6.4.2020

Johnny Castaway

Introduction

Inspiration to this project came from an old screensaver called Johnny Castaway, originally released in 1992 by Sierra On-Line. The screensaver was marketed as the world's first story-telling screensaver. (Wikipedia, 2020. Johnny Castaway). Among a variety of other events, the castaway Johnny is seen fishing, climbing into the single palm tree on the island and jumping into the water, building sand castles and sending and receiving bottle notes.

The scene is set somewhere along the south pacific on an island with a few palm trees and berry bushes, not to mention seagulls roaming around looking for a good place to fish. Castaway Johnny's basic needs are taken into account while creating this project. Johnny needs to be able to realize he's hungry or tired and act accordingly. While Johnny is a fairly autonomous character, he is also able to walk on the island along a path he calculated in order to reach a target position set to him by a click of a mouse. Johnny also knows not to value the click of a mouse more than his basic needs, so don't expect him to answer your call before he has a stomach full of berries and a well-rested night. (YouTube, Johnny Castaway.)

Technical decisions

Monogame worked as a base for the project. The decision to switch from Windows Form to Monogame came from increasing the performance of the simulation. Monogame also has a Vector2 (Microsoft Documentation, Vector2) ready to use within the program. Nonetheless Vector2D was implemented to the full extent and used it in the early stages of the project. Monogame also gives some extra features within the use of sprites and movement calculations (Whitaker's Wiki, Monogame - Drawing Text With Spritefonts & Monogame Spritebatch Basics). As Monogame also has features to use tiled maps, the map was created with a software Tiled and free tilesets found online. These tools made the flooding of the island with a graph fairly easy.

Project was conducted in C# as it's a familiar language to both authors and has a lot of useful libraries to manage a project of this magnitude and Visual Studio 2017 was used as a development environment due to its easy and responsive user interface and debugging features.

Steering Behaviour

Survivor

Survivor's initialized behaviour is IdleBehaviour, a new behaviour to keep the survivor from roaming around before it receives a target from either a decision to eat or sleep, or an external target to seek to. Survivor however fairly quickly realizes it's hungry and decides to follow a path to the nearest bush to eat berries and switches to arriving behaviour to traverse through the nodes along the way.

When implementing steering to the character, seeking and arriving were considered. Seeking is a faster way of transporting, but with a small island and a dense graph like in this project, arrival seemed like a better choice throughout. With arriving the hardest part was to get the survivor to stop on a given destination node as the survivor continues to infinitely walk over the destination point after reaching it. During implementation even adding friction was tried, but tweaking the values and applying that to the calculation proved to be too hard given the time frame of the project and so arriving remains imperfect. However with the given goals and other features, the arriving seems to work at a decent level to achieve a given criteria.

Arrival behaviour's calculation goes as follows: Calculate the difference between the survivor's current position and the destination position as a vector and get the length of this vector. Using a deceleration value given to the arrival calculation, determine the speed of moving the survivor towards its destination. Multiple decelerations used in a bigger graph could be then used in steps to make the survivor slow down as it gets closer to the destination. In this project, the survivor is given a smaller deceleration value of 1 to make it move faster towards the goal. Multiplying the length to the destination point with calculated speed value and dividing it with distance between the survivor and the destination point we can determine the desired velocity for the survivor. After this, subtracting survivors current velocity with the desired velocity we get the survivor to move in a smooth manner to the destination. Important to remember while this calculation is to check, that the destination is still further from the survivor as we want it to stop on the target. (Buckland, M. 93)

Seagull

The colony of seagulls is implemented using the Flocking behaviour. Within the Flocking behaviour is the Wander behaviour used as an base movement for the seagulls. The flocking uses the three base calculations: Cohesion, Alignment and Separation. Each seagull calculates its own neighbourhood. It will tag other seagulls that are nearby. This neighbourhood is used in each calculation of the steering force. All these values together make the steering force for the flocking. If there are no other seagulls in individual seagulls neighbourhood it will rely on the wandering behaviour. The seagulls won't interact or interfere with the survivor, so the flocking behaviour and its calculations aren't affected by the presence of the survivor. The seagulls won't consider the survivor as a neighbour so it doesn't get tagged. This is how we separated the seagulls from the survivor as seagulls fly on the sky and the survivor walks along the ground level. (Buckland, M. 113-126)

The most difficult part of the flocking behaviour is the balance between values. We spend a lot of time on finding the sweet spot for a good behaviour of the seagulls. In our opinion the seagulls look really natural after a while with the current values. The flocking behaviour is visible at all times

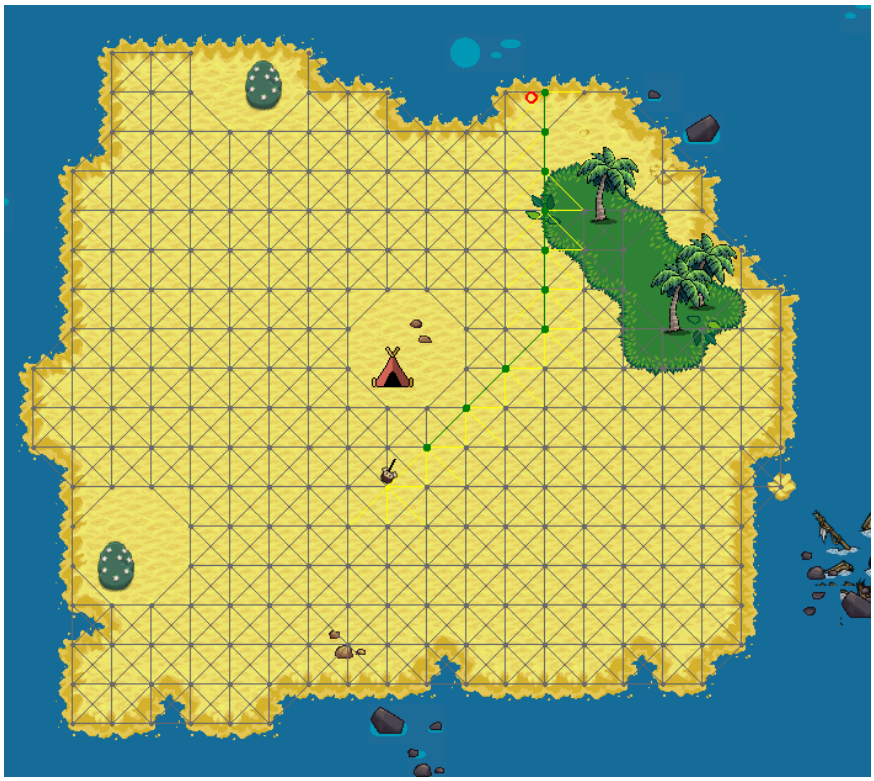
on the seagulls. After a minute the start spreading and the wander behaviour usually kicks in. If the seagulls meet again the flocking takes effect again and this can be seen with a seagull with smaller velocity turning in alignment with another seagull and following its route.

Path planning

Graph

Navigation graph was created to implement pathfinding for the survivor. Graph was implemented as an undirected and a dense one, as it seemed to work with the small field and works well with survivors arriving behaviour. (Buckland, M 2005. 193-242). Because Monogame supports tiled maps, creating a graph on a specific area was fairly easy as was filling and drawing the graph on the island. Graph is set according to the specific layer of tiled map and can be set with the identifiers of each tile. (Karim Oumghar 2015. Graphs and Dijkstra's Algorithm (C#).)

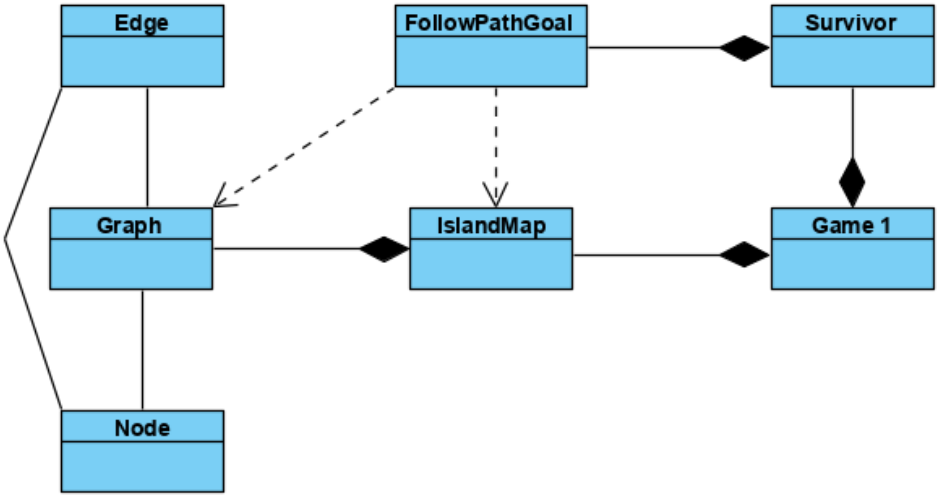
Since the island consists of multiple static objects like palm trees and berry bushes, the graph takes out nodes and edges around them according to their width and height. This part created some confusion as Monogames axises are set to 0 from the top left corner. This created some hardship with trying to create smaller openings to static objects and their offset, but since the survivor can still walk over the sand not filled by the graph, it did not matter too much.



A* and Manhattan heuristic

A* - algorithm uses the Manhattan heuristic to evaluate the estimate from the starting point to the destination point, counting the estimated distance in the x- and y-axises. To get a target in between nodes and edges, A* uses a method to get the nearest node for the start and the destination node. This method also keeps the pathfinder from trying to find its way outside

the graph, like the water surrounding the island. A* uses a node and edge lists to determine shortest edges to the next node and keeps a record of visited nodes. After calculating the heuristic values and use of priority queue, A* marks the drawable nodes and gives out a linked list of nodes to the pathfinder. Assembling this part did not go without difficulties and using the priority queue and receiving the right nodes could not have been done without outside help. (Buckland, M. 2005. 241-247) (TheHappieCat 2017. How to Do PATHFINDING: A* Algorithm (The Thing Most Games Actually Use)).



Goal based behaviour

Survivor has one default goal, MakeDecisionsGoal. Within this goal, the survivor gets composite subgoals based on either user input or its needs, hunger and fatigue that extend to fuzzy rules. Each goal has a set of parameters needed to create the desired goal and to complete it. With activation the base of the goal is set and on each goal's parent process the individual subgoal's processing is called.

Difficulty with calling subgoal within a subgoal within a subgoal created some confusion, but ultimately created a fairly autonomous character that chooses its own goals over user input but remembers the excess goal if these goals were to overlap. It was also important to check, that the composite goals did not complete before their atomic subgoals were achieved first or else the survivor would get stuck on for example eating infinitely, even if it would get tired and needed to go to sleep.

Atomic goals

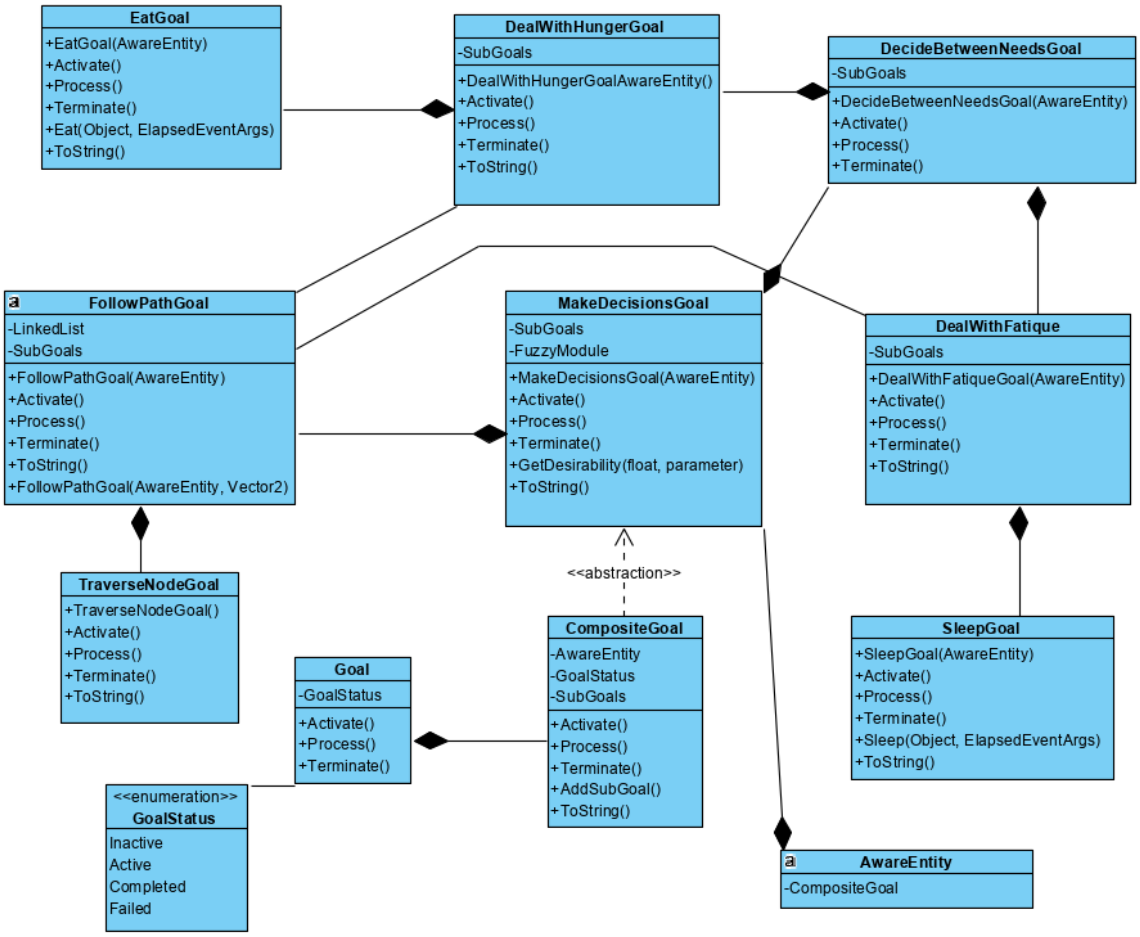
Three atomic goals were created for this project: Eat, Sleep and Traversing through a vector. These atomic goals made sure that simple tasks like increasing the value of hunger (meaning getting less hungry) or sleeping until no longer tired were processed. Atomic goals we're pretty simple to implement and using a Timers class made counting the increase of eating and sleeping easy. This also did cause the need to make sure in the parent class, that the atomic goal was actually done as the timer was elapsing according to the machine's timer. (Buckland, M. 2005. 382- 399.)

Composite goals

With MakeDecisionsGoal four other composite goals were implemented. Composite goals handle the bigger picture of the goals like FollowPath-Goal continues to process until all of the nodes found by A* are traversed by its subgoals and there are no more nodes to traverse to. Hunger and fatigue dealing composite goal first finds the needed static object like a tent or a bush, then uses the A* and PathFollowing goal to get to that object before calling the actual sleeping or eating goal. (Buckland, M. 2005. 385-399.)

See image below.

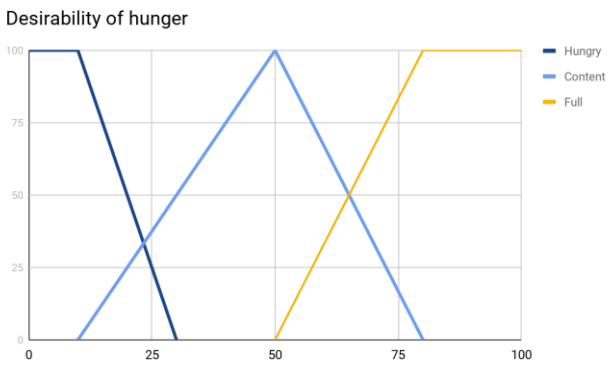
6.4.2020

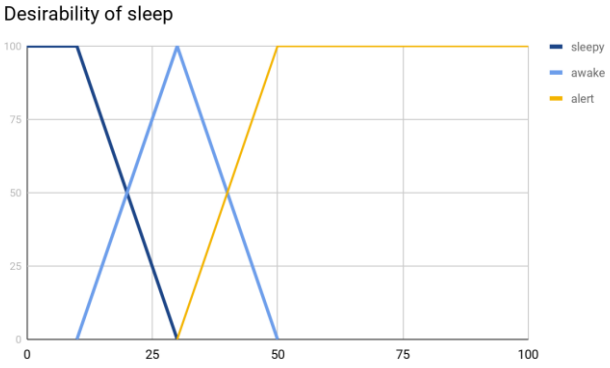


Fuzzy Logic

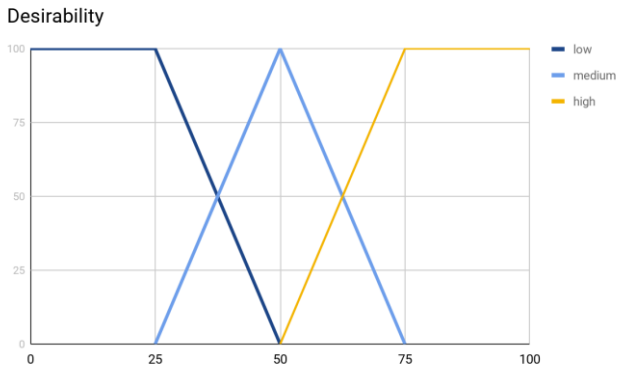
Antecedents and consequence

Survivor uses Fuzzy Logic to evaluate if it needs to eat or sleep. It is more important to eat, so attending to Hunger is prioritized. For the fuzzy logic variables (FLV) we used two antecedents, hunger and sleep, with one consequence, desirability. For each FLV we created a graph that shows the different factors, shown below.





6.4.2020



Fuzzy Rules

In combination with the FLV graphs we set out a set of rules to apply to the FLV's. Only the fuzzy AND was used within the rules, but the fuzzy OR is implemented and ready for use within the code. The rules for the fuzzy logic are as follows:

IF hungry AND sleepy THEN undesirable

IF hungry AND awake THEN desirable

IF hungry AND alert THEN very desirable

IF content AND sleepy THEN undesirable

IF content AND awake THEN desirable

IF content AND alert THEN very desirable

IF full AND sleepy THEN undesirable

IF full AND awake THEN desirable

IF full AND alert THEN very desirable

These rules are implemented within the CompositeGoal MakeDecisionGoal. In this class the fuzzy sets are created and the rules of the fuzzy logic are defined. The shapes within the fuzzy logic available are both the shoulders and the triangle. To create all the base structures for the fuzzy logic the book Programming game AI by example was used. It includes a clear view over the structure that the sets and operators use. (Buckland, M. 2005. 415-456.)

See image below.

6.4.2020

```
1 reference
public MakeDecisionGoal(AwareEntity me) : base(me)
{
    fuzzyModule = new FuzzyModule();

    FuzzyVariable hunger = fuzzyModule.CreateFLV("Hunger");
    FzSet hungry = hunger.AddLeftShoulder("Hungry", 0.0, 0.1, 0.3);
    FzSet content = hunger.AddTriangle("Content", 0.1, 0.5, 0.8);
    FzSet full = hunger.AddRightShoulder("Full", 0.5, 0.8, 1.0);

    FuzzyVariable fatique = fuzzyModule.CreateFLV("Fatique");
    FzSet sleepy = fatique.AddLeftShoulder("Sleepy", 0.0, 0.1, 0.3);
    FzSet awake = fatique.AddTriangle("Awake", 0.1, 0.3, 0.5);
    FzSet alert = fatique.AddRightShoulder("Alert", 0.3, 0.5, 1.0);

    FuzzyVariable desirability = fuzzyModule.CreateFLV("Desirability");
    FzSet unDesirable = desirability.AddLeftShoulder("Undesirable", 0, 0.25, 0.5);
    FzSet desirable = desirability.AddTriangle("Desirable", 0.25, 0.5, 0.75);
    FzSet veryDesirable = desirability.AddRightShoulder("VeryDesirable", 0.5, 0.75, 1.0);

    fuzzyModule.AddRule(new FzAND(hungry, sleepy), unDesirable);
    fuzzyModule.AddRule(new FzAND(hungry, awake), desirable);
    fuzzyModule.AddRule(new FzAND(hungry, alert), veryDesirable);
    fuzzyModule.AddRule(new FzAND(content, sleepy), unDesirable);
    fuzzyModule.AddRule(new FzAND(content, awake), desirable);
    fuzzyModule.AddRule(new FzAND(content, alert), veryDesirable);
    fuzzyModule.AddRule(new FzAND(full, sleepy), unDesirable);
    fuzzyModule.AddRule(new FzAND(full, awake), desirable);
    fuzzyModule.AddRule(new FzAND(full, alert), veryDesirable);
}
```

Test cases

To demonstrate the fuzzy logic there will follow two test cases where our rules and FLV's are used.

Test case 1

We have a survivor with the following properties: Hunger of 65 and Fatigue of 37.5.

Fatigue\Hunger	**Hungry** 0	**Content** 0.5	**Full** 0.5
Sleepy 0	0	0	0
Awake 0.5	0	0.5 desirable	0.5 desirable
Alert 0.5	0	0.5 very desirable	0.5 very desirable

Fuzzy conclusion: undesirable: 0, desirable: 0.5, very desirable: 0.5

Max very desirable: 100

Max desirable: 75

Max undesirable: 50

MaxAv =50 * 0 + 75 * 0.5 + 100 * 0.5 0 + 0.5 + 0.5 =87.51= 87.5

Test case 2

We have a survivor with the following properties: Hunger of 55 and Fatigue of 25.

6.4.2020

Fatigue\Hunger	**Hungry** 0	**Content** 0.8	**Full** 0.2
Sleepy 0.25	0	0.25 undesirable	0.2 undesirable
Awake 0.75	0	0.75 desirable	0.2 desirable
Alert 0	0	0	0

Fuzzy conclusion: undesirable: 0.25, desirable: 0.75, very desirable: 0

Max very desirable: 100

Max desirable: 75

Max undesirable: 50

MaxAv =50 * 0.25 + 75 * 0.75 + 100 * 0 0.25 + 0.75 + 0 =68.751=68.75

Conclusion

Meeting with criteria

As an end result the project meets its criteria and shows a fair amount of creativity with the theme and use of multiple subgoals and such. Project includes a castaway theme with corresponding environment and entities that rotate when moving according to their heading. Survivor mostly uses ArriveBehaviour to move, but also Seeking, Fleeing and Idle behaviours were implemented and can be used. Seagulls are another entity using Flocking and within flocking behaviours also use Wander and EnforceNonPenetrationConstraint to fly around the map without overlapping each other. Survivor uses a working path finder to walk around the island according to a graph drawn on walkable areas. Diagonal edges are more costly and survivor uses A* to calculate the shortest path around static objects around the map. Survivor uses Fuzzy Logic to evaluate its needs and knows they need to be attended to and is able to complete the goals itself without external input. Goals have been cut to composite and atomic goals and composite goals use other composite goals to achieve a more autonomous behaviour.

Technical evaluation

From a technical perspective, code is separated into smaller pieces and commented on, so it should be fairly easy to understand in the eyes of an outsider. Still, there are many ways the code could've been cut to even smaller, understandable and independent parts to make it more modular. Some values like hunger and fatigue could be tweaked and updated and increased somewhere else than in the AwareEntity class to optimize the use of game time, but for this project updating survivor's needs in the mentioned class works.

Issues and points of development in the future

Clear improvements would be fixing the arriving behaviour so that the survivor completely stops when arriving at its destination. In the current form the survivor moves fairly slow due to the arriving behaviour and to the fact that faster movement results in sloppy steering. For the seagulls flying looks good after the first minute when they get out of trying to stick together, so the first cohesion could be improved. Also without any seagulls flying close by a lonely seagull stays put as its not receiving force to move around. Another feature still missing but easily implemented would be to add static object avoiding for the survivor, so it won't walk over its tent if it somehow walks out of the graph. On the other hand this feature was not needed for now as the survivor walks calmly and faithfully along the edges of the graph. Survivor does not have a randomized behaviour, which could've been something like swimming in the ocean or socializing with a coconut to get someone to talk to.

Future insights

There were many things yet left unimplemented due to the time frame given for this assignment. With more time, the plan would be to continue the survivors goals to collect wood from the palm trees and eventually build a raft to get off the island and back to civilization. Other possible features could be a monkey to take care of keeping the survivors social meter up or like in the original Johnny Castaway, ships sailing near the island and the survivor trying to get rescued with calling for help or even receiving and sending bottle notes. All in all, this project was incredibly interesting and challenging and left a lot of inspiration to continue developing this animation further.