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. Tile-Based Game Design – Springer Link

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

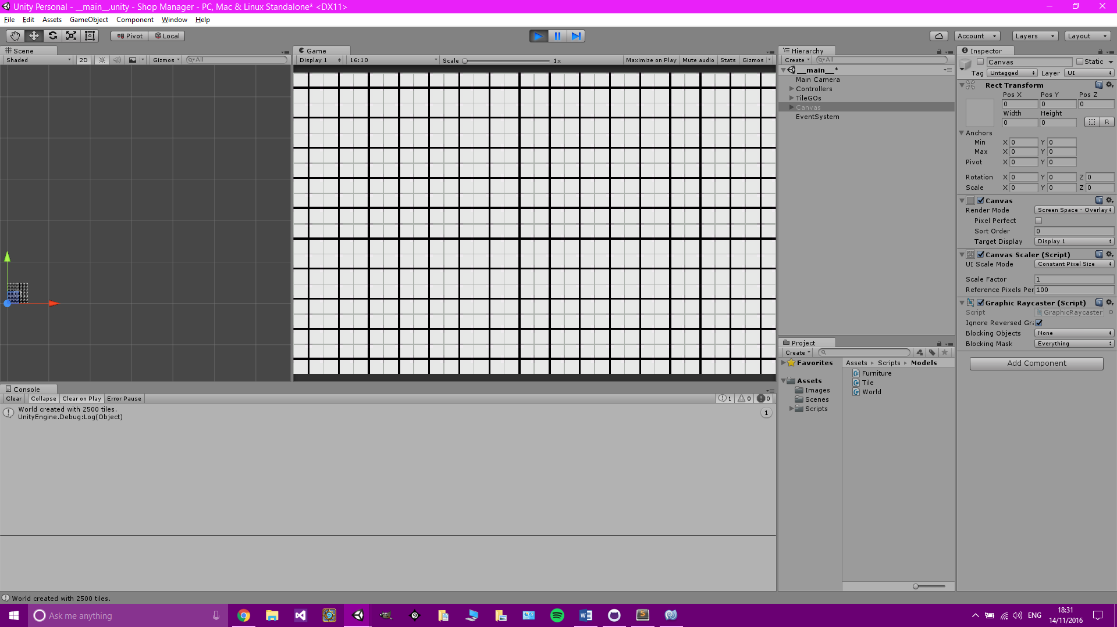
No visible changes on the game screen yet, all background processing.

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.

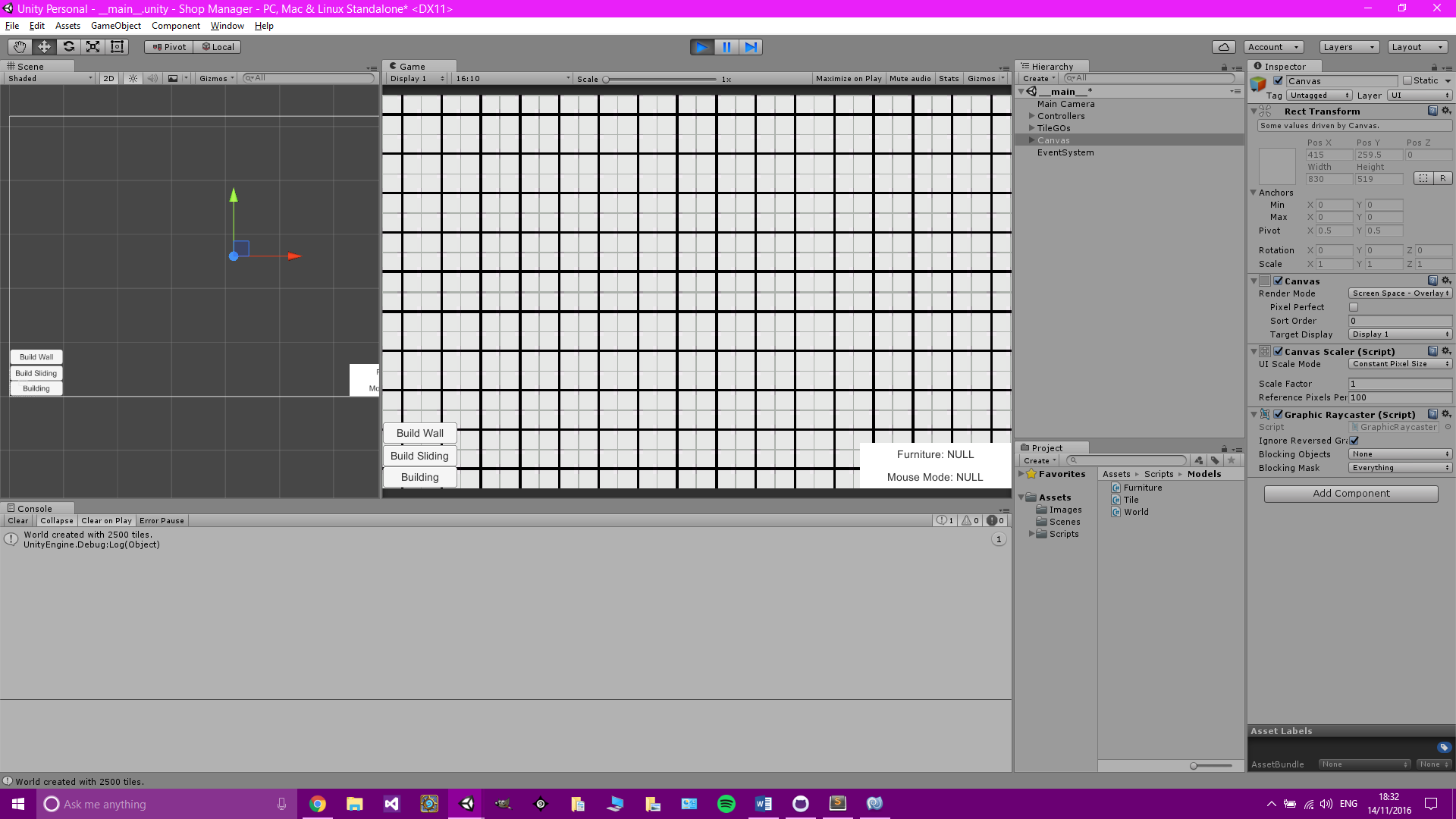
All that that is required is to add the prototype to the world class, and its parameters. Then add a button and the furniture will be created in the world when the button is clicked and a location is set.

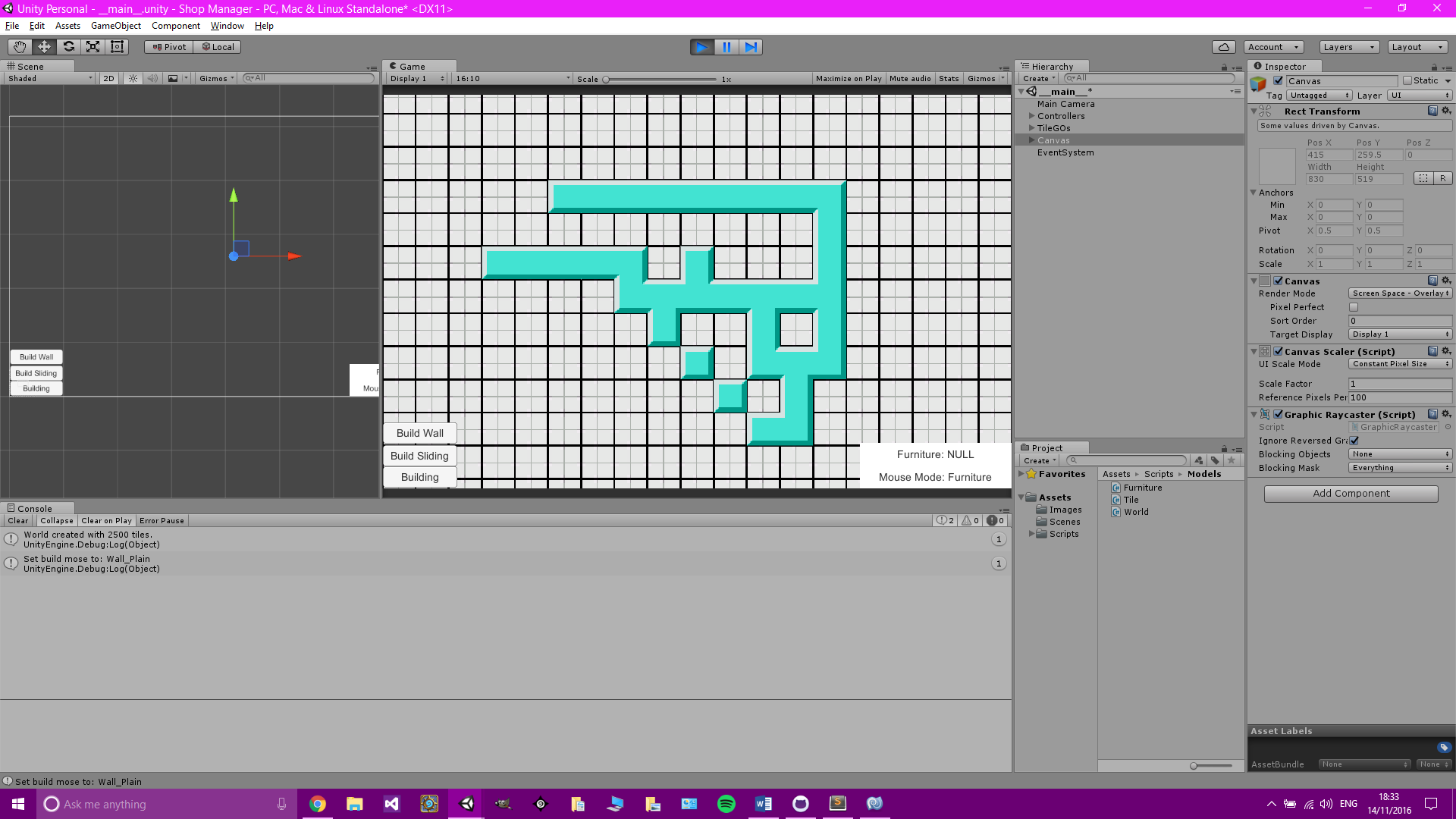
Created Scripts

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.

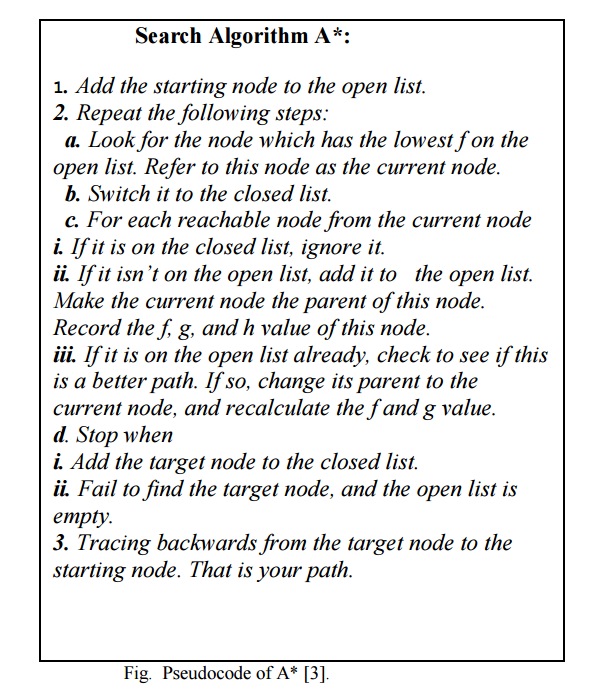
If the path is suddenly blocked by a mobile object it becomes obsolete and the agent has to recalculate all or parts of it. Extensive work has been done to modify A\* to work better in highly dynamic worlds. Silver (D. Silver, 2006) [1] proposes an addition of an extra time dimension to the pathfinding 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 it is unclear about whether either of these methods will be used in this project. Another review will be done on boids, and another on potential fields, and if the two are found to be useful then this paper will be revisited and the comparison will be commented on.

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.

1. 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

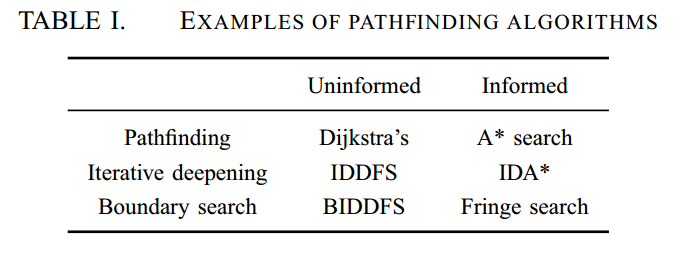
1. Uninformed Multigoal Pathfinding on Grid Maps <http://ieeexplore.ieee.org/document/6946181/>

Literature Details

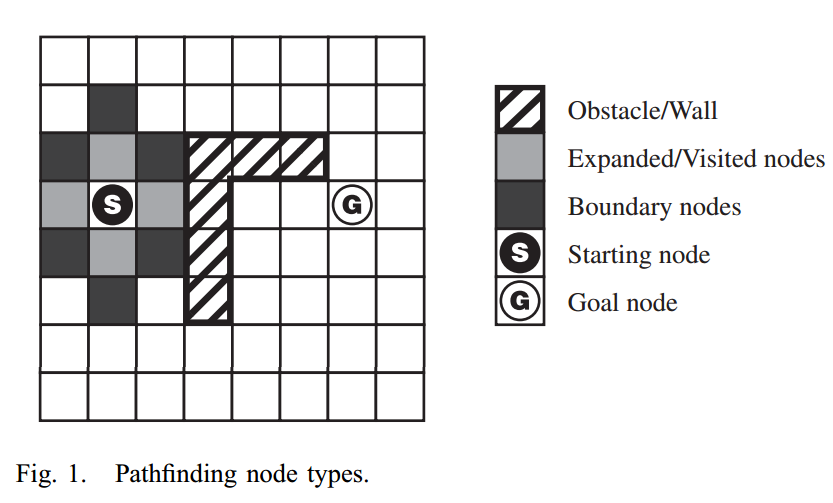
1. There are two classifications of pathfinding algorithms: Informed and Uninformed.

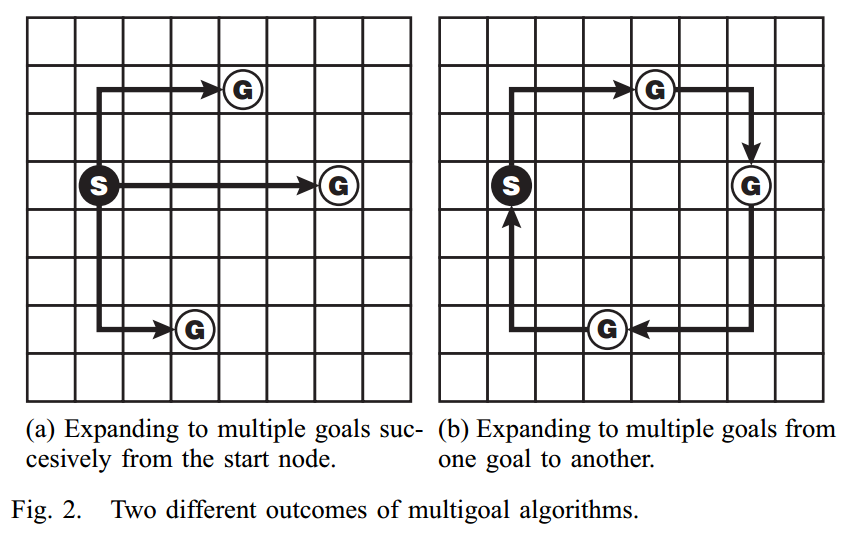
Informed: Involves the use of a heuristic function [1] to estimate the location of the goal. The direction of pathfinding is guided towards the estimate, making informed searches typically faster than uninformed searches (but can be less optimal)

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

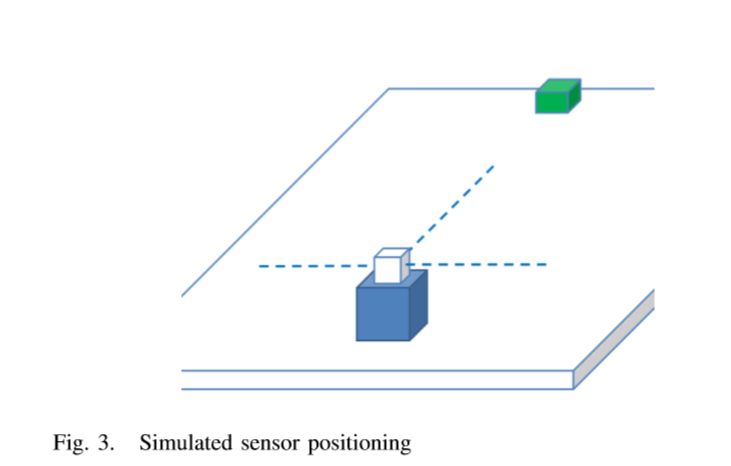
1. Pathfinding in Partially Explored Games Environments <http://ieeexplore.ieee.org/document/6930151/>

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.

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.