurniture includes FrontShelfs, Fridges, BigFridges, Freezers, BigFreezers. backFurniture includes backShelfs. This will be used when working stock, and when facing up. This also means that shelfs, freezers and fridges can be classed as backShelfs by adding them to the backFurniture list, and removed from frontFurniture list if it contains the furniture.

Employees will now face up. They will go through each furniture in the frontFurniture list and go to them. If they have been worked, they move on, if not they will work it. When it's at 100% the furniture is set to worked = true and they again move on.

Timers will now be implemented. When moving stock to and from furniture and when facing up, it will take a certain amount of time. This will be determined by the stock's weight. This will come with its own difficulties because now moving stock will be done over multiple frames.

After implementing the timer code, problems have arisen. Some parts of the code only allow stock to be picked up once per frame, such as the emptying of the stockcage code, this is good. This very simply slotted into the timer code perfectly and now it takes the correct amount of time to pick up each item. But the code to empty the trolley is wrong. That code loops through all the stock and moves it in one frame, this means the timer has not got enough time to activate before all the movement is done. Changes needed to be made so that the characters try move stock from the trolley only once per frame, instead of looping through everyone item in a single frame.

The FaceUp function now takes in a stock item and faces up that single stock. An issue was found where when the new m_faceUp bool would change, it would change every stock of the same name. After looking into it, the problem was found. Firstly, the m_faceUp variable was not getting copied in the Clone function, and secondly, the clone function was not being called at all. When stock was getting added to furniture, the exact instances of the stock prototypes were being used, so only 1 of every stock was getting created and added in different places, so changes one of these references of course changed the original which then changed all the other references. When adding new stock to furniture, the prototype's Clone function is now being called. This creates a brand-new instance of the stock with the exact variables from the prototype, but because these are all separate instances, they can be changed individually, which was supposed to happen in the first place. This is the first time its mattered because before this, the stock's variables were not being changed, just used for calculations etc.

6th March 2017

<u>Plans</u>

1. ~~Think about how the AI looks and think about ways to improve it more.~~
2. ~~Begin coding employee interactions and job giving.~~

<u>Work</u>

At the moment, all employees know everything about the world, and they can all create their own jobs. A hierarchy needs to be created of job roles. The order of command goes: Manager, Assistant Manager, Supervisor, Customer Service Assistant.

World now contains a reference to an employee; this employee is the employee in charge of the shop. If it is not this employee that wants a job, the employee that wants to job needs to seek out the employee in charge and receive a job from them. This is the first account of character interaction.

New Interaction function have been created. When two characters are next to each other, one of them can choose to request interaction with the other, this returns a bool. When a character receives an interaction request, they need to decide whether to accept it, this also returns a bool. When both these are true, the interaction occurs and both the character's interaction variables are set to each other. Some interactions can be done while walking, and so if this is the case, the character with more authority continues to walk to their destination and the other character needs to follow them.

There have been issues in the past where if every frame a character's destination gets set, there is some visual glitching. This has now been fixed. A flag was added to all characters which is used to reset the character's variables at the beginning of the frame of true. But if the dest tile doesn't change even though SetDestination is being called, the reset flag doesn't get set to true, and so the variables don't reset. This means if a character is trying to pathfind to a moving target, such as another character, they can do so, and update their destination without the glitching.

The world has a List which acts as the authority system. If a manager is added to the world, he gets inserted into the 0th index. The Assistant Manager gets inserted into the 0th index if the 0th index isn't the Manager, if it is, the Assistant Manager gets inserted into the 1st index. IF the employee getting added to the world is a supervisor, it will be inserted into the highest index that isn't a supervisor, assistant manager, or manager. This means supervisor that get added to the world after other supervisors will have a higher authority index, which is less authority. All CSAs are added to the end. The newest CSA to enter the world will have the highest authority index, which is the least amount of authority. These can be tweaked later, for example perhaps all supervisors have the same, and all CSAs have the same authority, but this cannot be done with a list.

When an interaction begins, a timer is started. At the beginning of DoThink(), if an interaction is taking place, the timer increases by deltaTime, the employee with the less authority will move to the employee with the more authority, if they can move for the ineraction. If one cannot move, then they both cannot move so the problem of one moving and one not will not occur.

8th March 2017

Plans

1. ~~Begin customer coding.~~
2. ~~Add customer shopping lists.~~
3. ~~Add customer searching and item collecting.~~

Work

The customer's logic is a bit different than the employee's logic. At this point, a customer doesn't know which furniture contains what stock. They need to enter the shop and then find the nearest furniture that can have stock, then go and look in it for the things they want, then repeat until they find all the things they want. This means that the line of sight logic may need to be implemented now. Instead, I could have the customers just go to each furniture with stock, so that the base customer logic can be coded and then can be altered later to create a more realistic AI.

Different shopping lists will be created, and when a customer appears in the world, they will randomly choose a shopping list to use. This can be altered later to create a more realistic AI.

In a similar way to the employee AI, the customer AI will need some kind of finite state machine so that the program can keep track of whether the customer is moving to a tile, searching a piece of furniture for stock, waiting at the till to be served etc. In the employee AI, these states were kept within the job class., since the customer does not have a job class, this will need to be done in a different way. If the customer was coding exactly like the employees, they would only have one primary state; find all their items on their shopping list. This make the coding of the customers much simpler. The would only have different secondary states, this means that these will be bumped up to be primary states.

Coding

Customers can now be spawned into the world. They will randomly pick from all the available shopping lists, which are stored in the world class. They will then go to each front furniture and search it for items on their shopping list. The searching is currently done instantly but the time it takes will eventually be inversely proportional to the item's size; for example, if something is smaller, it will take longer to find when searching. If an item is found, it will take a certain amount of time to pick up, and that length of time is proportional to the item's weight, this works in the same way as when employee pick up stock.

If a customer's list is full, they will next be coded to head to the area to be designated as queue, and if they are done searching for their items but have not found them all they will need to decide to either ignore the fact they couldn't find it, or try and find an employee to help them look.

Created Scripts

1. Customer

Purpose of Created Scripts

1. This inherits from the Character abstract class like the employee class does. The customer character will enter the shop, search each of the shop's front furniture and add items to their basket if they want them. If their collect all the items on their list, they go and queue at the tills.

9th March 2017

Plans

1. Continue customer code.
2. ~~Add queueing~~
3. Add customer/employee interactions when customer cannot find items.

Work

The customer should head towards the tills and then figure out how to join the queue once they see the queue, if there is one. This will be more optimal if the line of sight code is implemented. The customer will floodfill from the till until it reaches a tile marked as the beginning of a queue. If that tile is occupied, they continue until they find another tile marked as beginning of queue. After all beginnings of queues are full of characters, the floodfill will reset and then find standard queue tiles until it finds a queue tile that is unoccupied.

At the moment, the world is not aware of where characters are, and which tiles are occupied by characters. This means that characters can occupy the same tile. This would not be good for queueing as if a character wanted to know if there was another character it would need to cycle through every character to check the tile they are on against that one they want to be on, this is suboptimal, it would be better to check the tile they want to be on to see if any character is on it.

<u>Code</u>

Added a reference in tile to characters, this means that all the tiles know which character is in them. The old mouse mode UI text has been changed to show the name of the character in the hovering tile, if there is one.

When a customer is done with their shopping list, they will floodfill from themselves to find a free queue beginning. If they find one, they go there; if they don't, they will look for normal queue tiles, and then go to the nearest one. Once there, they will wait. Further optimisation needs to be made so that if there is a queue tile closer to the tile, they will go there, this could simple be solved by floodfilling from the tile, rather than from the customer.

Instead of standard queuing tiles and queue beginning tiles, the queue tiles are numbered from 1, which is the beginning. This means the customer can move up the queue if the queue tile next to it is unoccupied and its number is smaller than this one. Now, the customer will simply find the nearest queue tile from where they are, once they get there, they will move further and further up the queue until they reach a point that either they have reached the beginning or the next queue tile is occupied.

11th March 2017

<u>Plans</u>

1. ~~Continue customer code.~~
2. ~~Add till interactions between customers and employees.~~
3. ~~Add money to tills.~~

<u>Work</u>

Finished basic customer code. Without line of sight, the code is ok but can be improved to seem more intelligent. Employees are aware of the customer they are serving, if serving one. Employees in charge, and those already at the checkout, should know if more checkout staff are required, there more are require, the global bool is changed and then the next staff that checks will go and assign themselves to the checkout. This should be optimized later where there is an order of precedence of who goes on till, in general higher ranking employees will be behind others in order, unless they choose to go on the checkout.

<u>Code</u>

When a customer reaches the checkout, they will empty their basket onto it. Employees were already coded to pick up and scan items on the checkout. After this the customer then picks up all the items, pays, and leaves.

When stock gets put on the checkout, and an employee is manning it, the employee searches for the front of the till and assigns the customer there to them, so they know who they are serving. If all the stock is taken from the checkout, this must mean the customer took it, and the employee will keep asking the customer if they are done. Once they are, the customer 'gives' the value of the transaction to the employee, and the employee will add it to the checkout's mon ey variable. After this, the employee goes idle, which causes them to check to see if they need to stay at the till, if they don't they will find another job, if they do, they will stay idle until more stock is put onto the checkout. The customer will leave the shop after the transactions is completed.

13th March

<u>Plans</u>

1. ~~Add time and calendar to the world.~~
2. ~~Create customer spawn timings, with spawn rates being changed with time.~~
3. ~~Customer spawn rates.~~

<u>Work</u>

The time and calendar are used to keep track of shop opening hours and when more staff should arrive, as well as busy and quiet times of the day. The world will keep track of minutes and hours. When minutes reach 60, they are set to 0 and the hour increase s by 1, when the hour reaches 24, it resets back to 0 and the next day is started. To begin with, 1 second in real time will equate to 2 minutes in game time, this will cause 1 day in game to equate to 12 minutes in real time.

After implementing 1 second real time equates to 2 minutes in game time, it was found that it wasn't realistic enough, and even though most games have time altercations to keep it entertaining, this game will have exactly 1-1 game time/real time ratio. This increases the realism and creates a better simulation. Players can always increase the game speed to make it 10x as fast if wanted.

Due to an exact 1-1 ratio between in game time and real time, the speed buttons will be changed. The slowest one will be real time, the middle one will be 10x and the highest one will be 20x.

When a customer spawns, a number is picked from 0 to 20, this is the time until the next customer spawns. This largest value is changed depending upon the time of day and day of the week. This can be further refined to be different for special days of the year, but will not be for this project as it is not easily tested.

<u>Code</u>

The world now knows the time, and every second will increase the second int by 1, when that reaches 60 it is reset and the minute int increases by 1, then when that reaches 60 it is reset and the hour int increases by 1, and then when that reaches 24 it resets.

Dates added to game. World is aware of the day of the week, day of the month, month of the year, the year, and the week of the year, and these are all updated when they need to be at midnight. A UI has bene added which updates to the correct values each frame .

Known bug that if a customer's list was not complete, but all furniture had been searched has been fixed. They will now go to the tills anyway. Also, if there were no queue tiles unoccupied, an error would occur, this has been fixed. Now customer will look for the nearest queue tile, and go there; at the moment they all occupy the same tile, but when the code is fixed so that characters will not move to a tile already with a character on it, they will crowd around the tile, just like they would in real life.

Bug found where employees would not correctly change the number of checkouts worked when beginning a checkout job or ending one. Problem was that the employee was creating, and giving themselves, a new checkout job because there were enough customers in the queue to need another checkout opened, but this meant the world thought there were two checkouts being manned, when technically the same employee was manning both. This has been fixed so that if another checkout is required, when an employee checks to see if they should go and man a checkout, they decide not to if they are already manning a checkout.

The issue remains that when an employee changes job, the number of manned checkouts does not drop. This is now fixed, issue occurred because employees were creating brand new jobs, which meant the next for the old job being a checkout job wasn't working. New parameters have been added to the Job constructor so that it is aware of the previous job if there was one, and that has been used to check to see if the value of the number of manned checkout needs to be reduced.

Customer spawn rates have been added, at different times of the day, customers will spawn in at different rates. This was done by picking a random number between 0 and a given value. The value is increased if the spawn rate needs to be reduced, due to it being more likely that a higher number is chosen. Different days, and even months could have different effects on the spawn rates, for example Saturday and Sunday could have an increase in spawn rates due to it being the weekend. This was not implemented, due to its testing being an unnecessary amount of work for this project. If the program goes out for proper full release, this of course will be tested for flaws.

Bug found – Moving movable furniture in the way is broken. Will be fixed next session.

16th March 2017

<u>Plans</u>

1. ~~Fix movable furniture bug.~~
2. ~~Fix bug regarding customers seemingly not putting stock on checkout~~
3. ~~Stop allowing characters to occupy the same tile.~~

<u>Work</u>

After the first 2 customers have been and gone; the next customer seemingly doesn't put their stock onto the checkout. This cannot be confirmed as there is no way in-game to view what a character is holding. To fix this a UI will be created which will allow the user to look at different stats and information about a character if they are clicked on. This means code needs to be added so that character selection is actually possible, at the moment, this is only available for furniture.

When the characters path find, they cannot think of tiles with charcaters in them as obstacles, because by the time they get there, the tile is probably empty. Instead, they will treat those tiles are doors, which means when trying to enter a tile, if a character is there, the tile's enterability is set to SOON. This means that they will wait there for the character to move. This can have problem such as if two character are trying to pass each other in a 1 width corridor, or if the character they are waiting for doesn't move. Perhaps at this point, the character SHOULD path find in a way that takes the static character into account.

Code

The immediate problem was that the character's dest tile was getting reset, which screwed up the pathfinding to the free tile. This was due to the ignore task variable not being used unless the character had a job, which isn't always the case, so the ignore task check was moved to a different line to allow it to trigger every frame. Another issue was found where the moving furniture variable wasn't getting reset when required, for example when an employee abandons a job to serve on the checkout, this caused the employee to bring the trolley with them, or when the employee went back to work, it couldn't find a free trolley as the only trolley which was being used by them had a character assigned to it.

Added new UI that allows the user to click tile that contain a character. In a similar way to the furniture information UI, the character information UI shows the name of the character, the furniture it is currently using, and if there is stock they are carrying. If the character has a basket, it is displayed in the same way that furniture stock is displayed.

After creating the UI, the bug is more clear. If the customer does not find all their items, their currently searched furniture was set to null. This meant that the DoThink() function was returning right at the beginning of the checkout switch statement section. This has been resolved and now if their furniture is null, they search for the checkout if they are at the front of the queue.

The enterability solution described in the work section of this entry worked mostly correctly. Characters are slowing down and waiting for other characters to move if they are in the way. The issue of if two characters are trying to enter each other's tile is one which will occur more often that was originally thought. It will occur whenever two characters are moving into each other, not just in a 1 tile width corridor. A possible solution to this is if the two character talk to each other and find a solution between them. That will be thought about and resolved in the next session.

21st March 2017

Plans

1. ~~Resolve character waiting infinitely bug.~~
2. ~~Add character traits and interactions.~~

Work

Work on characters' thoughts and moods will be done before the character waiting bug because a better interface for the bug's solution may present itself once character thoughts and interactions are implemented.

Characters will have relationships with each other, they will like certain people more, depending on interests ect. This means they need to keep a record of who they have interacted with. This means every character that gets created needs to have their own personal ID which is used for interactions. When a new customer gets spawned, the computer will randomly choose between creating a brand-new character, or spawning in a character that has been used before. If it is an already existing character it will have the same ID and will have all the same stats.

The world needs to keep a dictionary of all created characters so that their stats and variables can be recorded and used if needed.

Code

All characters now have a unique ID. The world is aware of all characters, and they can be looked up if their ID is known. When characters interact, they both check their relationship dictionary with the character they are interacting with, if none of the relationships match then they must have not interacted before and a new relationship is created and stored.

New function has been created. It checks to see if the character has a relationship with the given chat cater, if not a relationship is created.

If two characters are next to each other, they check to see if their relationship level is above a certain point, if it is they say hello. If it's a high enough level, they will continue their conversation.

Characters can now be created without a specified tile; this means that they will not be spawned into the world. This is used when creating pre-existing characters at the beginning of the world creation; and can also be used in the future if maybe characters find out information about another character that isn't in the world, and has never been in the world.

Added traits to characters. Added public functions which add these traits. Some traits cannot be added if the character has the opposite traits, for example Quick and Slow. Some traits such as quick, athletic and slow, modify the character's max speed.

If two people are adjacent to each other, one of them will test to see if they should talk. If their relationship level is above 30, they will say hello to each other, then if their traits are correct, or their relationship is above 70, they will talk for a random amount of time, depending upon their relationship level, the higher the level the longer the conversation will be on average. There is a flag to determine if two people have spoken recently, this means that they will not continue to try to talk to each other after the conversation is over.

If a character is in the way of another character, the character that is moving tries to find another way to their destination. If they cannot find one, they will ask the character that is in their way to move; at that point the character that needs to move will set their destination to the nearest free tile. A check has been added for interactions. When character 2 checks if it's interacting with another character, character 1, then it will check to see if character 1 is still interacting with them, and if they are not, they will stop their interaction.

Instead of immediately trying to find a way around a blocking character, the character will wait up to 5 seconds before trying to find a way around. If the blocking character is also waiting for another character to move or is interacting, then the character doesn't wait the 5 seconds, its immediate.

28th March 2017

Plans

1. ~~Add employee/customer interaction at checkout.~~
2. ~~Add full conversation/interaction logic.~~
3. ~~Add relationship UI information.~~

Work

Added new UI information. When a character is selected and they are interacting with someone, their relationship level is shown.

Code

New functions added. The character that requested the interaction is the first to talk. They randomly choose a number between 0 and the max interaction time, they speak for that length of time. Once the finish talking, depending upon their traits and their relationship level with the other character, they will have said something negative or positive, this is determined by the speaker's traits. The relationship change is chosen at random, and will change from the number of negative traits the character has, to the number of positive traits that the character has. For example, if the character has 3 negative traits and 4 positive traits, then a number will be chosen between -3 and 4, and that value is the relationship change between the two character. If the listening character has certain positive traits such as understanding, their negative relationship change will be less, but also, if the character has certain negative traits, such as jealous, then the positive relationship change might be less also. If the other character has certain negative traits such as annoying and doesn't have the positive trait of quiet, they have a 20% chance of interrupting the speaking character. If they do, the speaking character will not mind that if their relationship is good enough, however, if their relationship is not good enough and they have certain negative traits such as dramatic, they will have a negative relationship change with that character. At that point they will stop talking and then the conversation is flipped. This will continue until the interaction timer reaches the maximum. Characters' relationship levels with certain characters will only change if that character said something positive or negative.

Fixed an issue with characters not realising they have finished a conversation with someone. Now if a character needs to end a conversation, either because they were asked to move, or because the interaction is over, then they will tell the other character they are saying goodbye. At which point both characters set their m_interactingCharacter variable to null and allow interacting again.

Employees will now start conversations with customers they are serving. Their conversations are like normal conversation except they do not stop the characters from doing their tasks or jobs, which means the employee can still scan items, and the customer can still put them down and pick them up.

There was a problem with the conversations where if it was finished early, no relationship changes occurred. This has been changed so that if there were no relationship changes throughout the conversation, then at the end, even if interrupted, a relationship change will occur.

The amount of change in the relationship is now affected by more factors. These include starting relationship levels; higher it is, the more positive the change will be after the conversation; and also a topic level; this randomly chooses a value between 5 and -5 which simulators a positive or negative topic, this is factored into the final relationship change; and a factor that was already in the system was the positive and negative traits of the person speaking.

A potential issue was found where employees could say very, very negative things to customers, and this should be capped as employees would never speak that negatively while working. This was resolved by capping the change if the relationship was with an employee, to -3.

Added shop influence. If an employee is rude to a customer, the customer has a chance of losing relationship with the shop. The amount is affected by the traits possessed by the customer and the starting relationship level of the shop. As the level increases, bad influences are lessened, but also, as the level decreases, positive influences are lessened. As the level increases, above 80 and 100, it is less likely that good influences will increases the amount more, so the positive influenced are divided by 2 and 4 to simulate this. Also, if the level is below 30 and 10, negative influences will have an even greater impact on customers, so those influenced are timed by 2 and 4. Once a relationship level with the shop goes below 0, the customer will refuse to come to the store, and if they are chosen by the world as the next customer to spawn, they simply won't, this means that the rate of customers is reduced.

6th April 2017

Plans

1. ~~Fix furniture movement bug.~~
2. ~~Add event messages.~~
3. Set up scenario.
4. ~~Add more sprites for characters.~~
5. ~~Add sounds.~~
6. Add beginning and end screens.

Work

A bug was found which meant that the characters were using old nearestFreeTile information to set their moving furniture's destination tile. This occurred when the recursive nearestFreeTile function was written. It worked perfectly for when you didn't nee d the furniture, but when the character did need the furniture, it would use the last nearestFreeTile information. One solution would be to set the nearestFreeTile back to null when it was finished being used, but then a replacement for it must be found. Using the same system as the nearestFreeTile function, when a character is next to where they need to be, there isn't any information about where to put the furniture if it is in the way because the character's path ends on the next tile. The solution was t o simply find the nearestFreeTile at that point, and the issue is resolved because the information it was using was up-to-date information.

Event messages will be used to inform the player about various events that occur in the simulation. For example, when an employee says something negative to a customer, and if a customer's relationship with the shop drops.

For testing purposes at the end of the project, a scenario needs to be developed to unsure that the testers can get a good feel for the best parts of the AI, so they can give honest, informed feedback on the realism and intelligence of the simulation. The time of day will be mid-morning, and the scenario will only last a few hours. Due to the simulation running in real time, and the maxi mum speed up time is 20x, then even that amount of time might not be got through by the testers. Customers will be in the shop, along with two or three employees. The customers will have different items they require, and the spawn rates of the customers that enter the shop will be different each time the simulation is played. This will give the testers the best chance at a good fell for the AI in the game.

At the moment, employees and customers look the same, this needs to change. Employees will have a darker, suit looking uniform on, whereas the customers will all have brighter clothes on.

Sounds will be added subtly, such as when a transaction ends, or when customer enter or exit the world.

The beginning and end screens will be there to allow the user to begin the simulation in a controlled manner when they wish to begin it, it will also contain brief instructions about controls and the idea of the simulation, and after the few hours of game time is over an end screen will appear to remind the user on how to complete the analysis form required for the project, and instructions on how to restart the simulation if they want to.

Code

Changed the employee sprite, and added 4 customer sprites. When a customer spawns, a random number is picked between 1 and 4, and then the customer sprite associated with that number is assigned.

Added three sounds to the game. The first is general background music which plays on loop. The other two are sounds which play when a transaction finishes, and when a customer spawns.

Added customers reducing the faced up percentage of furniture when they pick up items. Added UI onto the furniture information so that the user can see the faced up percentage of the furniture.

Discovered a big bug with the furniture movement code. When a character wanted to move a trolley by pushing it and it did so, and reached one tile away from where it needs to go, since the character does not have pathfinding information for the tile the trolley needs to get to, the character finds the nearest free tile as its reference, however this shouldn't happen. The character has the final tile the furniture needs to get to, so it should use that instead, since it should be one away from that tile anyway. This does not work however is for some reason the character does not know the final tile the furniture needs to get to, in which case it will fall back on the nearest free tile code to find a good spot for the furniture.

Another bug was found where occasionally a trolley would not get deleted from a tile it has been moved from. This causes the tile to still believe it has a trolley assigned to it, and when another character comes along it thinks there is a trolley there, but cannot move it since it isn't actually there. This causes some odd behaviour where the character will try to go to where the trolley actually is, which may be the other side of the shop. It is unknown how this bug occurs since when a trolley is moved onto a tile, it is automatically deleted from the previous one, since no cause has been found, a solution has also not yet been found.

13th April 2017

Plans

1. ~~Finish scenario~~
2. ~~Add full beginning and end screens.~~
3. ~~General Alpha Testing~~

Code

The camera is now clamped to the size of the world, which means that the user cannot drag the camera away from the shop. Also, the maximum size of the camera view is now 6, meaning the screen will be zoomed in more. This was done as it felt like being able to see the entire shop one screen was not ideal.

The bug mentioned in the previous entry has not been fixed, and so a 'cheat' fix has been implemented so that the program can be released to the AI testers. The fix solves the problem like this: the world asks the trolley where it is, it then goes through each tile, and if the tile contains a trolley that isn't the tile that the trolley is actually on, it removes the furniture. Another small addition needed to also be made in the furnitureActions class, which counters the other 'cheat' fix if it was to ever mess up. If the program did ever get the conditions slightly wrong and caused a logic error, then the furnitureActions class will put the trolley back down on the furniture's main tile if it was to ever not be there. These two tixes allow the AI and pathfinding to work properly as intended without an unforeseeable number of man-hours being wasted on a small fix.

The second checkout was removed since it wasn't needed and the queue AI is not fully implemented in regards to customer to knowing that the checkout they are on isn't working. The queue tiles have also been changed and updated to go along with the removal of the second checkout.

Added small amount of code which means that after a customer has looked in a piece of furniture for items they need, they will wait 5 seconds. This means that they do not automatically stop and go right past a piece of furniture that does not contain any items they need.

Similar code was also added to the employee's facing up logic. Now whenever they face up a piece of stock, they will endure the same time waiting as if they picked it up themselves. This means once again they are not whizzing from tile to tile quickly facing up the entire store in seconds, each furniture takes a few minutes to completely face up.

Added code to allow a random chance of an already created customer to come back to the world after leaving it. There is a 50/50 chance that an old customer will return instead of a new one being created. Fixed issues with re-spawning customers that had already left the map.

When a character is created, a random number of traits are added to them, and their name is picked from random from a list.

16th April 2017

Plans

1. ~~Create the analysis questions.~~
2. ~~Polish and complete the start and end screens.~~
3. ~~Create the draft email to be sent to testers.~~
4. ~~Create webpage for the testing.~~

Work

Start and end screens completed as well as a brief instructions screen. The analysis questions have been created with a disclaimer at the beginning emphasising the need to focus only on the AI and ignore other under-developed systems such as the art and general game-play.

A problem was found with embedding the program onto a webpage. Instead of this path the program will instead be uploaded to the github page and testers will need to download the program and run it, which is not ideal but the best option at this stage. The webpage still contains all the information required for the testers to learn about the program, its aspects and the process of downloading the program and the questions.

20th April 2017

Plans

1. Beta Testing

Work

Requests for help on the dissertation have been sent out, and responses are being waited for. When they arrive, the email will be sent and them it is up to the testers to test the program, and give the analysis feedback in their own time.

4th May 2017

Work

Initial dissertation write up has begun. Most of first section. Word count at ~1200 words, has been completed.

7th May 2017

Work

Completion of the first section has been done. Word count at ~1500.

Moving onto second section: Background Theory and Literature Review. Finished with word count of ~2300.

Moving onto third section: Design, Development and Testing Process. Finished for the day at the pathfinding stage, word count at ~3800.

8th May 2017

Work

Carrying on with third section: Design, Development and Testing Process. Finished up to testing phase, ~7500 words.

9th May 2017

Work

Carrying on wiith rest of the dissertation. Finished Dissertation with ~10500 words.