Algorithms & AI – Final assignment

Table of Contents

Introduction

In this paper, I will go over the process and research I have done while working on this final assignment. The chapters are chronologically ordered i.e. categorized per week & assignment part. The project is made in the C# programming language with Visual Studio 2017, and can be found in "xxx.*rar*". Please take the time to go through the settings & controls table below, some features might be hidden on default to increase the overall aesthetics of the simulation. Also note that some screenshots in this paper were made during the development process, resulting in some screenshots which may differ visually in contrast to how they look now, such as sprites and obstacles.

Settings & Controls

Key	Effect
B	Toggle the Bounding box (*on*/*off*) of obstacles with sprites enabled.
G	Toggle the Graph (*on*/*off*).
P	Pause or unpause the game world.
S	Toggle Spatial partitioning & Spatial partitioning grid (*on*/*off*).
T	Toggle between showing the goal on screen (*on*/*off*).
Settings → Cursor mode	Change between the 'Seek' and 'Debug' cursors.
Settings → Pathfinding	Toggle between the A* and Dijkstra algorithms, also includes toggles for path following such as 'Show path' which shows the generated path, 'Show visited' which shows all the visited vertices and edges during the path calculation and 'Show target' which shows the current target during the execution of the path following behaviour.
Settings → Heuristic	Toggle between the *Euclidean* and *Manhattan* heuristics, utilized by the A* algorithm.
Settings → Flocking	Toggle the non-penetration constraint (*on*/*off*), when turned on, this constraint does not allow flocking unites to overlap each other.
Settings → Spatial partitioning	Toggle the spatial partitioning grid (*on*/*off*) or show the adjacent buckets for the current debug entity (debug cursor mode required).
Spawn Agent (button)	This button allows the creation of new agents during runtime. A corresponding list of all the possible steering behaviours allows you to pick the steering behaviour of the next to-be spawned agent.

Figure 1: table which shows the settings and controls of my application

Steering behaviours

Seek

Implementing the seek behaviour was trivial and went without any major problems. The logic used to find the desired vector is as following:

$$Vector2D\ desired = target - position$$

Figure 2: calculation for the 'Seek' steering behaviour

Arrive

I have both a Seek and an Arrival steering behaviour class. My arrival steering behaviour class uses the seek steering behaviour under the hood. During certain goals such as *TraverseEdge,* either Seek or Arrive is chosen depending on whether or not the last edge is being traversed.

Flee

Implementing the flee behaviour was a lot harder than I expected. The flee behaviour itself was easy to implement, because the only change was the calculation of the desired position. Instead of calculating the target minus the position, we calculate the current position minus the target (inverted *seek*) because we want to *flee* away from the target:

$$Vector2D\ desired = position - target$$

Figure 3: calculation for the 'Flee' steering behaviour

The hardest part was to stop the flee behaviour after an arbitrary distance; this was eventually solved by gradually setting the speed of a fleeing vehicle to zero if an arbitrary distance condition was met.

Wander

Implementing the wandering was not that hard. Unfortunately I encountered a bug when combining the wandering behaviour with the pursuit behaviour. Because I had the flocking steering behaviours enabled for both the 'Pursuit' and 'Wander' behaviours, I encountered the following unwanted behaviour:

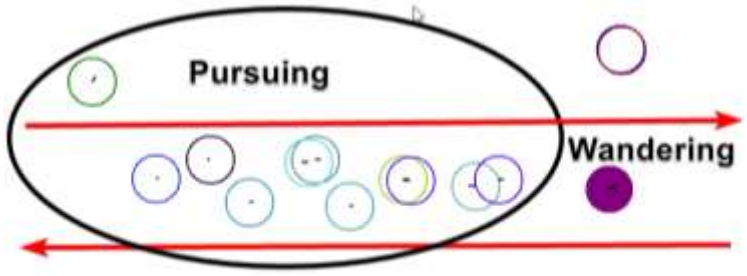


Figure 4: overview of a problem I encountered while implementing wander behaviour

At a later stage, this problem was somehow fixed when I decided to implement the non-penetration constraint.

Custom behaviour: flee & wander

To make my game world a bit more alive, I combined both the Flee and Wander behaviours into a new behaviour called *Slowpoke*. This behaviour works as following: the entity with this behaviour assigned wanders around the game world, with a very slow speed, on default. When a seek vehicle is nearby i.e. in panic distance, this behaviour will quickly duck (flee) away from the incoming seek vehicle. In a way you could argue that this behaviour has some kind of small finite state machine built in, to switch between either flee or wander behaviour.

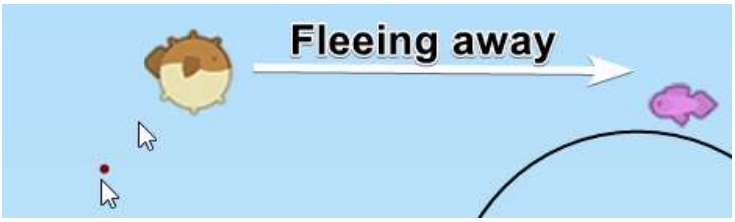


Figure 5: a target is set at the red ellipse, this in turn triggers a condition which calculates the flee forces

Pursuit

Implementing pursuit steering behaviour was trivial. Basically all I am doing is using “Seek” steering behaviour to seek the future position of a given agent. This in turn makes it look realistically because this position is constantly updated.

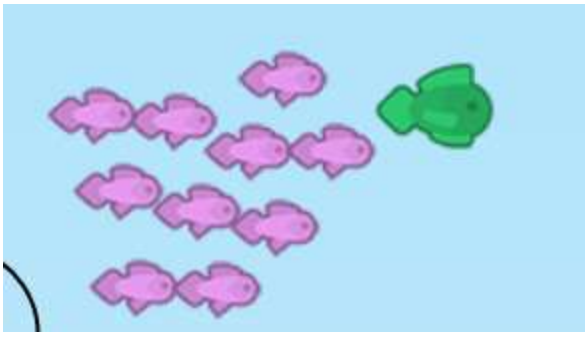


Figure 6: pursuit steering behaviour in action, showing purple fish following a green wandering fish

Explore

My implementation for this behaviour explores my game world and scans actively for an edible object. This happens by utilizing the wander steering behaviour. An agent with explore behaviour applied has radar technology implemented (as an attribute ‘RadarRadius’) which is the radius around the agent in which objects can be “smelled” (found). If an agent “smells” (finds) a given object, the mission is accomplished and the explorer resets his radar so he can look for the next object.

Figure 7: showing an exploring unit with its radius

Note: this behaviour is used in conjunction with goal driven behaviour. At first sight it might look pointless; however this behaviour is crucial to satisfy a goal such as CleanUpDeadFish.

Overview

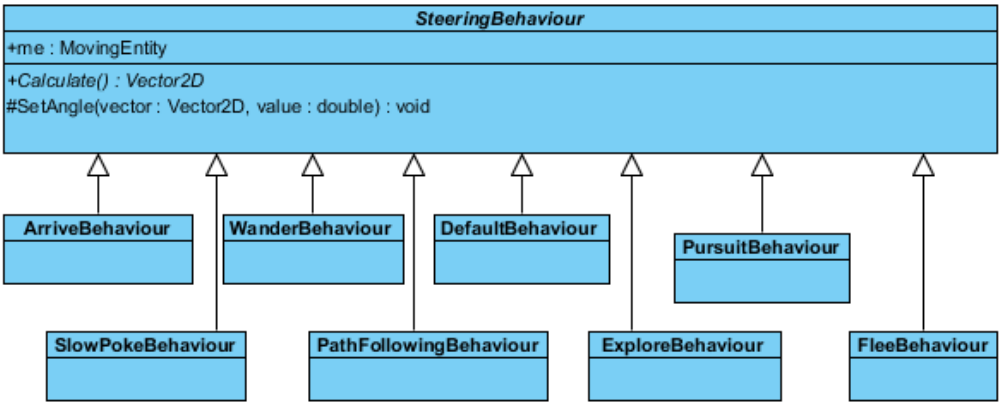


Figure 8: class diagram which shows an overview of all the implemented steering behaviour classes

Flocking behaviours

Overview

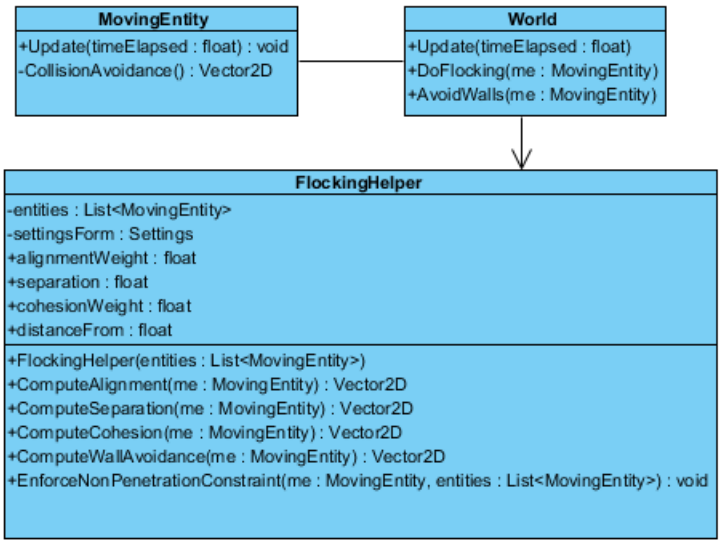


Figure 9: class diagram which shows all the flocking related classes

Calculation

I first tried the *weighted truncated sum* approach: during this approach, each steering behaviour is multiplied by its weight. While trying to tweak the settings, I noticed that it was very hard to tweak the settings with this method. I eventually found the right settings, but the flocking appearance still looked weird from time to time. Luckily, Buckland's book [1] came with a solution: *Weighted Truncated Running Sum with Prioritization*. I had implementing this solution marked as a TODO item; unfortunately due to time issues I was not able to implement this more advanced solution.

Figure 10: due to time issues I was not able to implement a more advanced calculation method, TODO's are sometimes just a nice idea on paper only

Within the update method of my world, I call *DoFlocking* for every moving entity which has flocking enabled. See the code fragment below for more details.

```
/// <summary>
/// Enable flocking on a moving entity.
/// </summary>
/// <param name="me"></param>
private void DoFlocking(MovingEntity me)
{
    if (Settings.Get("EnforceNonPenetrationConstraint"))
    {
        var entities = GetVehiclesForType(typeof(PursuitBehaviour));
        FlockingHelper.EnforceNonPenetrationConstraint(me, entities);
    }
    var alignment = FlockingHelper.ComputeAlignment(me);
    var cohesion = FlockingHelper.ComputeCohesion(me);
    var separation = FlockingHelper.ComputeSeparation(me);

    me.Velocity.X += alignment.X * FlockingHelper.AlignmentWeight + cohesion.Y *
    FlockingHelper.CohesionWeight + separation.X * FlockingHelper.SeparationWeight;

    me.Velocity.X += alignment.Y * FlockingHelper.AlignmentWeight + cohesion.Y *
    FlockingHelper.CohesionWeight + separation.Y * FlockingHelper.SeparationWeight;
}
```

Figure 11: code fragment showing my *DoFlocking* method which calls the right flocking methods for a given MovingEntity within *FlockingHelper* and calculates the updated velocity

Obstacle avoidance

Wall avoidance

In the *Update* method within my world, I call the *AvoidWalls* method on each moving entity while iterating through the list of entities. This method in turn calls the method *ComputeWallAvoidance* within the flocking helper which returns a vector to avoid walls.

Collision avoidance

Something else I implemented is collision avoidance. This means that an obstacle added to the game world can have collision with a *MovingEntity*. Within the *Update* method of my *MovingEntity* class, the following happens in a nutshell (small implementation details removed):

```
public override void Update(float timeElapsed)
{
	var steeringForce = SB.Calculate();
	steeringForce += CollisionAvoidance();

}
```

Figure 12: code fragment showing the *Update* method within my *MovingEntity* class, this method calculates the steering forces and the collision avoidance force

Non-penetration constraint

Figure 13: enabling the constraint

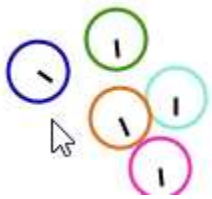


Figure 14: non-penetration constraint in action

I have added support for the non-penetration constraint. This constraint does not allow flocking agents to overlap each other, which results in non-overlapping agents (see fig 14). Here's how it works: if the calculated overlap of an arbitrary agent 1 with agent 2 is bigger than zero, the implementation method for this constraint will push this agent away from the overlapping agent by assigning a newly calculated position to agent #1's *Position* vector.

```
foreach (var entity in entities)
{
	if (entity.Equals(movingEntity))
		continue;

	var toEntity = entity.Pos.Clone() - movingEntity.Pos.Clone();
	var distFromEachOther = toEntity.Length();

	var overlap = movingEntity.Scale + entity.Scale - distFromEachOther;

	/*
	 *  If the distance is smaller than the sum of their radii,
	 *  then this entity must be moved away in the direction parallel
	 *  to the toEntity vector.
	 */
	if (overlap >= 0)
	{
		var newPos = entity.Pos.Clone() + (toEntity.Divide(distFromEachOther)) *
	overlap;
		entity.Pos = newPos;
	}

}
```

Figure 15: code fragment of my non-penetration constraint implementation in C#

Path planning

Overview

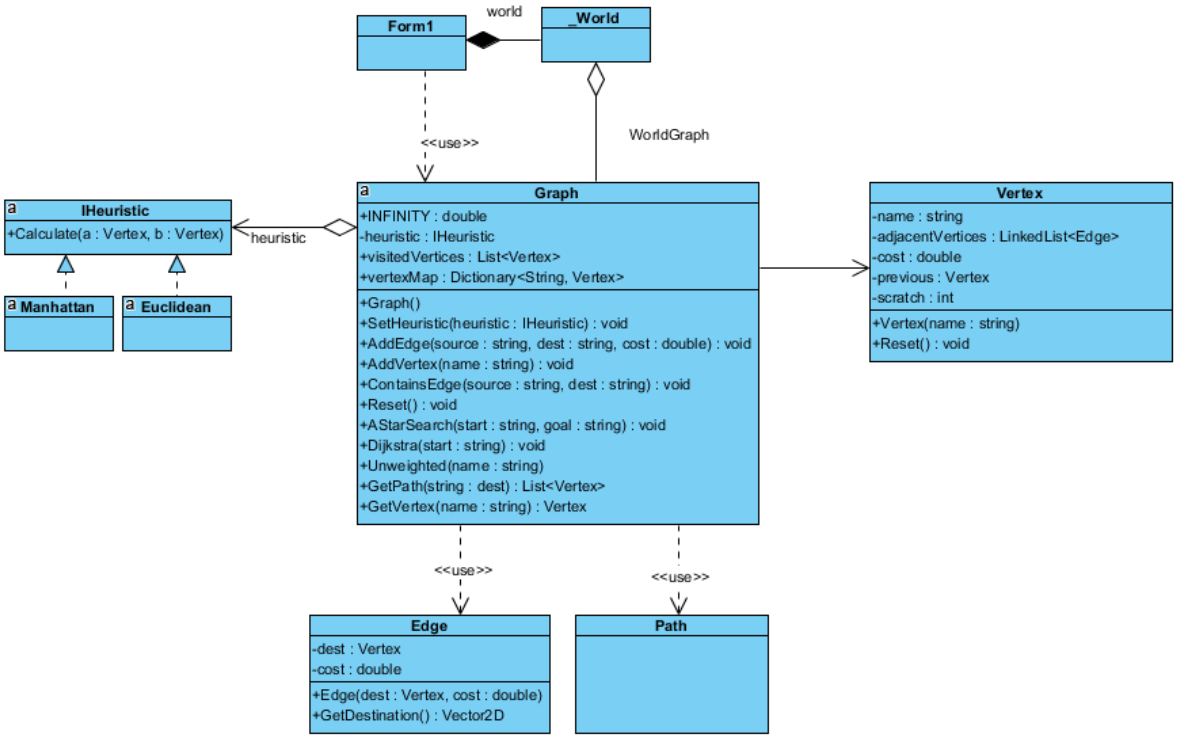


Figure 16: an overview of all my path planning related classes

Graph

Generating the graph

To generate the graph, I am using a flood-fill-like algorithm to progressively make a graph of the whole game world. To avoid obstacles which do not allow collision, I am using a function which returns the distance between 2 points, which are the current position within the algorithm and the obstacle position respectively. If the distance between a vertex and the midpoint of an obstacle is smaller than a specified value – around 1.5 times the scale of the object after tweaking – then it can be said that the vertex has, or almost has collision with the game world obstacle. Thus, the vertex is not added to the underlying graph of the whole game world.

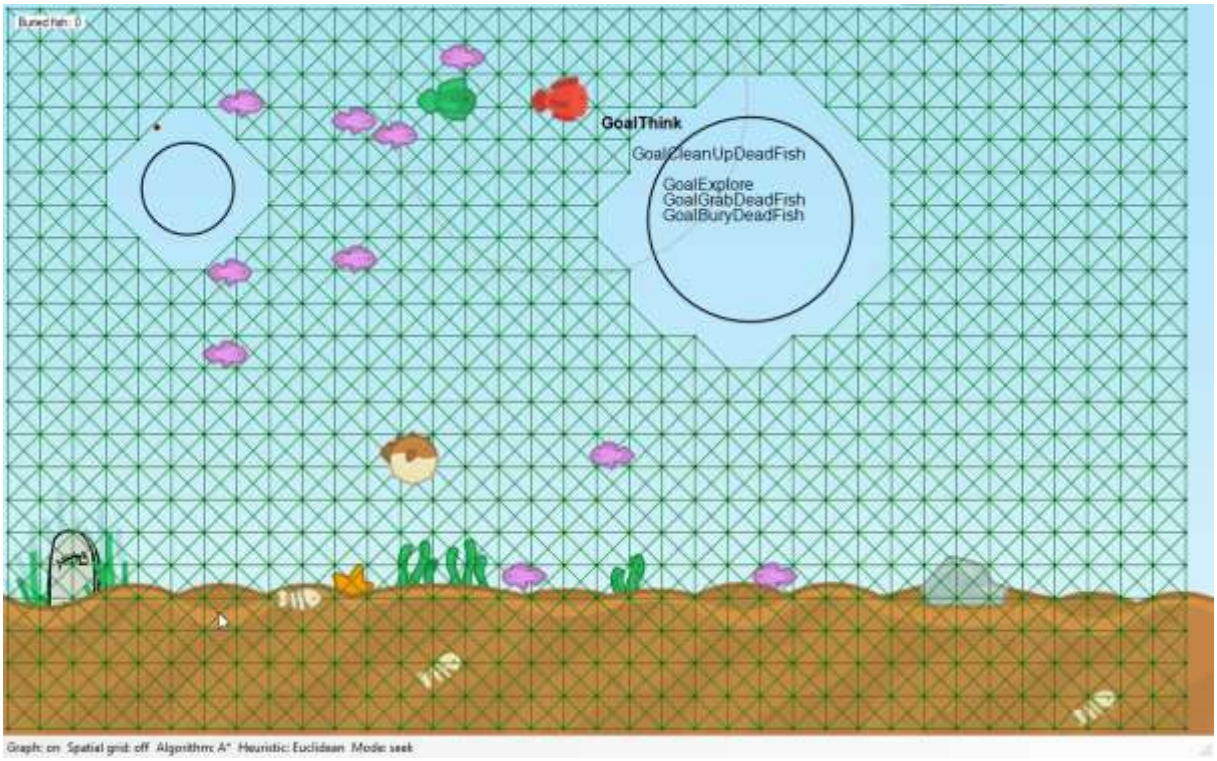


Figure 17: a flood-filled finely grained graph in my game world

Generating the edges

At a first glance, it looked like my flood-fill algorithm (same as above) was working fine. However, after trying out different pathfinding algorithms such as Dijkstra and A*, I concluded that the edges I had generated were behaving like one way (**directed**) edges whilst they should behave as **undirected** edges. This meant that finding a path downwards within my generated graph was possible, but when finding the path to a point above the current position was not working: the edges were simply missing.

I eventually solved this by redesigning my flood-fill algorithm.

- Improved the edge generating process
- Now checks if the vertex has an edge with its neighbours, if not → add the edge

Generating the path

Before going straight to implementing the A* algorithm, I wanted to implement path generation in small increments. I started with implementing Dijkstra as my first path finding algorithm. An issue I quickly ran into was that on occasion there wasn't any path available between two different points. After debugging, I noticed that my Dijkstra algorithm was missing some vertices every now and then. I eventually fixed this issue by replacing my *priority queue* with another implementation [2]. When I was done testing Dijkstra, I decided to start working on my A* algorithm.

A* differs from Dijkstra by utilizing a particular heuristic, the following equation shows this very well:

$$f(n) = g(n) + h(n)$$

Where "f" is the value which will be added to the priority queue, "g" the cost to get to that particular vertex and "h" the heuristic value of that vertex. In comparison to Dijkstra, the A* algorithm utilizes a

heuristic value whereas Dijkstra's algorithm does not include the heuristic value, "+ h(n)", hence the A* algorithm with a bad heuristic – or no heuristic at all – is actually Dijkstra.

Only a few small changes were necessary to get A* up and running:

- Implement one or more distance heuristic(s) (see below).
- Modify the code so that the heuristic value of the current vertex is added on top of the cost to get to that vertex, which ultimately will be added to the priority queue ("f").
- Check for an early exit, which means that the algorithm is done for if the **current dequeued vertex** is equal to the **target vertex**.

Heuristics

For the A* algorithm I've implemented both the Manhattan distance and Euclidean distance heuristics by utilizing the **strategy pattern**.

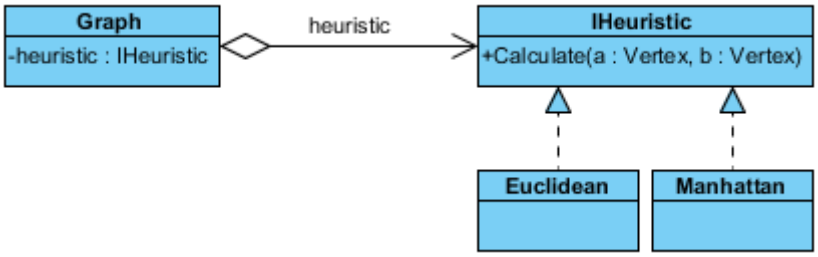


Figure 18: class diagram which shows the A* distance heuristic implemented with the strategy pattern

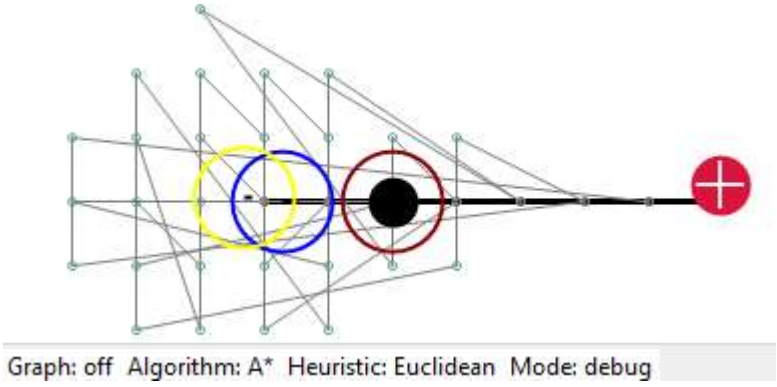


Figure 19: Euclidean distance heuristic in action, the shortest path and all visited edges/vertices are made visible

Displaying the generated graph

I ran into an issue while I was trying to show the generated graph on the screen. Because of the panel's refresh rate, the panel refreshes its content every tick. This is quite a CPU-heavy task and made the application lag because this also meant that the whole graph was redrawn every tick. I solved this issue by making the graph smaller. Placing a vertex every 24 units (x or y) made the graph substantially smaller, which eventually solved the problem.

Displaying the computed path

When *Show generated path* is turned on, the computed shortest path (through either Dijkstra or A*) is made visible in red.

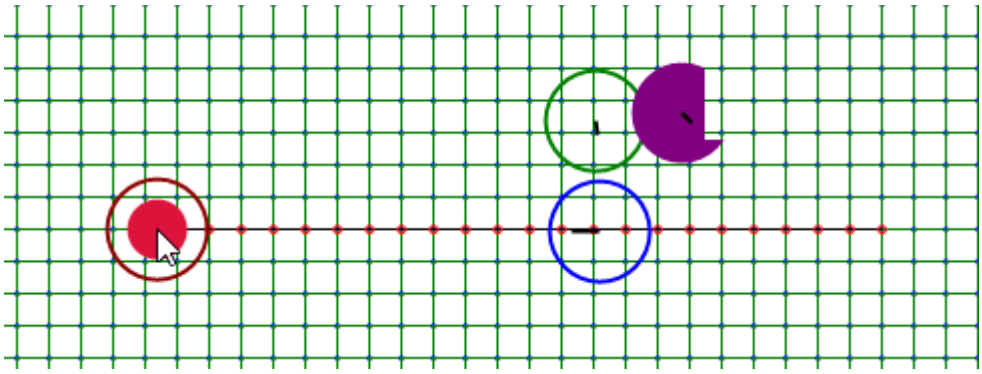


Figure 20: displaying the computed path

Path following

A big part of the path planning, especially on a visual level, comes down to following the generated path. Implementing path following took me longer than expected, mostly because I was struggling with the implementation of this steering behaviour. After digging more into path following itself I finally got a "Eureka!" moment and got the behaviour working. In a nutshell, there is a generated path which needs to be followed, this path has different point on it and the path following vehicle has to "latch" onto this path. This behaviour is implemented within the PathFollowing steering behaviour, in a later stage of the assignment I decided to allow path following only within one of my composite goals called "*FollowPath*". The steering behaviour still exists, there are however no other agents available showing this behaviour, only the main agent can execute the *FollowPath* goal by right clicking on navigable place in the game world.

Path smoothing

To make the path smoother, I included a small but simple implementation detail within the function which returns the target node to latch onto. In essence, this function looks ahead an X amount of nodes/vertices and returns this future position (waypoint). Because of this, the path following vehicle behaves a lot smoother and doesn't seem to jump around if the path is straggly, because it can "look" ahead.

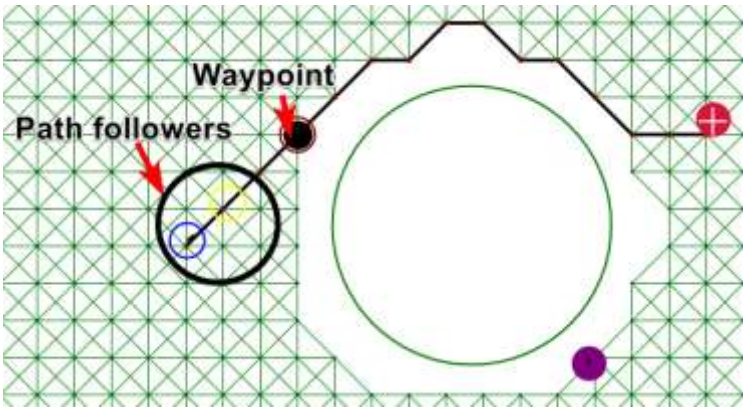


Figure 21: overview of my path smoothing implementation

To illustrate my algorithm, I have provided an annotated example to the right (fig 22). The black line with its vertices marked in red denotes the generated shortest path. Because the waypoint is ahead of the current path follower's position, the path follower seeks this waypoint, instead of the "current point + 1" approach where the line is precisely followed. This results in a slightly changed direction, because of the underlying "Seek" on the waypoint. This in turn results in a natural "smoothening" of the followed path.

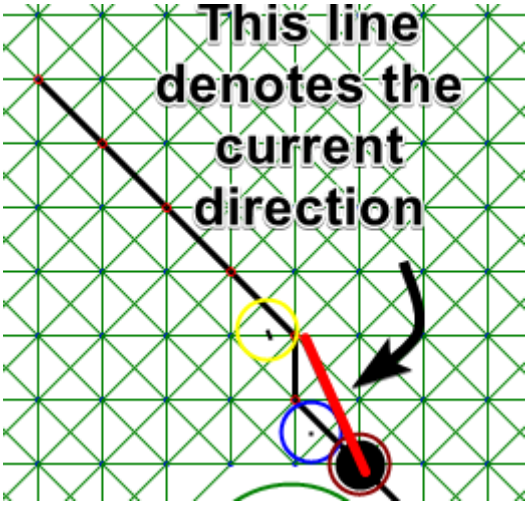


Figure 22: path smoothing in action

Because I still think my explanation thus far falls short, I've created a simple overview of the visual side of my path smoothing algorithm below:

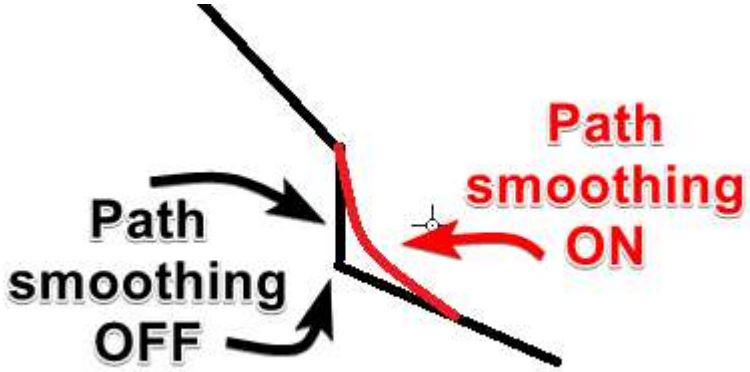


Figure 23: the result of turning path smoothing ON or OFF

Behaviour

I have chosen to implement goal-driven behaviour (week 5: assignment 1). It took me quite a while to understand how exactly goal oriented behaviour worked. After multiple re-readings of chapter 9 of Buckland's book [1] and observing Buckland's code, I finally saw the light which meant I could connect the dots between the theory and practice.

GoalThink (brain)

My behaviour is evaluated within the *Arbitrate* method of my *GoalThink*. I have created evaluators for the following behaviours:

- CleanUpFish
- ExploreGoal
- WanderGoal

Each evaluator returns a desirability as a float; unfortunately I do not really have that much of an advanced underlying logic think tank. The only thing that really stands out within my Evaluators is the following: if a dead fish is present in the game world, the *CleanUpFishEvaluator* returns the highest possible value, thus taking over any other goal that was currently ongoing. This in turn makes sure that my game world stays clean of dead fish.

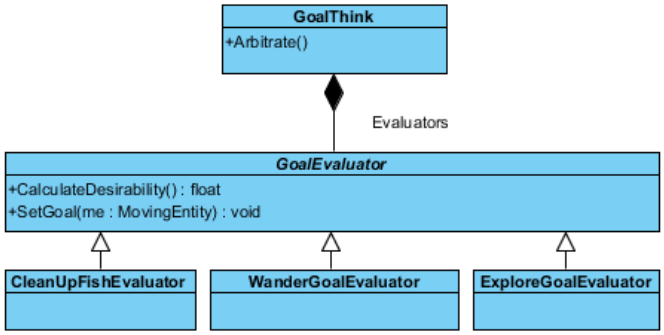


Figure 24: class diagram of implemented evaluators

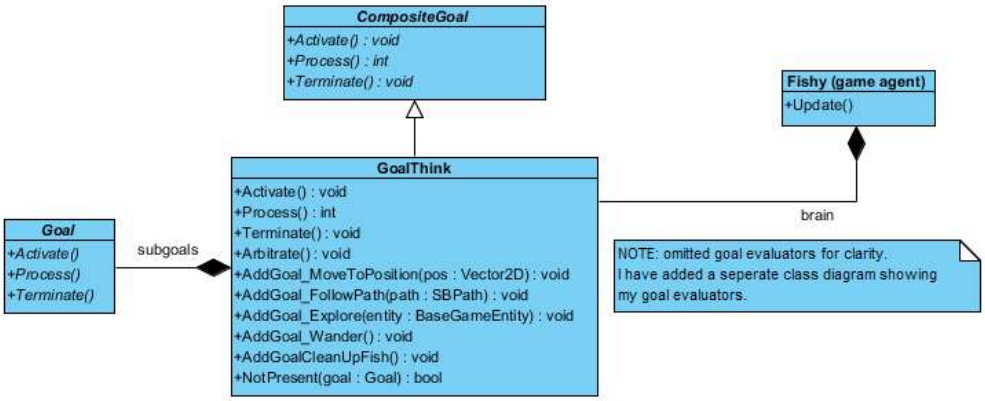


Figure 25: class diagram overview of GoalThink

Overview

First I will give an overview of all my goal classes (both atomic and composite). Afterwards, I will explain each goal on its own.

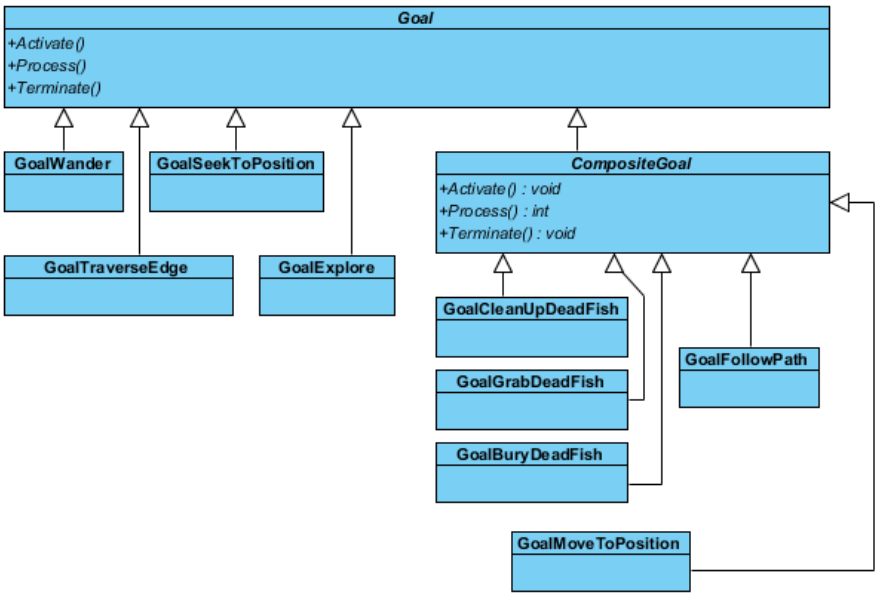


Figure 26: a class diagram giving an overview of all my goal classes, it is missing a few details but hopefully my intention is clear

Atomic goals

SeekToPosition

This goal seeks a given target (cursor) within the game world. This goal is part of a composite goal called MoveToPosition (see below). The goal itself arranges the *Activation*, *Processing* and *Termination*. Activation is done by making sure the steering behaviour of the corresponding moving entity is set to *Seek*, and the target to the given target (cursor). Processing is done by checking if the bot is stuck or if the bot has arrived at its destination.

I have expanded the *IsStuck* method with custom logic to make sure that this atomic goal keeps retrying a couple of times before terminating completely, to avoid false positives. An attribute 'TimesStuck' counts how many times the entity has been stuck, and within the *Process* method this attribute gets evaluated. If the amount of times an entity has been stuck is less than the set amount, the goal will execute the *ReactivateIfFailed* method. If the amount of time an entity has been stuck is bigger than a set amount, the goal is timed out; goal status FAILED is returned which in turn terminates this goal.

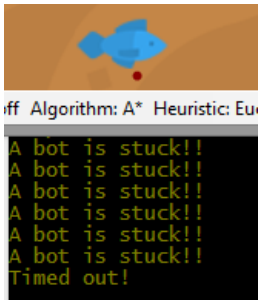


Figure 27: my custom implementation of the *IsStuck* method, showing a non-accessible target which in turn times out the goal

TraverseEdge
This goal traverses a given edge within the game world. This goal is part of a composite goal called FollowPath, which allows a moving entity to follow a calculated path along the game world graph.

Explore
This goal utilizes my custom steering behaviour *Explore*. This goal explores the game world until a given *BaseGameEntity* is found. The *Process* method consists of checking if the bot has located the given entity with the radar from the *Explore* steering behaviour.

Composite goals

MoveToPosition
I have implemented this goal according to Buckland's implementation. This goal has an atomic goal called *SeekToPosition*. When **left clicking** anywhere in my game world, my bot uses the *MoveToPosition* goal to get there.

FollowPath
This behaviour follows a given path. I have altered my implementation of this behaviour a bit to be able to utilize my custom created helper class *SBPath* which is a wrapper around a generated path.

Figure 28: showing the *FollowPath* behaviour in action

When **right clicking** anywhere in my game world, my bot uses the *FollowPath* goal to get there. Try it!

CleanUpDeadFish
The underwater world is a recurring theme in my simulation. Because I have chosen to implement fish sprites, I decided to include a goal which looks for dead fish, retrieves them and buries them. This goal itself consists of multiple composite goals: *Explore, GrabDeadFish* and *BuryDeadFish*.

GrabDeadFish
As the name implies, this goal "grabs" a dead fish. This goal is used in conjunction with the *CleanUpDeadFish* goal. This goal utilizes the *SeekToPosition* goal to position itself near the dead fish. When this goal terminates, the "CarryingFish"-Boolean is enabled which helps with further handling of this dead fish.

BuryDeadFish
This goal "buries" a dead fish at the graveyard in my game world. This goal is also used in conjunction with the *CleanUpDeadFish* goal. When this goal terminates, the "BuriedFish" count is incremented and the dead fish is removed from the game world.

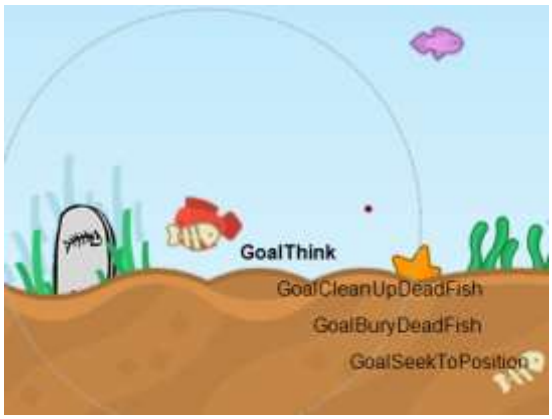


Figure 29: *BuryDeadFish* behaviour in action, showing a bot bringing a dead fish to its well-deserved resting place

Fuzzy logic

I have implemented fuzzy logic to change the way how my fish agent's perceives the world in some interactions, such as while cleaning up old dead fishbone and while searching for food. While in **reality** mode, my fish agent renders a dead fishbone as is. While in **fantasy** mode, my fish agent can renders a dead fishbone as a fish. While in **outer space** mode, my agent sees things notably differently.

Fuzzy linguistic variables

Distance to Target (antecedent 1)

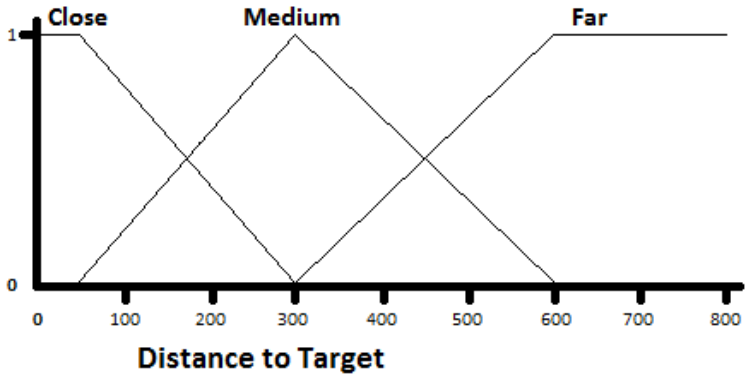


Figure 30:: FLV graph for 'Distance to Target' antecedent

Amount of Hunger (antecedent 2)

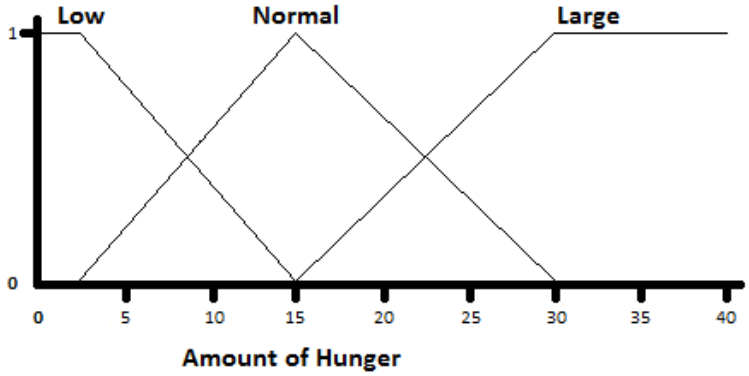


Figure 31: FLV graph for 'Amount of Hunger' antecedent

Perception (consequent)

I decided to have a *perception* consequent. This means an agent's perception changes based on his level of hunger and distance to target. During *reality*, an agent perceives things how they really are. During *fantasy,* an agent perceives some things differently such as *fishbone*, this will be rendered as a normal fish during this state of perception. Finally, *outer space* is when an agent totally loses control and sees things really in a different way.

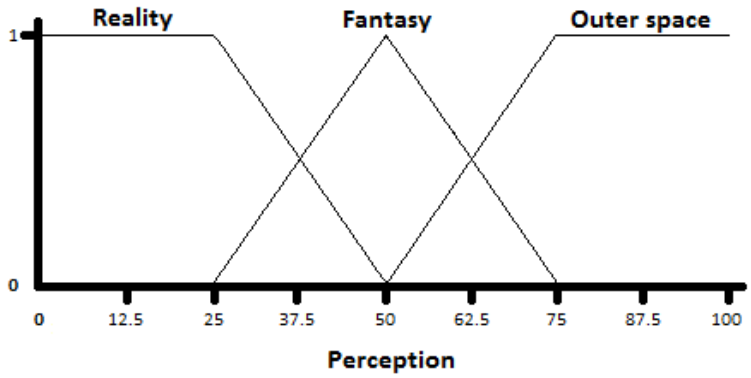


Figure 32: FLV graph for 'Perception' consequent

Fuzzy rules set

	Antecedent 1 (Distance to Target)		Antecedent 2 (Amount of Hunger)		Consequent (Perception)
IF	*Target_Far*	**AND**	*Amount_Large*	**THEN**	OUTER SPACE
	Target_Far		*Amount_Normal*		FANTASY
	Target_Far		*Amount_Low*		REALITY
	Target_Medium		*Amount_Large*		OUTER SPACE
	Target_Medium		*Amount_Normal*		FANTASY
	Target_Medium		*Amount_Low*		REALITY
	Target_Close		*Amount_Large*		FANTASY
	Target_Close		*Amount_Normal*		REALITY
	Target_Close		*Amount_Low*		REALITY

Figure 33: the final design of my fuzzy rules set

Test cases

Test case 1

Input

Name	Value
Distance to Target (antecedent 1)	450
Amount of Hunger (antecedent 2)	20

Figure 34: input variables for test case 1

Fuzzyfication

Distance to Target	
$F_{close}(450)$	0
$F_{medium}(450)$	0.50
$F_{far}(450)$	0.50
Amount of Hunger	
$F_{low}(20)$	0
$F_{normal}(20)$	0.65
$F_{large}(20)$	0.35

Figure 35: fuzzification of input variables for test case 1

Fuzzy associative matrix

→ Distance to target ↓Amount of hunger	**CLOSE**	**MEDIUM**	**FAR**
LOW	REALITY MIN(0, 0) = **0.0**	REALITY MIN(0.5, 0) = **0.0**	REALITY MIN(0.5, 0) = **0.0**
NORMAL	REALITY MIN(0, 0.65) = **0.0**	FANTASY MIN(0.5, 0.65) = **0.5**	FANTASY MIN(0.5, 0.65) = **0.5**
LARGE	FANTASY MIN(0, 0.35) = **0.0**	OUTER SPACE MIN(0.5, 0.65) = **0.5**	OUTER SPACE MIN(0.5, 0.35) = **0.5**

Figure 36: fuzzy associative matrix filled with values after fuzzification test case 1

Inferred conclusions

Consequent	Value
Reality	MAX(0, 0, 0, 0) = **0**
Fantasy	MAX(0.5, 0.5, 0) = **0.5**
Outer space	MAX(0.5, 0.5) = **0.5**

Figure 37: inferred conclusion test case 1

Defuzzification with MaxAv

Maxima	Calculation	Result
Max Reality	(0 + 25) / 2	12.5
Max Fantasy	50 (center)	50
Max Outer space	(75 + 100) / 2	87.5

Figure 38: calculating maxima for each state of perception for test case 1

$$MaxAv = \frac{12.5 * 0 + 50 * 0.5 + 87.5 * 0.5}{0 + 0.5 + 0.5} = 68.75$$

Figure 38: MaxAv scaled average calculation for test case 1

Test case 2

Input

Name	Value
Distance to Target (antecedent 1)	700
Amount of Hunger (antecedent 2)	5

Figure 39: input variables for test case 2

Fuzzyfication

Distance to Target	
$F_{close}(450)$	0
$F_{medium}(450)$	0
$F_{far}(450)$	1.0
Amount of Hunger	
$F_{low}(20)$	0.77
$F_{normal}(20)$	0.23
$F_{large}(20)$	0

Figure 40: fuzzification of input variables for test case 2

Fuzzy associative matrix

→ Distance to target ↓Amount of hunger	**CLOSE**	**MEDIUM**	**FAR**
LOW	REALITY MIN(0, 0.77) = **0.0**	REALITY MIN(0, 0.77) = **0.0**	REALITY MIN(1.0, 0.77) = **0.77**
NORMAL	REALITY MIN(0, 0.23) = **0.0**	FANTASY MIN(0, 0.23) = **0.0**	FANTASY MIN(1.0, 0.23) = **0.5**
LARGE	FANTASY MIN(0, 0) = **0.0**	OUTER SPACE MIN(0, 0) = **0.0**	OUTER SPACE MIN(1.0, 0.0) = **0.0**

Figure 41: fuzzy associative matrix filled with values after fuzzification test case 2

Inferred conclusions

Consequent	Value
Reality	MAX(0, 0, 0.77, 0) = **0.77**
Fantasy	MAX(0, 0.5, 0) = **0.5**
Outer space	MAX(0, 0) = **0**

Figure 42: inferred conclusion test case 2

Defuzzification with MaxAv

Maxima	Calculation	Result
Max Reality	(0 + 25) / 2	12.5
Max Fantasy	50 (center)	50
Max Outer space	(75 + 100) / 2	87.5

Figure 43: calculating maxima for each state of perception for test case 2

$$MaxAv = \frac{12.5 * 0.77 + 50 * 0.5 + 87.5 * 0}{0.77 + 0.5 + 0} \approx 27.26$$

Figure 44: MaxAv scaled average calculation for test case 2

Overview

Unfortunately, due to a lack of time I was not able to create a class diagram of my fuzzy classes. Please check the classes within my project, I did spent a lot of time implementing and creating it.

Within the *Fishy* class, I initialize the fuzzy module with two antecedents and one consequent in *InitializeFuzzyModule* method. I also add multiple fuzzy rules in here. Within my *HandleFuzzyLogic* method, I handle the fuzzy logic during each game tick.

```
private void InitializeFuzzyModule()
{
    FuzzyVariable DistToTarget = FuzzyModule.CreateFLV("DistToTarget");
    FzSet Target_Close = DistToTarget.AddLeftShoulderSet("Target_Close", 0, 50, 300);
    FzSet Target_Medium = DistToTarget.AddTriangularSet("Target_Medium", 50, 150, 300);
    FzSet Target_Far = DistToTarget.AddRightShoulderSet("Target_Far", 300, 600, 1000);

    FuzzyVariable Hunger = FuzzyModule.CreateFLV("Hunger");
    FzSet Large = Hunger.AddRightShoulderSet("Hunger_Lots", 10, 30, 100);
    FzSet Normal = Hunger.AddTriangularSet("Hunger_Normal", 0, 10, 30);
    FzSet Low = Hunger.AddTriangularSet("Hunger_Low", 0, 0, 10);

    FuzzyVariable Perception = FuzzyModule.CreateFLV("Perception");
    FzSet Reality = Perception.AddRightShoulderSet("Reality", 50, 75, 100);
    FzSet Fantasy = Perception.AddTriangularSet("Fantasy", 25, 50, 75);
    FzSet OuterSpace = Perception.AddLeftShoulderSet("Outer space", 0, 25, 50);
}
```

Unfortunately, due to the time constraint I could not get fuzzy logic fully working due to unexpected errors. Although I do have implemented all the necessary base classes to handle fuzzy logic. The bigger picture is available and I hope my test cases and other features make up for not implementing this feature completely.

Extra's

Debug mode

Eventually I stumbled upon the problem of debugging individual vehicles or entities during runtime. After a short thought process I came up with a feature which would make it possible to see the information of a chosen (by cursor) entity which is present within the world of the simulation.

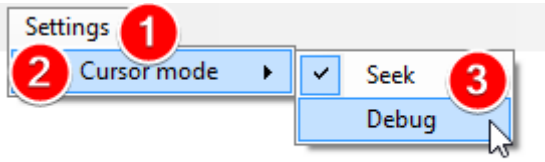


Figure 45: enabling the debug mode

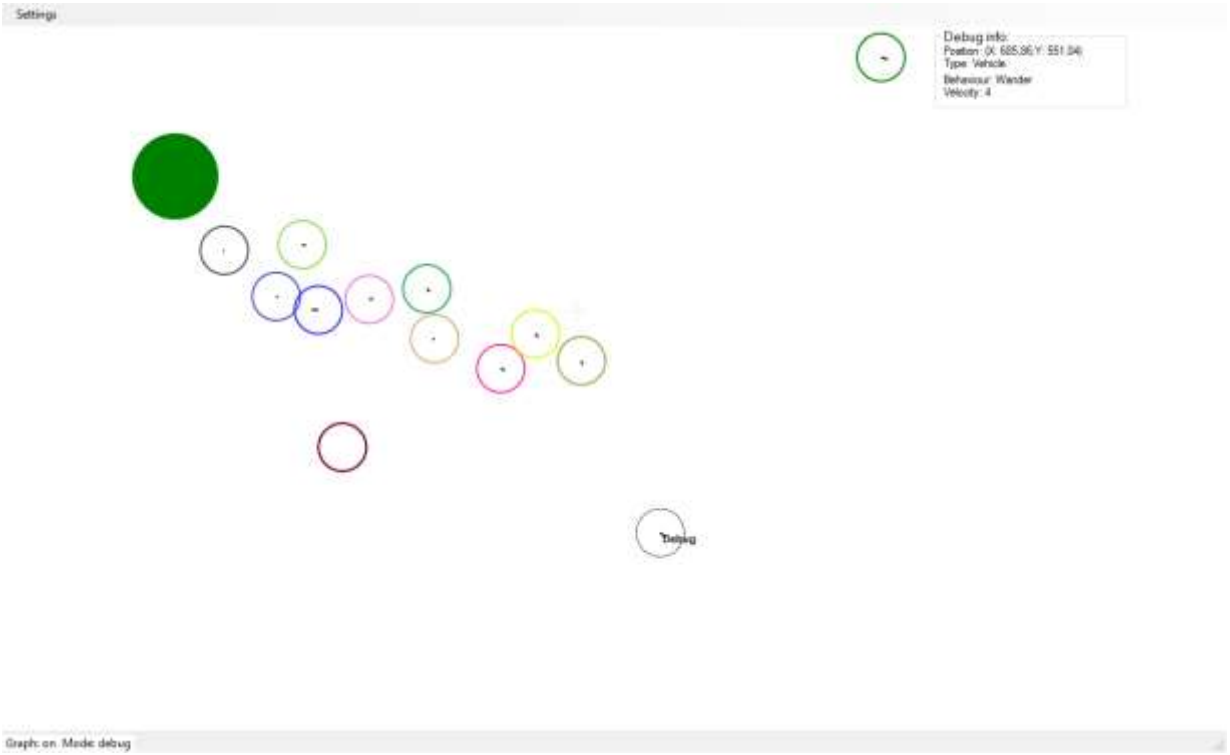


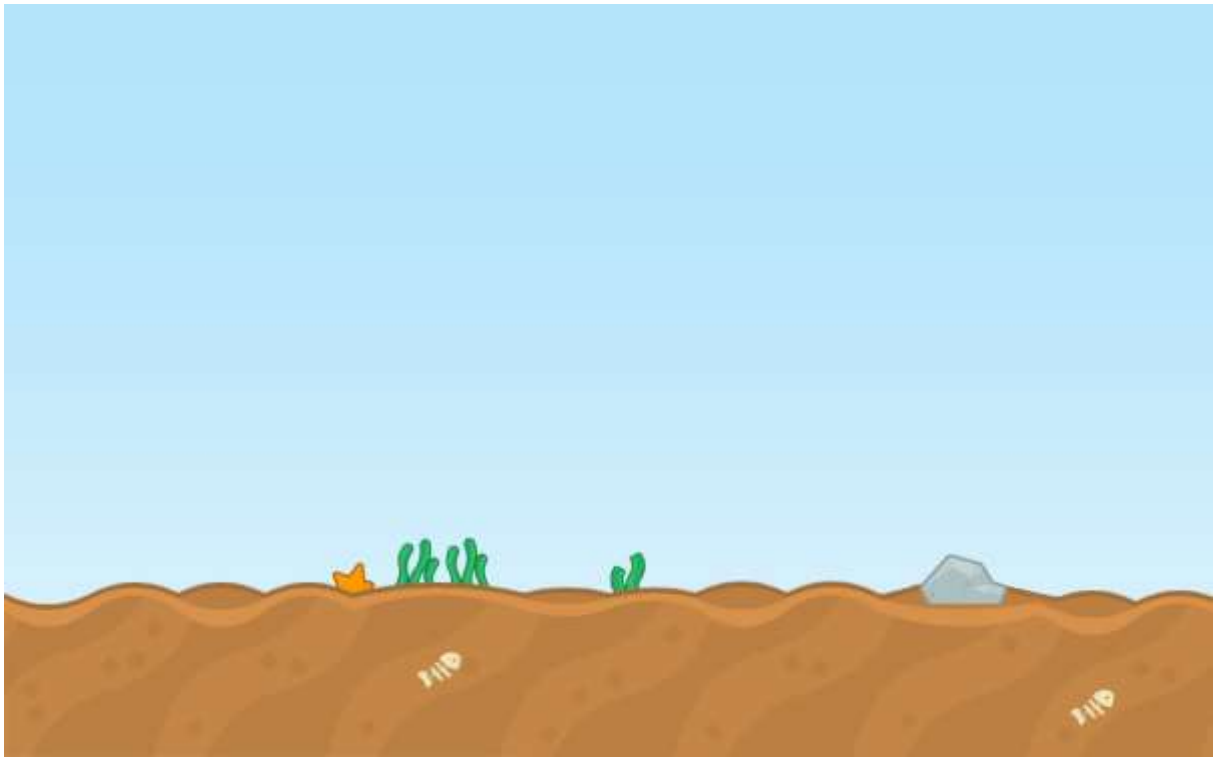
Figure 46: debug mode in action

Sprites

I decided to improve the looks and ambiance of my game world by utilizing sprites. I have used several different resources found online to create a small underwater world.

My game world exists of a background made in Photoshop (see figure below) and a plethora of other underwater sprites to create an underwater theme.

Figure 47: game world background which I have made in Photoshop



All entities

I have added support for (rendering) sprites by utilizing the strategy pattern. Because of utilizing this pattern, it makes my code a lot more maintainable and in turn easy to add new sprites. When populating my game world with new agents, a strategy of the type `ISpriteMode` can be chosen. Whilst inside the `Render` method of a class derived from `BaseGameEntity`, the following code handles the rendering of a sprite for a given entity:

```
public override void Render(Graphics g)

{

        SpriteStrategy.RenderSprite(g, this);

        // *snip*
}
```

Figure 48: code fragment which shows how a sprite is rendered for a given entity

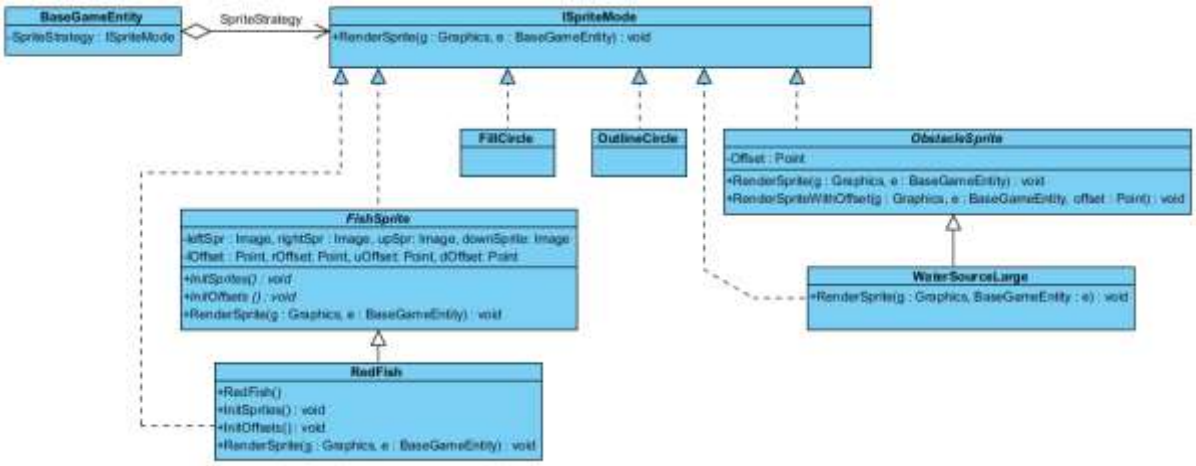


Figure 49: class diagram of my implementation which supports sprite rendering by utilizing the strategy pattern

Obstacles

An extension to the existing solution was needed for static obstacle sprites. Because each obstacle can have a different sprite, the standard position settings of an obstacle render a sprite out of bounds. To fix this, I had to come up with a generic and reusable way of rendering sprites with an offset for a given obstacle.

Figure 50: sprite rendered with the wrong (default) offset

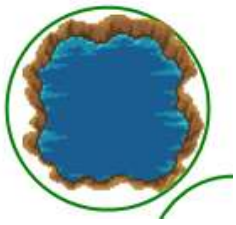


Figure 51: sprite rendered with the right offset

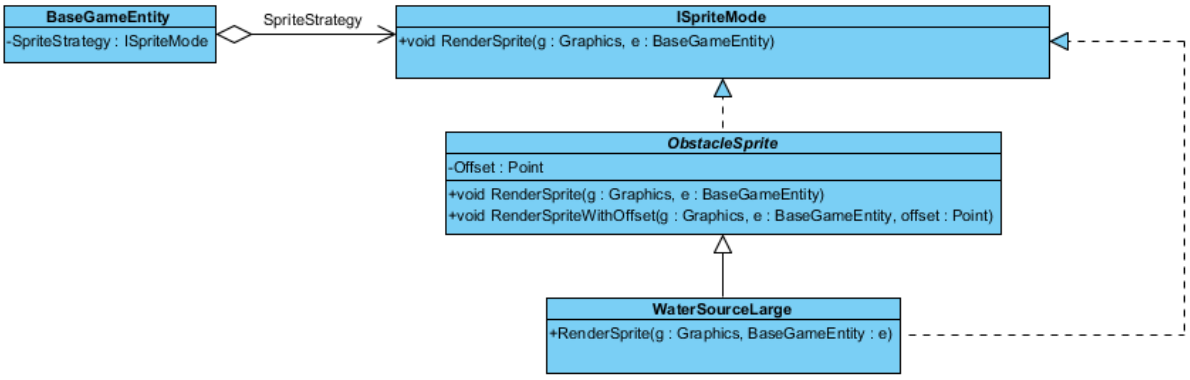


Figure 52: class diagram of my solution for rendering sprites for obstacles with a given offset (all non-related classes omitted)

My solution for this problem is an *abstract* base class `ObstacleSprite` which implements `ISpriteMode`. The `ObstacleSprite` class contains the logic to render a sprite through its implemented RenderSprite method, such that every class which derives from this abstract class complies with `ISpriteMode`. The *RenderSprite* method within `ObstacleSprite` uses the attribute 'Offset' while drawing the image. The *RenderSpriteWithOffset* method is a driver method for calling the *RenderSprite* method from outside in a derived concrete class with a given offset, such as `WaterSourceLargeSprite.`

`WaterSourceLargeSprite` extends `ObstacleSprite` and implements `ISpriteMode`, thus complying fully with `ISpriteMode` while being able to call the *RenderSpriteWithOffset* method in its parent class `ObstacleSprite`. Concrete classes which derive from `ObstacleSprite` fully comply with the rendering method specified in figure 48.

```
public class WaterSourceLargeSprite : ObstacleSprite, ISpriteMode
    {
        public new void RenderSprite(Graphics g, BaseGameEntity e)
        {
            int scale = (int)e.Scale;
            RenderSpriteWithOffset(g, e, new Point(-scale + 10, -scale + 5));
        }
    }
```

Figure 53: code fragment of a resulting concrete obstacle sprite class

Moving entities

Because moving entities can have multiple directions, I had to come up with a different way of render sprites to accompany the changing directions and offsets. I quickly noticed myself violating a DRY principle: copy & pasting the code to render a sprite, and only changing the code to initialize offsets and sprite image locations. To prevent this violation, I have created an abstract base class `FishSprite` which has properties for the sprites and offset, abstract initializer methods for both the sprites and offset and of course the *RenderSprite* method, which utilizes the initialized properties, to comply with `ISpriteMode`. Any concrete class deriving from `FishSprite` now only has to override (concretize) the initializer methods, such that all properties are initialized before calling `base.RenderSprite(g, e)`.

The result is a flexible and extendible set-up which does not violate the DRY-principle anymore.

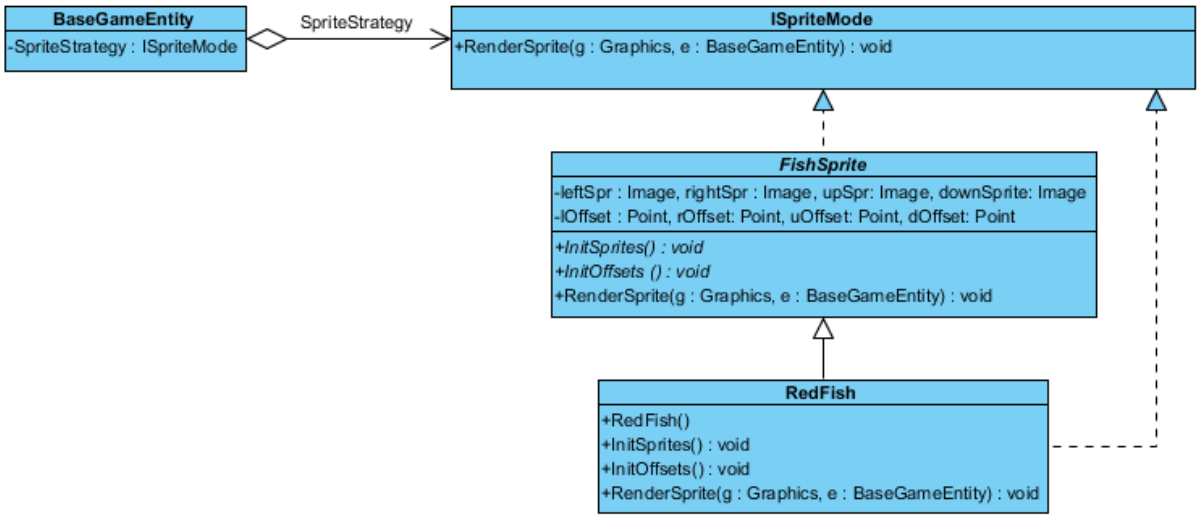


Figure 54: class diagram of my solution for rendering sprites for moving entities with a given offset (all non-related classes omitted)

Spatial partitioning

As an extra challenge – and optimization for my simulation – I have implemented spatial partitioning. The gist of this comes down to dividing the game world in one *grid* consisting of multiple *buckets*. Each bucket is updated during runtime such that each entity gets added to their respective bucket. Because calculating the destined bucket for each entity is trivial, this is not a resource intensive task and definitely outweighs the performance benefit created by utilizing spatial partitioning [3].

- Solved intersection by calculating if a circular obstacle intersects with a bucket's rectangle.
- Check current bucket – together with all the 8 adjacent buckets – to check for collision.
- Change the underlying Linked List data structure to a List to receive better concurrency support.

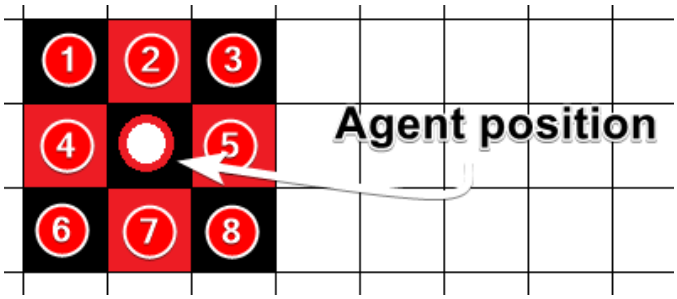


Figure 55: all surrounding buckets for a given agent

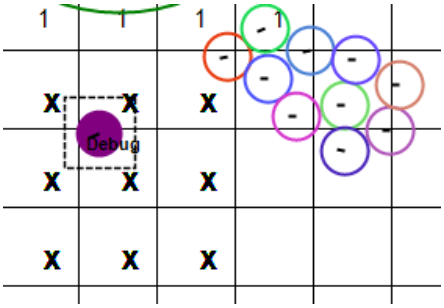


Figure 56: live display of surrounding buckets for a given agent

Encountered problems

Collision

Because the list of surrounding buckets (see figures above) is not that big, a narrower line of sight regarding incoming obstacles exists. This in turn resulted into problems where agents were colliding with static obstacles for a brief amount of time. While implementing spatial partitioning, I had not changed anything related to the collision avoidance, thus my immediate hypothesis was that the obstacles were spotted too late, resulting in a small moment of collision.

Upon further inspection in my collision avoidance class, I noticed that a `MAX_SEE_AHEAD` constant was set at 25 units. In a non-spatial partitioning setting this might have worked fine, however in my case when spatial partitioning was enabled, it certainly was not working properly. My hypothesis was correct and I eventually fixed this issue by increasing the `MAX_SEE_AHEAD` constant to 40 units.

Concurrency problems

Because the entities in each bucket are updated each game tick, I ran into concurrency problems. This resulted in errors such as looping with a *foreach*-loop through a modified collection. Most of it was solved by *thread locking* certain loops, and by using *for*-loops instead of using *foreach*-loops.

```
lock(buckets)
{
	foreach (var bucket in buckets)
	{
		lock (bucket.Entities)
		{
			for (int i = 0; i < bucket.Entities.Count; i++)
			{
				// *snip*
			}
		}
	}
}
```

Figure 57: code fragment which shows how I handled concurrency problems

Conclusion

Difficulties: path following, steering behaviours in general, flocking, spatial partitioning, goal driven behaviour

Late to the party: I first tried to make everything with the tutorials from *Envato Tuts+*, unfortunately those implementations are way simpler, which resulted in conflicting code/ideas when discussing the theory with my peers and researching the theory through online sources. Eventually I (re)discovered the book of Buckland and that helped a tremendous amount.

Goal driven behaviour: lots of ideas, really hard to implement

Fuzzy logic: quickly had an idea, execution was hard. I also had to redesign my consequent FLV graph because it was not possible to calculate maxima within my designed graph.

Spatial partitioning: fun, challenging and informative to implement, I probably should not have been working on this before I had not even finished goal driven behaviour & fuzzy logic though.

Time spent: because of my eye for detail, I really want to do everything perfect. This takes a lot of time, it also did not help that I decided to do this assignment on my own. In hindsight, it is definitely an assignment for two people. ☺

Possible improvements: spatial partitioning can be further optimized; path smoothing can be redone so it also works with the *FollowPath* goal.

Works Cited

[1] M. Buckland, Programming Game AI by Example, Wordware Publishing, Inc., 2015.

[2] J. McCaffrey, „Priority queues with C#,” [Online]. Available: https://visualstudiomagazine.com/articles/2012/11/01/priority-queues-with-c.aspx. [Geopend 15 March 2018].

[3] R. Nystrom, „Spatial Partition - Optimization,” n.d. [Online]. Available: http://gameprogrammingpatterns.com/spatial-partition.html. [Geopend 30 march 2018].

[4] mackworth, „Stackoverflow,” [Online]. Available: https://stackoverflow.com/questions/22135712/pygame-collision-detection-with-two-circles/22135814#22135814. [Geopend 11 March 2018].

[5] envatotuts, „Understanding Steering Behaviours,” envatotuts, 2012-2013. [Online]. Available: https://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732. [Geopend 8, 9, 10, 11 March 2018].

[6] F. Bevilacqua, „Understanding Steering Behaviors: Path Following,” envatotuts, [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-

path-following--gamedev-8769. [Geopend 12, 14, 15 March 2018].

[7] Red Blob Games, „Implementation of A*," [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/implementation.html#csharp. [Geopend 15 March 2018].

[8] D. Shiffman, „7.6: Checking Objects Intersection Part I - p5.js Tutorial," 2015.

[9] Kenney, „Kenney - Fish pack," [Online]. Available: https://kenney.nl/assets/fish-pack. [Geopend 04 04 2018].