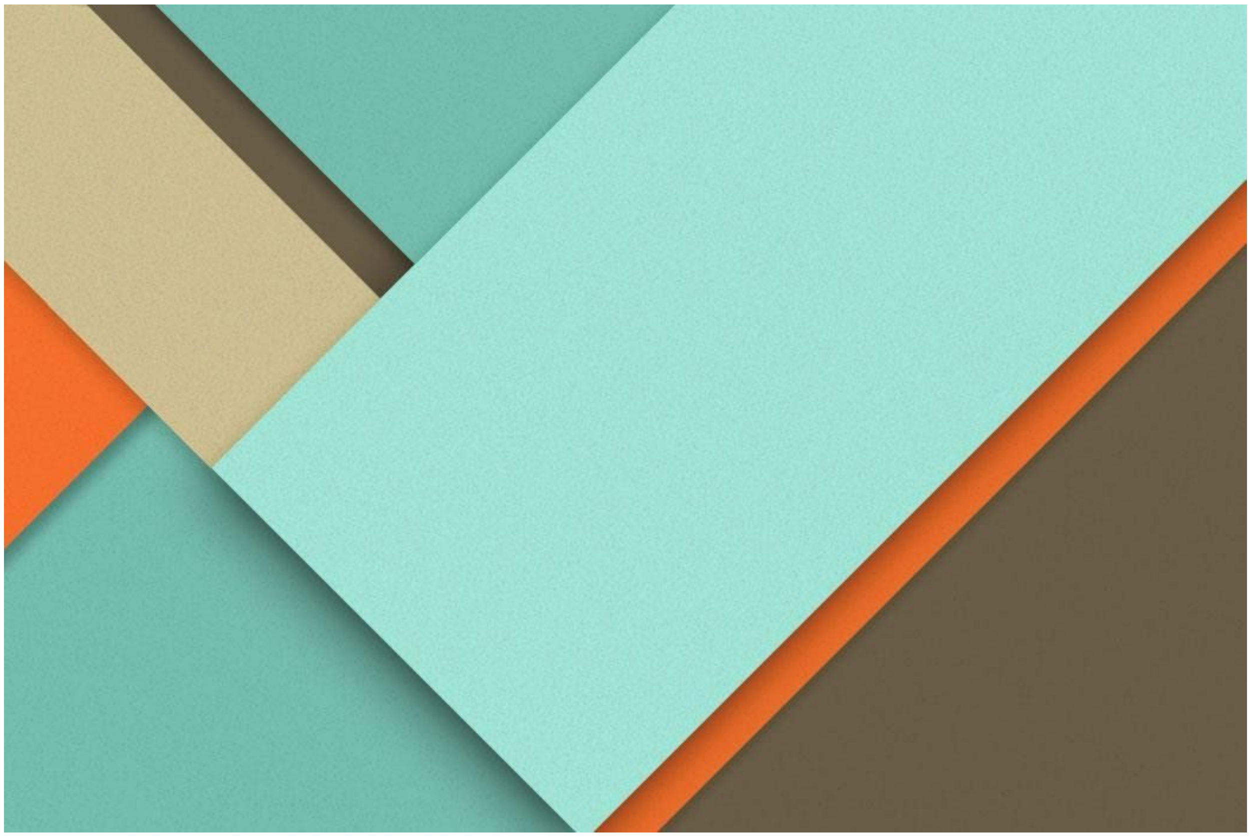


Simulator Report

28-03-2020

—

Buster Bosma
Felix Beuving

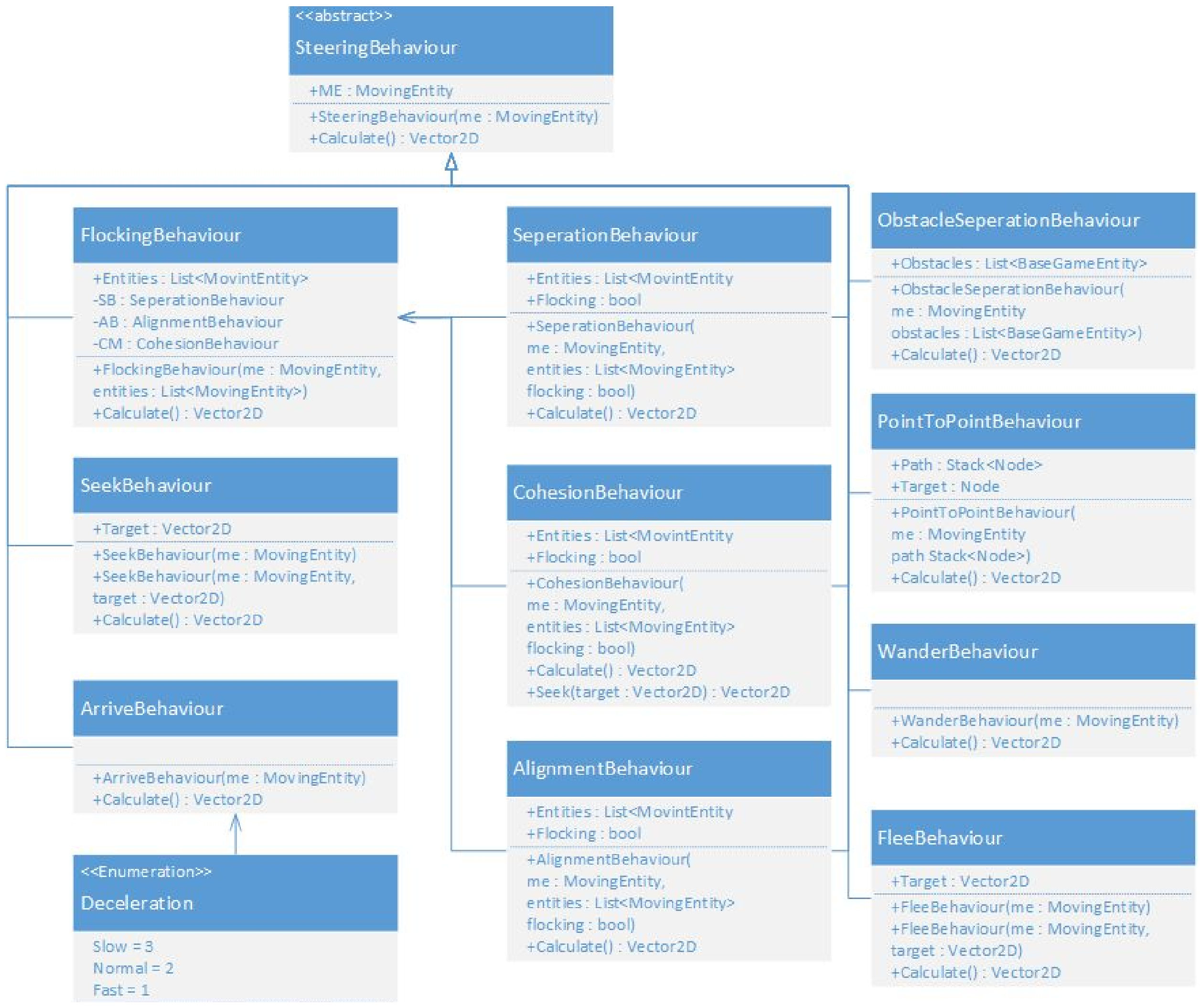
Table of contents

Introduction

In this document we will provide additional information about our Game AI Simulator.
The simulator has basic moving entities that simulate game behaviours (such as flocking, wandering, seeking, obstacle avoidance and much more). In the menu that can be found on the left-hand side of the screen a set of parameters can be modified to alter the entities' behaviour. The application also includes an A* pathfinding algorithm, a target can be set by clicking somewhere in the world, the algorithm will then find a path to the selected point. Finally, there is a Battle Royale gamemode where game agents will fight to be the last one alive.

1. Steering

This chapter will explain all the steering behaviour used in our simulator. All behaviours explained in this chapter are based on the code from the book.[1]
All of the behaviours mentioned, except for point to point, can be turned on using the checkboxes in the simulator. The forces of these behaviours can also be changed in the forces tab within the simulator.



<<abstract>>
SteeringBehaviour
+ME : MovingEntity
+SteeringBehaviour(me : MovingEntity)
+Calculate() : Vector2D
FlockingBehaviour
+Entities : List<MovintEntity>
-SB : SeperationBehaviour
-AB : AlignmentBehaviour
-CM : CohesionBehaviour
+FlockingBehaviour(me : MovingEntity, entities : List<MovingEntity>)
+Calculate() : Vector2D
SeperationBehaviour
+Entities : List<MovintEntity
+Flocking : bool
+SeperationBehaviour(me : MovingEntity, entities : List<MovingEntity> flocking : bool)
+Calculate() : Vector2D
ObstacleSeperationBehaviour
+Obstacles : List<BaseGameEntity>
+ObstacleSeperationBehaviour(me : MovingEntity obstacles : List<BaseGameEntity>)
+Calculate() : Vector2D
SeekBehaviour
+Target : Vector2D
+SeekBehaviour(me : MovingEntity)
+SeekBehaviour(me : MovingEntity, target : Vector2D)
+Calculate() : Vector2D
CohesionBehaviour
+Entities : List<MovintEntity
+Flocking : bool
+CohesionBehaviour(me : MovingEntity, entities : List<MovingEntity> flocking : bool)
+Calculate() : Vector2D
+Seek(target : Vector2D) : Vector2D
PointToPointBehaviour
+Path : Stack<Node>
+Target : Node
+PointToPointBehaviour(me : MovingEntity path Stack<Node>)
+Calculate() : Vector2D
WanderBehaviour
+WanderBehaviour(me : MovingEntity)
+Calculate() : Vector2D
ArriveBehaviour
+ArriveBehaviour(me : MovingEntity)
+Calculate() : Vector2D
AlignmentBehaviour
+Entities : List<MovintEntity
+Flocking : bool
+AlignmentBehaviour(me : MovingEntity, entities : List<MovingEntity> flocking : bool)
+Calculate() : Vector2D
FleeBehaviour
+Target : Vector2D
+FleeBehaviour(me : MovingEntity)
+FleeBehaviour(me : MovingEntity, target : Vector2D)
+Calculate() : Vector2D
<<Enumeration>>
Deceleration
Slow = 3
Normal = 2
Fast = 1

[1] "Programming game ai by example by Mat Buckland"

1.1. Seek behaviour

This behaviour returns a desired velocity vector based on the target that is given. With this velocity the entity will move towards the target. This behaviour also has the option to stop the entity when in a certain radius of the target.

1.2. Flee behaviour

The flee behaviour is basically the opposite of the seek behaviour. A target is given and the entities will try to move away from that targets position.

1.3. Arrive behaviour

The arrive behaviour is the same as the seek behaviour. The only difference is that the closer the entities get to the target the slower the move.

1.4. Point to point behaviour

Point to point behaviour is used for the path following. The behaviour gets a stack of nodes and pops the first node and makes the entity move towards this node. When the entity gets within a certain radius, a new node is popped from the stack. It does this until the stack is empty and has thus reached the end of the path.
The use of the point to point behaviour will be further explained in chapter 2.

1.5. Obstacle separation behaviour

Obstacle separation causes entities to be pushed away from a given list of obstacles. The entities are basically separated, like with the separation behaviour, from the given obstacle when in a certain radius.

1.6. Flocking behaviour

The flocking behaviour creates, like the name implies, a flocking behaviour where the entities move in unison. Like a flock of birds or a school of fish.
To create this effect the flocking behaviour uses three behaviours: Separation, Cohesion and Alignment.

Separation causes the entity to move away from each other when within a certain radius. Cohesion does the opposite and pulls entities together. And alignment causes entities to move in the same direction.

A problem we encountered while making the flocking behaviour was that the entities would only go into groups and then not move. This was an issue in the combining of the 3 behaviours. Which we solved by creating the SetMagnitude function in the vector2d class. This function normalizes the vector and multiplies it by the magnitude (in this case the max speed).

1.7. Wandering behaviour

This behaviour returns a random velocity towards a random target. The target is within a certain radius and distance of the entity. This causes a wander effect as the entity will travel to random points in the world.

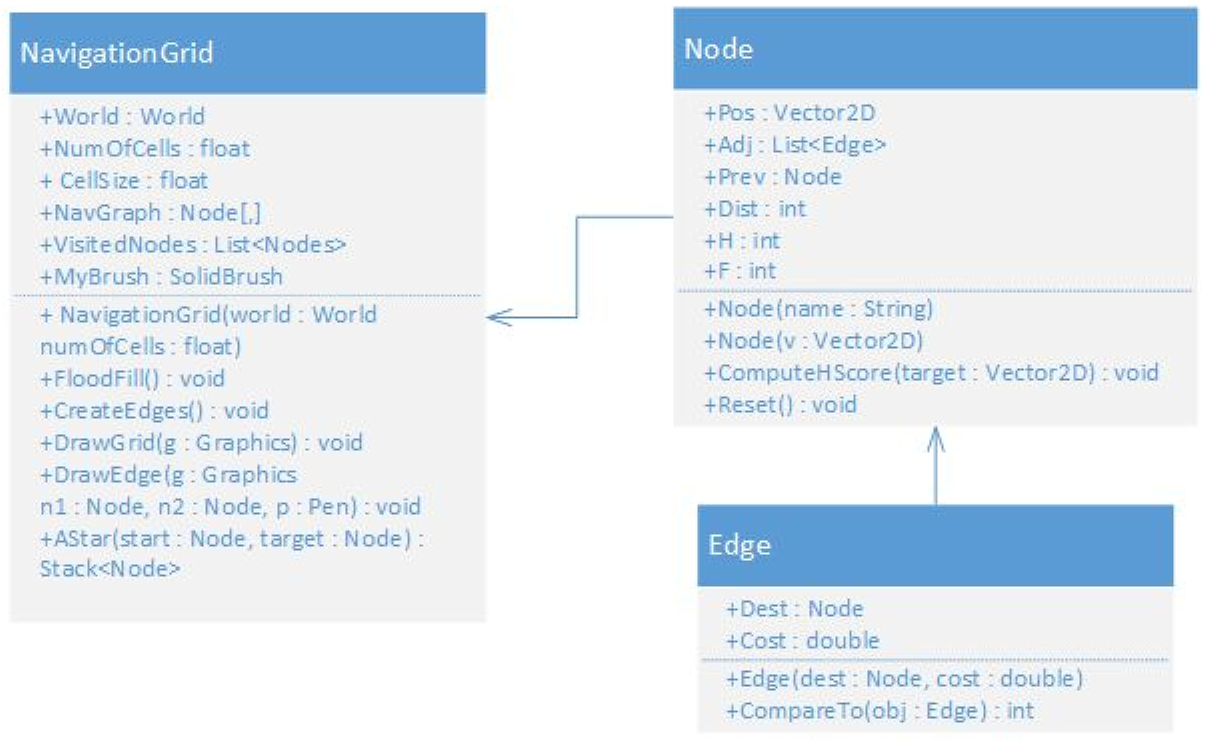
1.8. Combining behaviours

All moving entities have a list of steering behaviours. When the update function is called all these steering behaviours are calculated and added to the steering force. Then the position is updated using the updatePosition function. This function will also check if the position of the entity is off the screen and will do a wrapAround. Causing the entity to appear on the other side of the screen.

In our simulator the user can activate multiple behaviours using checkboxes. When one of these checkboxes is clicked all entities will get this behaviour in their steeringBehaviour list. Causing the entity to follow this behaviour.

2. Path Planning

This chapter will explain how we implemented path planning in our simulator.
It will do that by explaining how the navigational grid is generated and how the A* algorithm is used to calculate the shortest path.

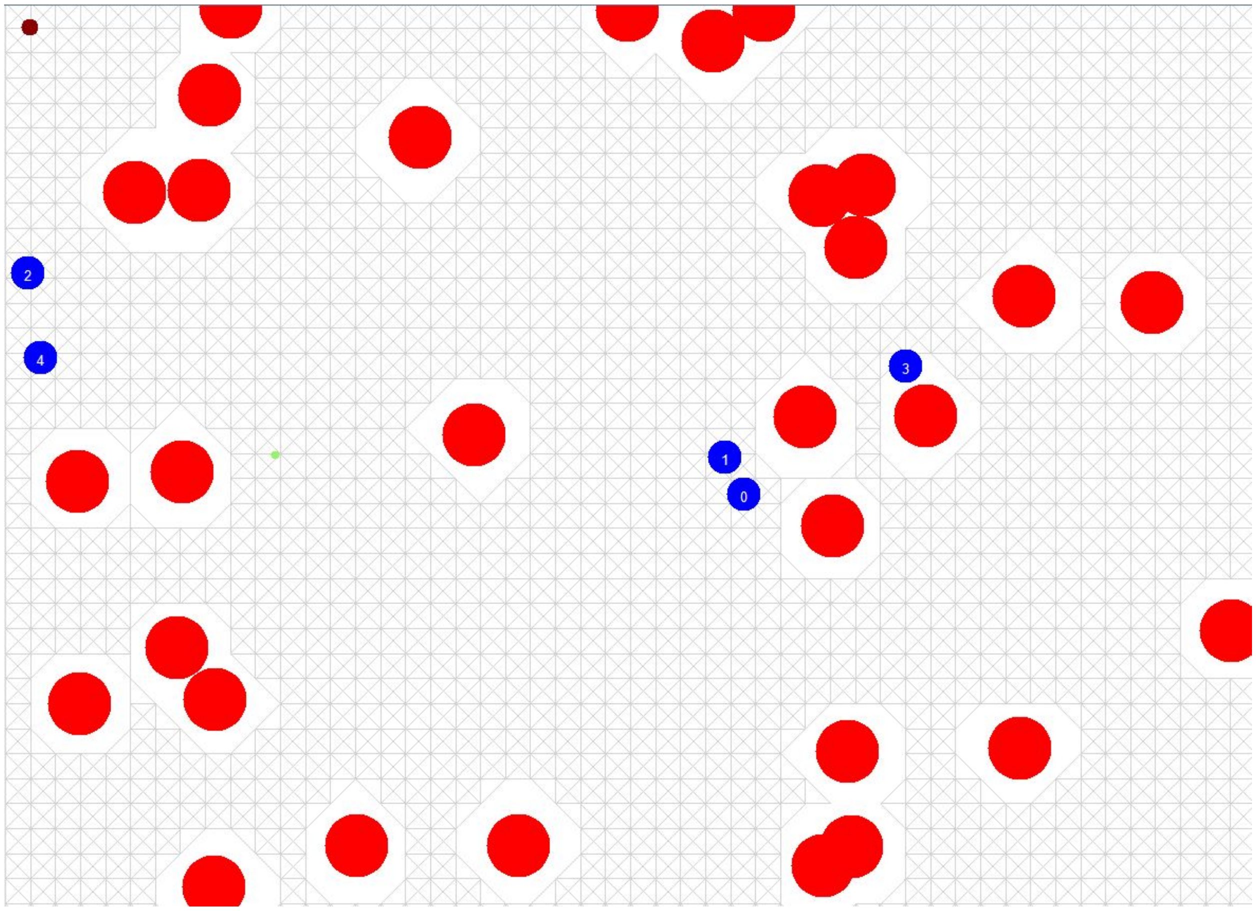

NavigationGrid
+World : World
+NumOfCells : float
+ CellSize : float
+NavGraph : Node[,]
+VisitedNodes : List<Nodes>
+MyBrush : SolidBrush
+ NavigationGrid(world : World numOfCells : float)
+FloodFill() : void
+CreateEdges() : void
+DrawGrid(g : Graphics) : void
+DrawEdge(g : Graphics n1 : Node, n2 : Node, p : Pen) : void
+AStar(start : Node, target : Node) : Stack<Node>
Node
+Pos : Vector2D
+Adj : List<Edge>
+Prev : Node
+Dist : int
+H : int
+F : int
+Node(name : String)
+Node(v : Vector2D)
+ComputeHScore(target : Vector2D) : void
+Reset() : void
Edge
+Dest : Node
+Cost : double
+Edge(dest : Node, cost : double)
+CompareTo(obj : Edge) : int

2.1. Navigational grid

The "navigationGrid" class handles the creation of the navigation grid and calculates the shortest path using A*.

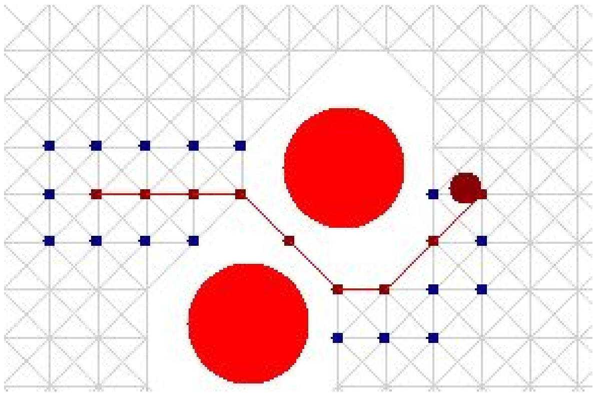
When the navigationGrid constructor is called, a grid will be created using the flood fill method. This method checks if every node is within a certain radius of an obstacle. If the node is not too close to an obstacle it is added to the "NavGraph", which is a 2 dimensional array of nodes.
After all the nodes are selected the edges between these nodes are added, an edge is basically a connection between nodes.
When this is done the navigational grid is completed and will have no nodes too close to obstacles.

The figure below shows the drawn result of the created grid.



2.2. A*

The A* algorithm calculates the shortest distance to a target. Our A* algorithm uses the Manhattan distance as its heuristic and returns a stack of nodes. This stack of nodes functions as the path the entity can follow towards its target.



In the figure you can see this path visualized. The red line and nodes are the path and the blue nodes are the nodes that the a* algorithm checked.

As a basis for our A* algorithm we used the explanation on this site: [2]

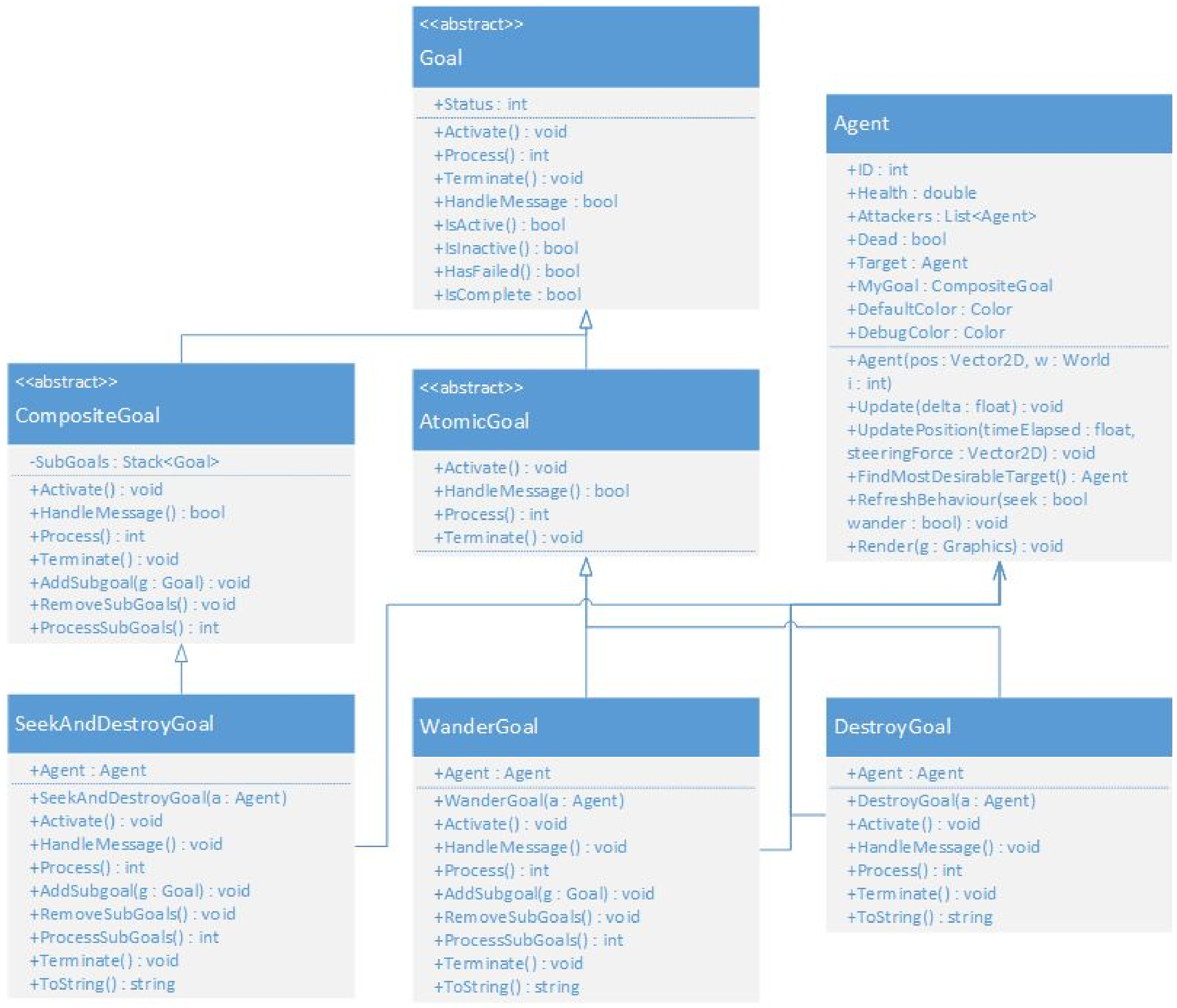
[2] https://gigi.nullneuron.net/gigilabs/a-pathfinding-example-in-c/

2.3. Path following

In our simulator the only entity that uses path following is the target entity. This target entity is used for the seek- and arrive behaviours.
When the user clicks on the screen the closest node is selected as the target for A*. When the shortest path is calculated the target entity will move along this path using the point to point behaviour.

3. Behaviour

This chapter will elaborate on the game agents and their behaviours. As seen in the class diagram a goal driven behaviour is used. It's not standard implementation which would start with a think goal, because the agent in this case has only one ultimate goal, to be the last one alive. To accomplish this it must seek enemies and destroy them. How that works is explained in this section.



3.1 Agents

Aside from the entities, the application also has game agents. These agents are more complex than normal entities because they are goal driven. The agent is the same as a normal entity but it also has a health property and a goal which it will try to achieve.

These agents have a debug mode in which the user can see their current goals with text and by color of the agent. These modes can be toggled in the sidebar. In this menu the health of each agent can also be monitored.

3.2 Goals

By default an agent has a SeekAndDestroy goal, this is a composite goal which has more subgoals. The SeekAndDestroy goal starts seeking a goal by default, if it can't find any goals it will add a WanderGoal to the sub goals. While wandering the agent will switch to seeking if it detects a nearby enemy agent. While it has the seeking goal active it will move towards the selected target using the earlier described SeekBehaviour. Once it gets close enough it will start damaging the target, if the target dies or the target gets out of range the destroy goal will terminate itself and the agent will get a new seek goal (unless it can't find a target, in which case it will default to wandering).

An agent can be targeted by multiple attackers at once, the damage it takes every game cycle is multiplied by the amount of attackers. The damage value is also slightly randomized to prevent agents from constantly killing each other at the same time.

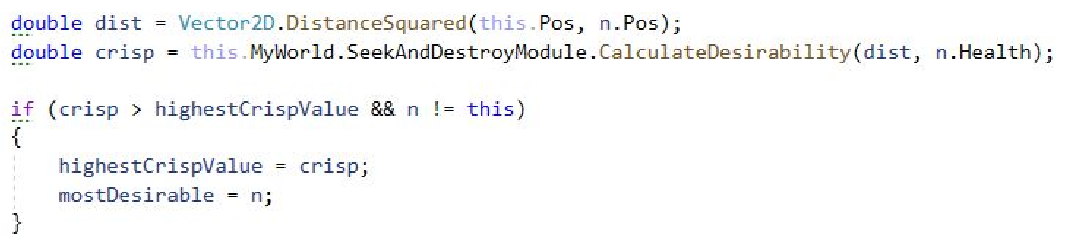
3.3 Battle Royale

Using these goals a Battle Royale gamemode was created, it's basically a 'free for all and the last agent to be alive wins'. In the sidebar there is a seperate tab for this gamemode, it can be paused and reset (these functions will only affect the agents, not the normal entities). The health and amount of attackers is also displayed here.

4. Fuzzy Logic

The fuzzy logic we use is based on the code in the book: “Programming game ai by example by Mat Buckland”. The classes are the same as in this book.

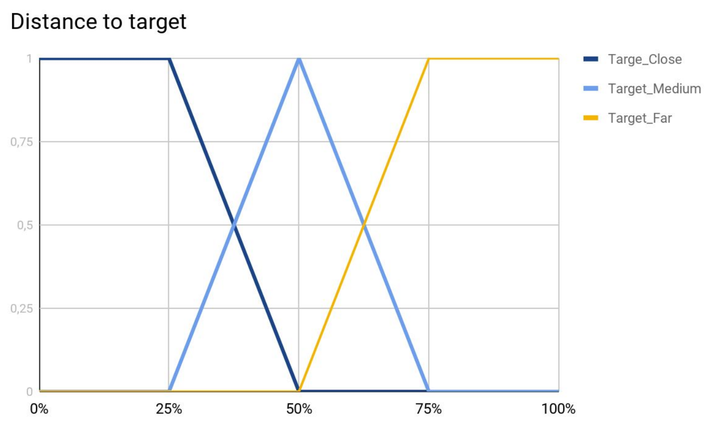
Fuzzy logic is used in our battle royale simulation for the seek and destroy goal. When a agent is seeking a target it will calculate the desirability of every agent within a certain radius. The agent with the highest desirability is chosen.

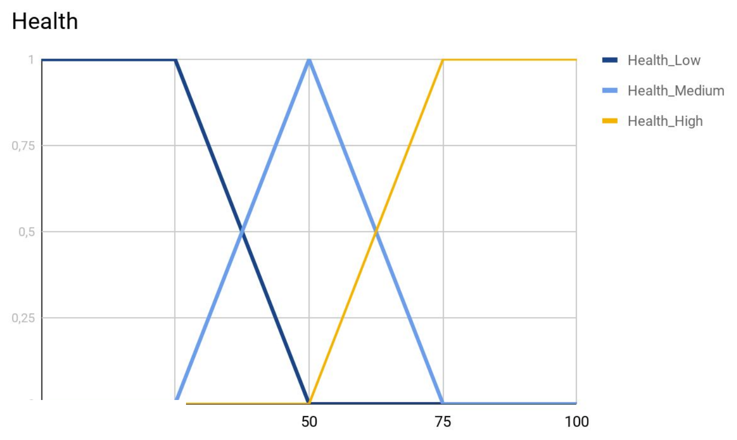


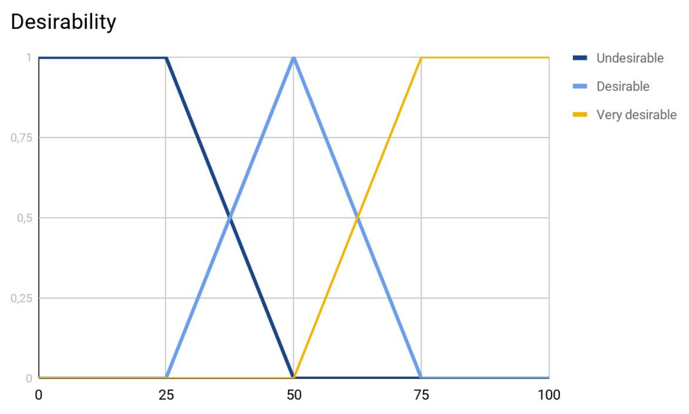
```
double dist = Vector2D.DistanceSquared(this.Pos, n.Pos);
double crisp = this.MyWorld.SeekAndDestroyModule.CalculateDesirability(dist, n.Health);

if (crisp > highestCrispValue && n != this)
{
    highestCrispValue = crisp;
    mostDesirable = n;
}
```

The fuzzy logic uses two antecedents and one consequent FLV.

The two antecedents are the distance to the target and the target health.
The max health of an agent is 100. The max distance a target can be is the value “AgentSeekDistance”. When a target is outside of this range the desirability will be zero.
The FLV for the distance to target will use 25, 50 and 75% of this value.
The consequent we use is the FLV desirability.







For the fuzzy logic we also designed a set of rules:

1. **IF Target_Far AND Health_High THEN Undesirable**
2. **IF Target_Far AND Health_MediumTHEN Undesirable**
3. **IF Target_Far AND Health_Low THEN Desirable**
4. **IF Target_Medium AND Health_High THEN Undesirable**
5. **IF Target_Medium AND Health_Medium THEN Desirable**
6. **IF Target_Medium AND Health_Low THEN Very desirable**
7. **IF Target_Close AND Health_High THEN Desirable**
8. **IF Target_Close AND Health_Medium THEN Very desirable**
9. **IF Target_Close AND Health_Low THEN Very desirable**

4.1 Test cases

We will calculate two test cases with different values for distance to target and health. The max distance to target will be 100 for both of these test cases.

4.1.1 Test case 1

Distance to target = 80

Health = 10

Fuzzy Associative matrix:

	Target_Close	Target_Medium	Target_Far
Health_Low	Very desirable : 0	Very desirable : 0	desirable : 1
Health_Medium	Very desirable : 0	Desirable : 0	Undesirable : 0
Health_High	Desirable : 0	Undesirable : 0	Undesirable : 0

Calculate maxima

•Max Undesirable: (0 + 25)/2 = 12.5

•Max Desirable: 50

•Max VeryDesirable: (75 + 100)/2 = 87.5

MaxAv

(12.5 * 0 + 50 * 1 + 87.5 * 0) / (0 + 1 + 0) = **50**

4.1.2 Test case 2

Distance to target = 40

Health = 60

Fuzzy Associative matrix:

	Target_Close	Target_Medium	Target_Far
Health_Low	Very desirable : 0	Very desirable : 0	desirable : 0
Health_Medium	Very desirable : 0.4	Desirable : 0.6	Undesirable : 0
Health_High	Desirable : 0.4	Undesirable : 0.4	Undesirable : 0

Calculate maxima

•Max Undesirable: (0 + 25)/2 = 12.5

•Max Desirable: 50

•Max VeryDesirable: (75 + 100)/2 = 87.5

MaxAv

Undesirable = 0.4

Desirable = 0.6

Very desirable = 0.4

(12.5 * 0.4 + 50 * 0.6 + 87.5 * 0.4) / (0.4 + 0.6 + 0.4) = **50**

5. Conclusion

In the end we managed to create a nice simulator which can emulate various game behaviours and we even got a gamemode working where game agents can find each other. We think the end result is pretty cool and even though there are some flaws, we are happy with the result.

We could improve the battle royale by letting agents use pathfinding instead of the normal seek algorithm. The current implementation of goal driven behaviour is also not completely according to the book. Instead of having a Think goal class we made a SeekAndDestroy composite goal and used that as the basis for the other goals. We feel a think goal would be unneeded because the agents are trying to win, and for that they will always have to default to seeking a target and destroying it until they are the last one alive.

6. Sources

1 : Book : “Programming game ai by example by Mat Buckland”

2 : https://gigi.nullneuron.net/gigilabs/a-pathfinding-example-in-c/

3 : https://docs.microsoft.com/en-us/dotnet/csharp/

4 : https://stackoverflow.com/questions/1348080/convert-a-positive-number-to-negative-in-c-sharp

5 : https://stackoverflow.com/questions/186891/why-use-the-ref-keyword-when-passing-an-object

6 : https://stackoverflow.com/questions/30063061/how-can-i-make-a-transparent-tabpage

7 : https://stackoverflow.com/questions/15131779/resize-controls-with-form-resize

8 : https://stackoverflow.com/questions/14339875/how-to-create-transparent-element
hosts-in-winforms-wpf