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>

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 multiplies 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.

Coding

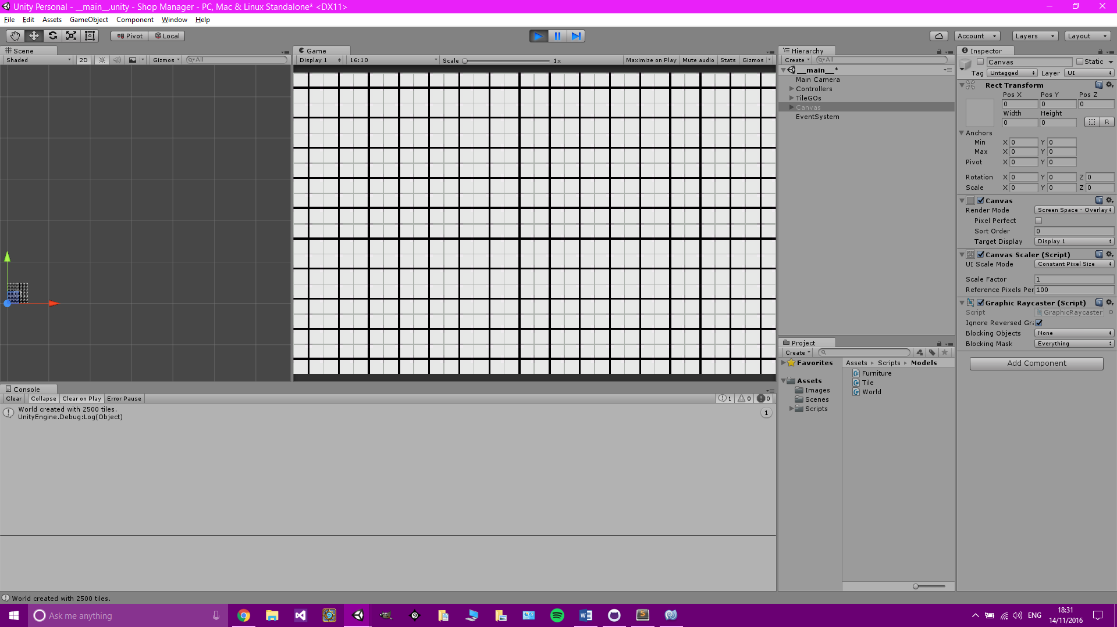
No visible changes on the game screen yet, all background processing.

List of created scripts:

1. World Controller
2. World
3. Tile

Purposes of created scripts:

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.



7th November 2016

Plans

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.

Coding

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.

Created Scripts

1. Mouse Controller

Purposes of created scripts

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

Plans

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

Coding

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

Created Scripts

1. Furniture
2. Furniture Sprite Controller

Purposes of created scripts

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

Plans

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

Coding

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

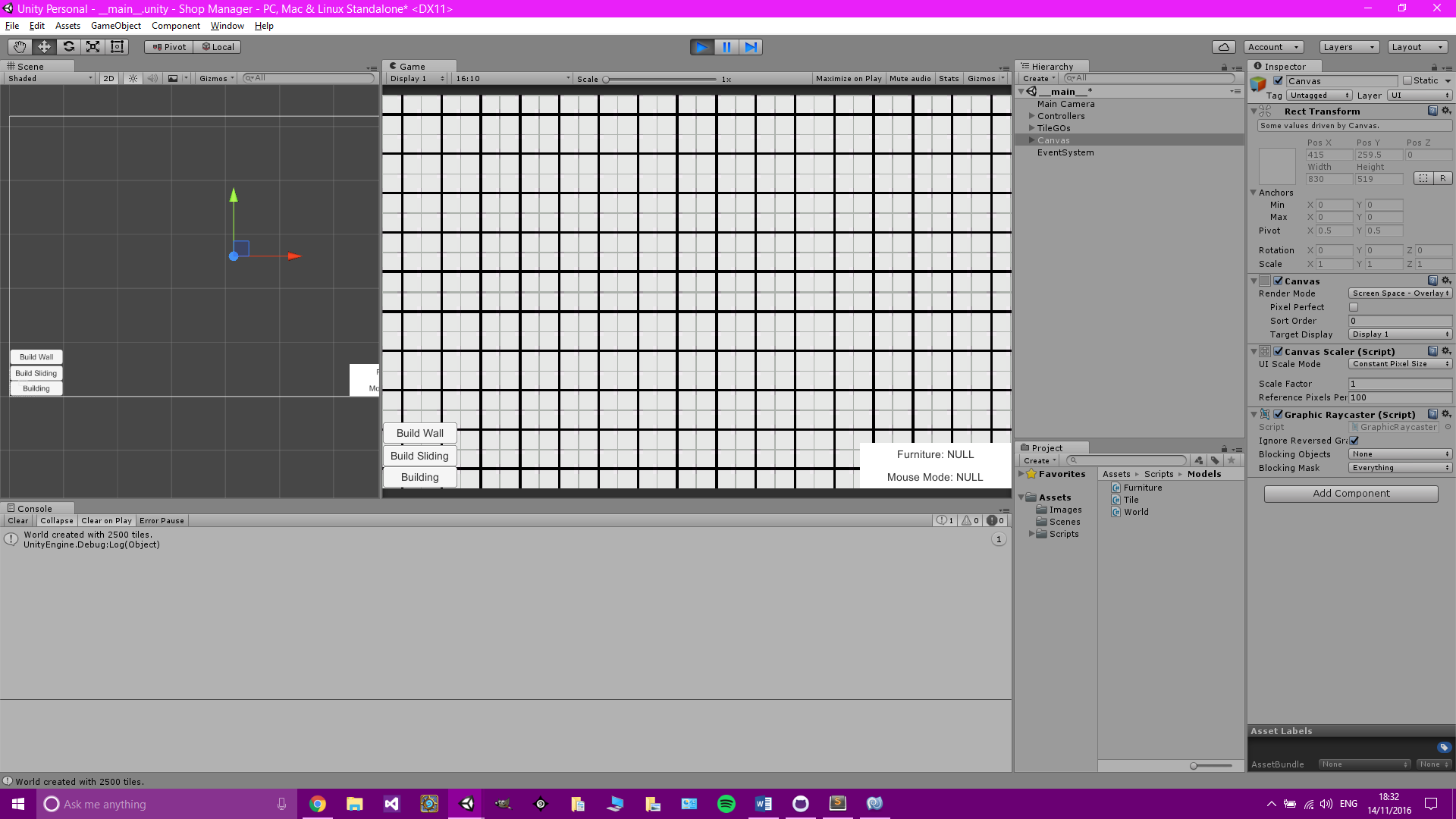
All that that is required is to add the prototype to the world class, and its parameters. Then add a button and the furniture will be created in the world when the button is clicked and a location is set.

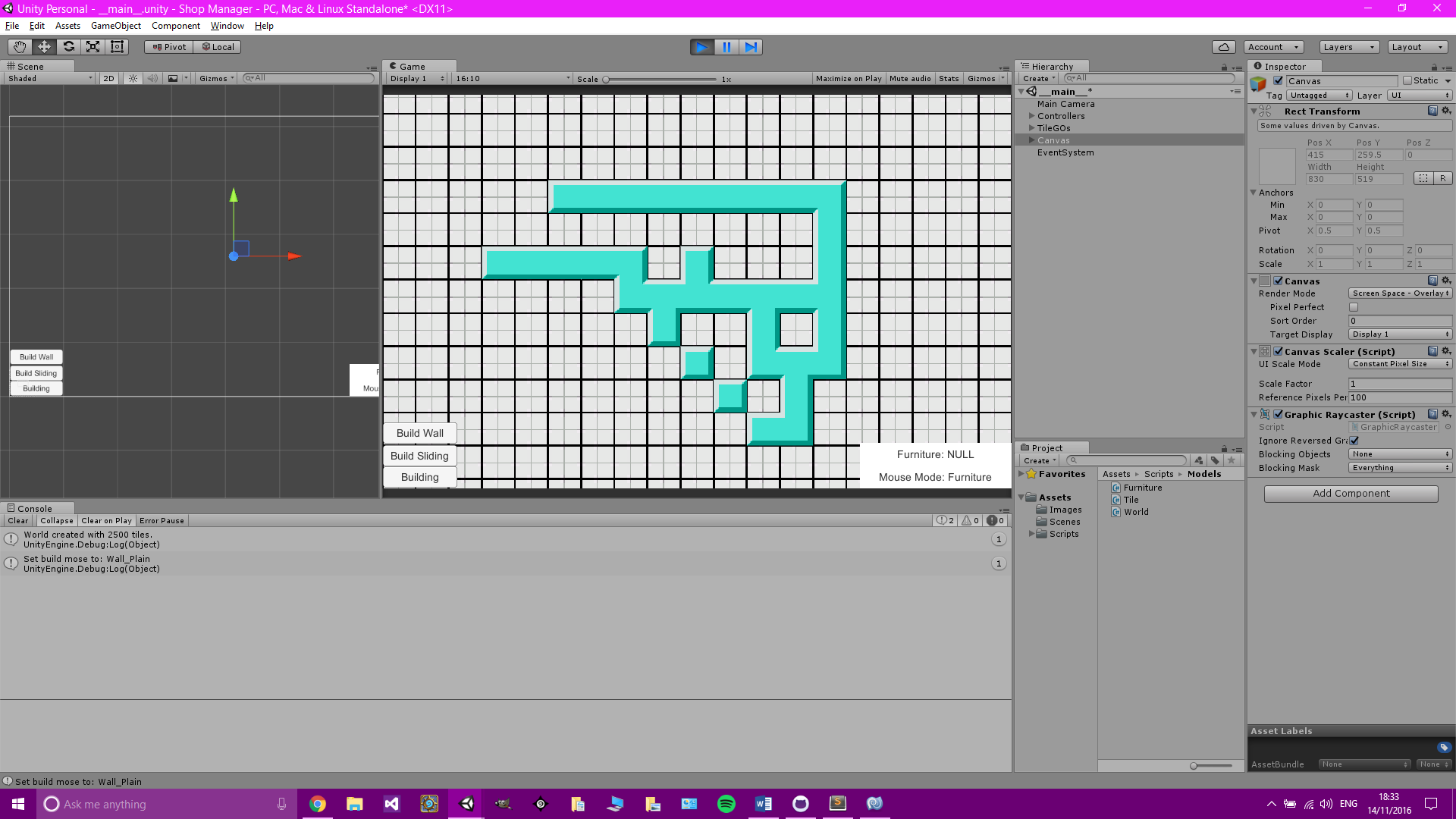
Created Scripts

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

Purposes of created scripts

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.





13th November 2016

Plans

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

Coding

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.

Created Scripts

1. Furniture Actions

Purposes of created scripts

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.

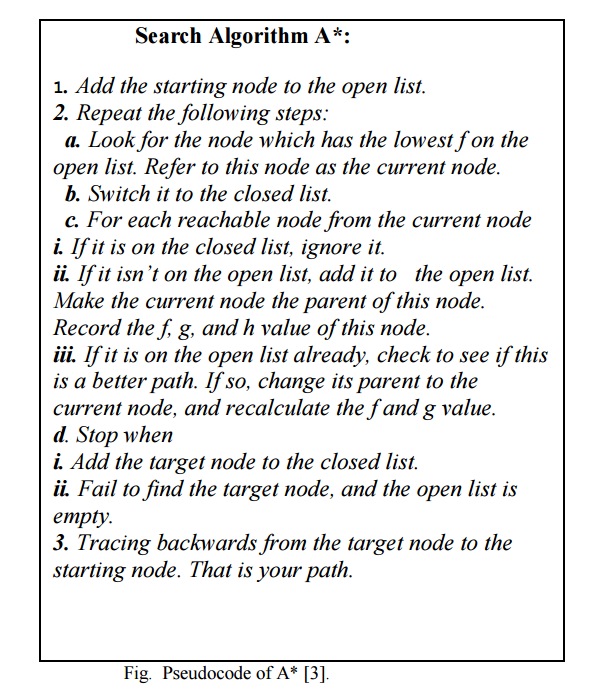
If the path is suddenly blocked by a mobile object it becomes obsolete and the agent has to recalculate all or parts of it. Extensive work has been done to modify A\* to work better in highly dynamic worlds. Silver (D. Silver, 2006) [1] proposes an addition of an extra time dimension to the pathfinding 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-tile 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.***

1. 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 its 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.

[6]

**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:

* 1. Fit a coarse grid on top of the fine grid. Precompute the shortest path between any pair of coarse grid locations.
  2. 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:

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

Literature Comments

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.

References Used in the works

[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);2009ˈ 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

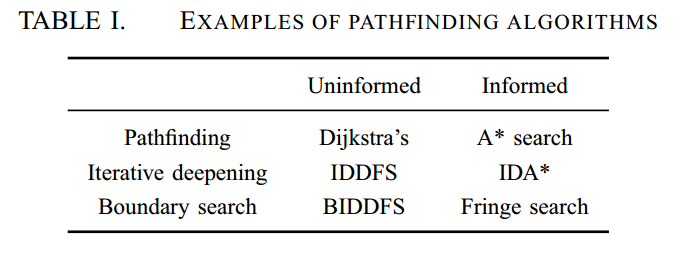
1. Uninformed Multigoal Pathfinding on Grid Maps <http://ieeexplore.ieee.org/document/6946181/>

Literature Details

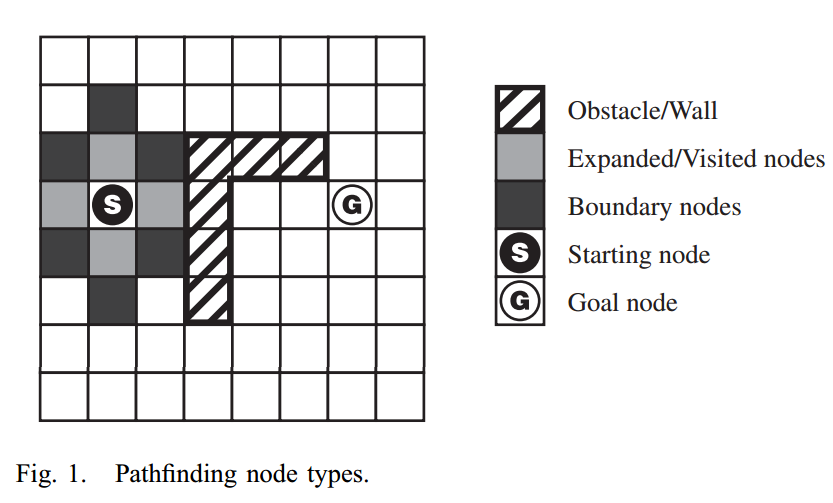
1. There are two classifications of pathfinding algorithms: Informed and Uninformed.

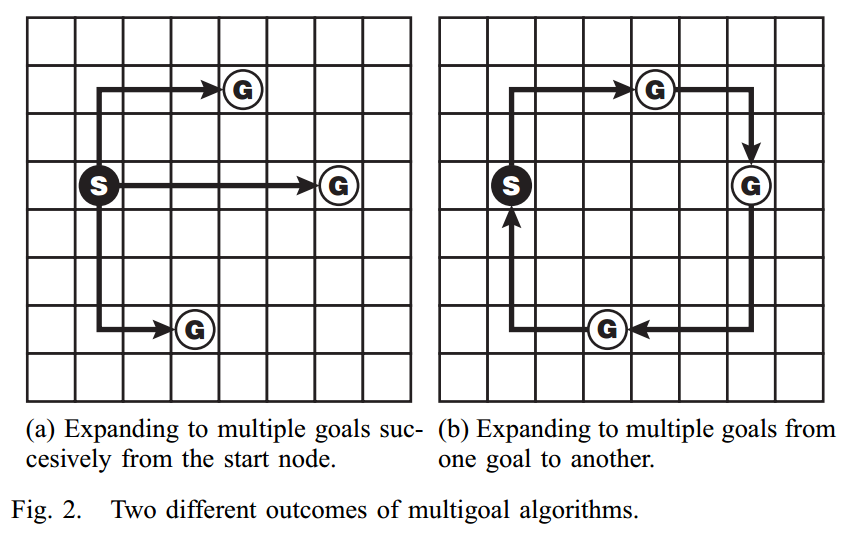
Informed: Involves the use of a heuristic function [1] to estimate the location of the goal. The direction of pathfinding is guided towards the estimate, making informed searches typically faster than uninformed searches (but can be less optimal)

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].

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.

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].

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.

1. 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.

1. 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

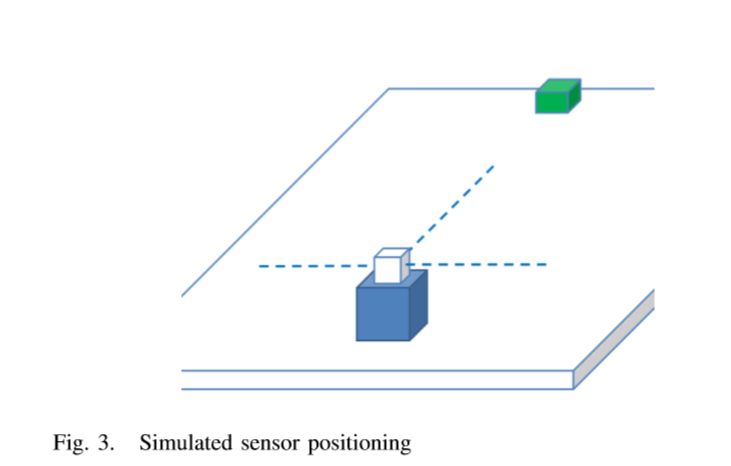
1. Pathfinding in Partially Explored Games Environments <http://ieeexplore.ieee.org/document/6930151/>

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.

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

Purpose of Created Scripts

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

Plans

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

Coding

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 should 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.

The Model

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 α∈[0,1] denotes the fuzzy value and b>0 denotes forgetting rate.

α = 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.

Basic Emotions

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.

1. 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 Ill, 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 serve 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 well 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.

Literature Comments

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.
3. .

References Used in the works

[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 DeIrt, Netherlands, 200 I.

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/>

1. 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.

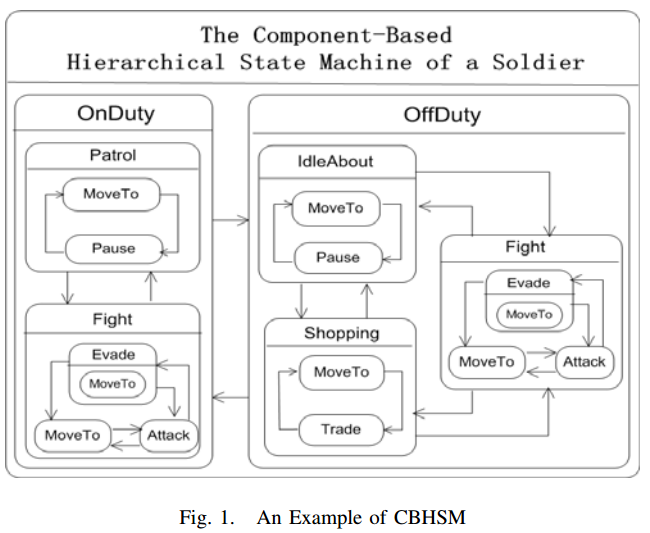
1. Traditionally, computer games developers use a small set of techniques over and over again in the implementation of artificial intelligence (AI) functionalities in video games, especially Finite State Machine (FSM) [9] [10] is the most frequently used one. In video games, FSMs are typically used to 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 [14]. Over class inheritance, object composition, another common technique for reuse, is favoured by ‘Gang of Four’ [3], 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.

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] O. Castillo and P. Melin, Type-2 Fuzzy Logic Theory and Applications. Berlin, Germany: Springer-Verlag, 2008.

[2] H. Hagras, “Type-2 FLCs: A new generation of fuzzy controllers,” IEEE Comput. Intell. Mag., vol. 2, no. 1, pp. 30–43, Feb. 2007.

[3] J. M. Mendel, Uncertain Rule-Based Fuzzy Logic Systems: Introduction and New Directions. Upper Saddle River, NJ: Prentice–Hall, 2001.

[4] D. Wu and W. W. Tan, “Genetic learning and performance evaluation of type-2 fuzzy logic controllers,” Eng. Appl. Artif. Intell., vol. 19, no. 8, pp. 829–841, 2006.

[5] D. Wu and W. W. Tan, “A simplified type-2 fuzzy controller for real-time control,” ISA Trans., vol. 15, no. 4, pp. 503–516, 2006.

[6] D. Wu and W. W. Tan, “A type-2 fuzzy logic controller for the liquid-level process,” in Proc. IEEE Int. Conf. Fuzzy Syst., vol. 2, Budapest, Hungary, Jul. 2004, pp. 953–958.

[7] H. Hagras, “A hierarchical type-2 fuzzy logic control architecture for autonomous mobile robots,” IEEE Trans. Fuzzy Syst., vol. 12, no. 4, pp. 524–539, Aug. 2004.

[8] L. A. Zadeh, “The concept of a linguistic variable and its application to approximate reasoning-1,” Inf. Sci., vol. 8, pp. 199–249, 1975.

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

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

[11] GIRAULT, A., BILUNG, L., LEE, E., 1999. Hierarchical finite state machines with multiple concurrency models. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 06 August 1999 Berkeley. California: IEEE Press, 742-760.

[12] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In Proc. of the 26-th International Colloquium on Automata, Languages and Programming, ICALP’99, LNCS 1644, pages 169-178. Springer-Verlag, 1999.

[13] Valery Sklyarov, Hierarchical finite-state machines and their use for digital control, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.7 n.2, 222-228, June 1999.

[14] 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

Literature Details

* 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.

Finite State Machines

* 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 states 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.

Decision Trees

* 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.

Fuzzy Logic

* 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.

Literature Comments

1. Clearly, there is a lot of information in this book. Most of it is appropriate and informative.

Finite State Machines

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.

Decision Trees

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.

Fuzzy Logic

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

Plans

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.

Coding

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.

Created Scripts

1. Employee
2. Job

Purpose of Scripts

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.

Created Scripts

1. Stock

Purpose of Scripts

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.

Found Problem

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.