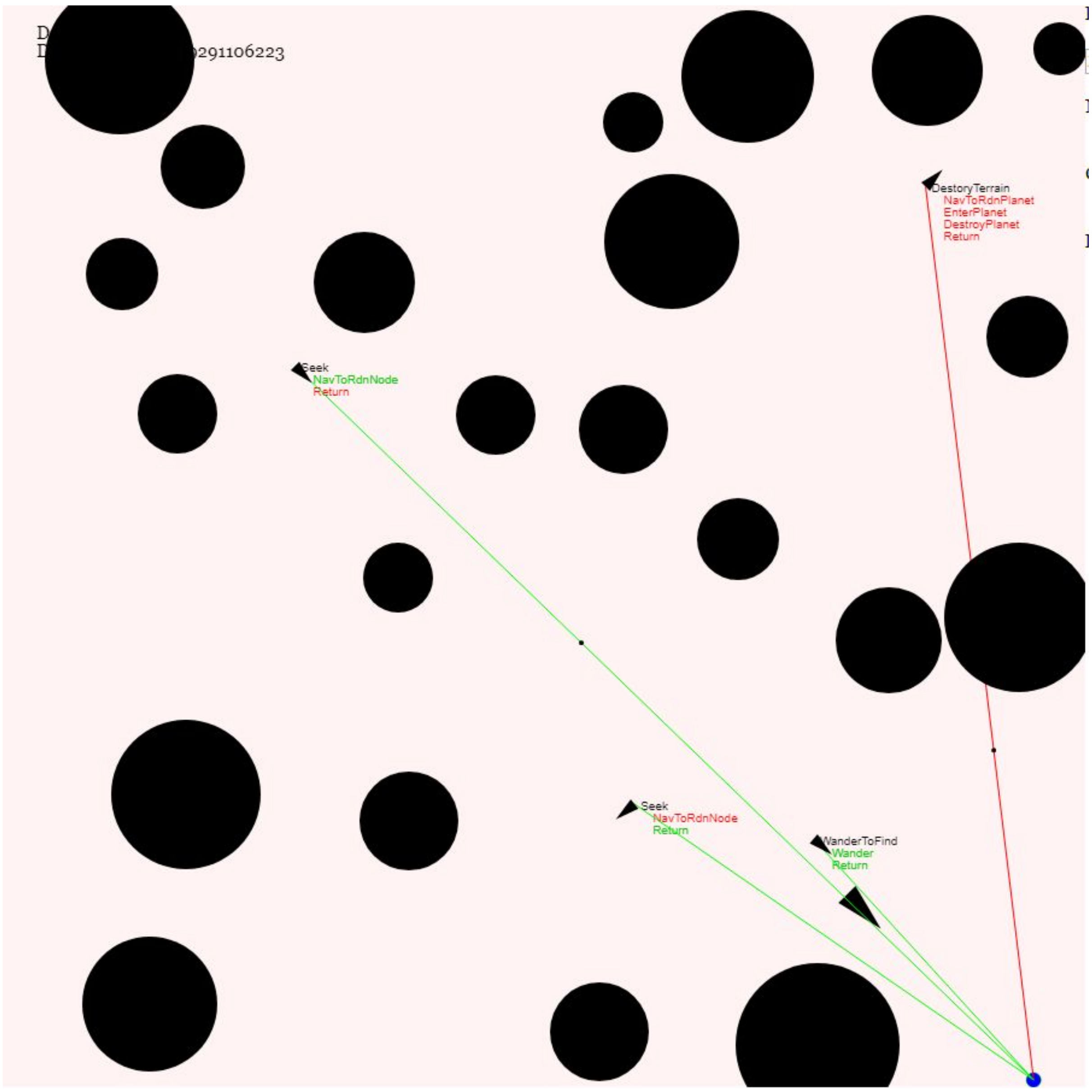
Grote Zwarte Driehoek

By Bart van Mierlo & Nicander Mohrmann



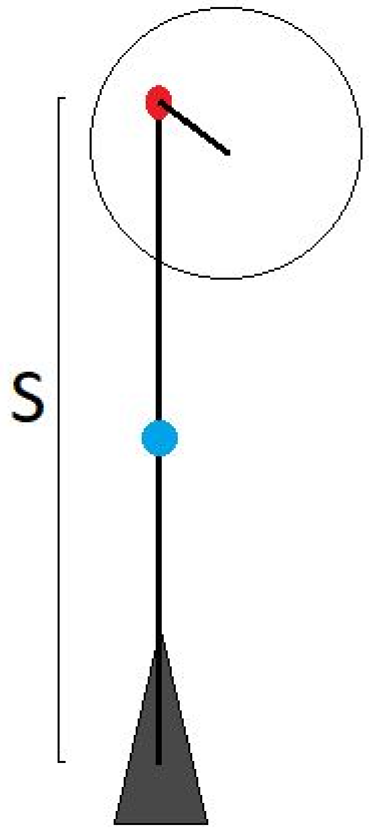
Github: https://github.com/Ricebal/GroteZwarteDriehoek
Date: 01-04-2019

Behaviours

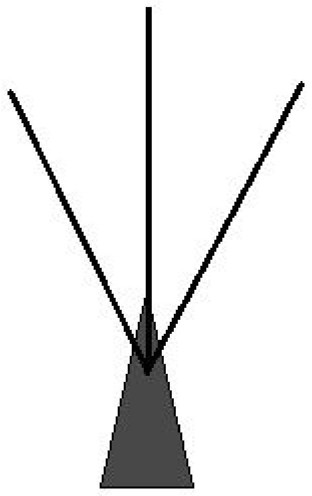
In this project we had to implement a few behaviour for the agents to follow so they can move about in the game world. We have made the following behaviours:

Obstacle avoidance behaviour

Obstacle Avoidance Behaviour
(Even though this explanation uses triangles, we are not shapist and circles can also have obstacle avoidance, any movingGameEntity can)
The obstacle avoidance Behaviour makes use of two sensory nodes in front of the triangle. Two nodes because it results in smoother behaviour. With one node it wants to either make shortcuts through objects if it's too far, or reacts too late if it's too close. The nodes are not always in the same place, it's dependant on the velocity of the triangle. The red node is twice as far as the blue node. To detect objects the circle checks with the planet if the distance from the node to the position of the planet is smaller than the radius (size) of the planet. If this is the case the node must be in the planet. It will react the closest planet first.



One of the problems we encountered was jittering of the triangle. This is because the triangle wants to steer away from the circle, but right after it is steered back into it. This effect is stronger because the amount of steering we allow our triangle is dependent on its velocity. Which in space makes sense and creates quite natural movements in the simulation. The exception is when it steers away and loses velocity it will jitter because it's low velocity will allow for basically free steering.
This could possibly be prevented using a system of two or three sensory nodes, or implementing a minimum speed, or capping the amount of steering that is allowed, even at low speeds.



Seek behaviour

The seek behaviour is used to move from a position towards a defined other position, be it stationary or moving. This behaviour won’t stop moving once it hits the target location. The behaviour first gets the desired vector by subtracting the target vector from the agent's position. This vector is then normalized and multiplied by the max speed of the agent. After that we make a steer vector that is the desired vector subtracted by the agents velocity, which we then limit to the agents maximum force.

Arrive behaviour

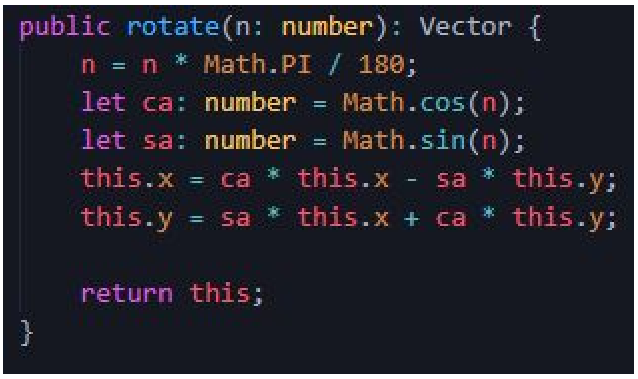
The arrive behaviour works much the same as the seek behaviour as in it moves the agent from a position to another defined position, but this time it stops when it hits the target location. This is done by having the speed of the agent relative to the distance between the starting position and the target location. We get the same desired vector by subtracting the agents position from the target vector and normalizing it. Now instead of just multiplying by the max speed of the agent, we multiply by the squared distance of between the agent and the target and dividing it by the squared distance between the starting position and the target position, and multiplying this by the agents max speed. After that we proceed with the same calculation to apply the agents maximum force.

Flee behaviour

The flee behaviour has a variable that stores how close an agent has to be before it wants to flee, the panic distance. We check if the distance squared between the agent and what it’s fleeing from is smaller than the panic distance squared and if it is we do the same thing as the seek behaviour, but in reverse, so it runs away instead of towards something.

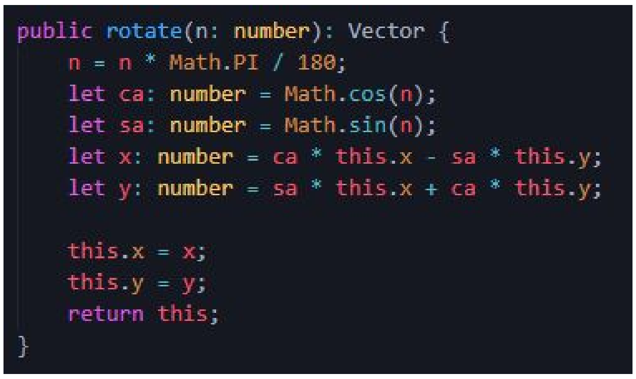
Follow behaviour

Our implementation of the follow behaviour is a bit different than normal, instead of just following a target, what the follow behaviour does is it works in a group. If multiple agents are following the same target they will form a formation behind the target, as if they were flying in a fleet. This is done by applying the following formula: 180 / [amount of agents in the group] * [index of agent in the group] + 90 + (180 / [amount of agents in the group] / 2). To find the location for a specific agent all we have to do is take the velocity of the target they're following, rotate it by the solution of the formula and then normalizing it and multiplying it by the size of the agent. It then uses the same ways of seek and arrive to get there.
The only problem we ran into developing this behaviour was that the rotate function was behaving oddly because we were modifying the x in a vector, and then using that modified x in the calculation to modify y, so it would spit out wrong numbers. We fixed this by changing the code from how it was on the left to how it is on the right.



```
public rotate(n: number): Vector {
    n = n * Math.PI / 180;
    let ca: number = Math.cos(n);
    let sa: number = Math.sin(n);
    this.x = ca * this.x - sa * this.y;
    this.y = sa * this.x + ca * this.y;

    return this;
}
```

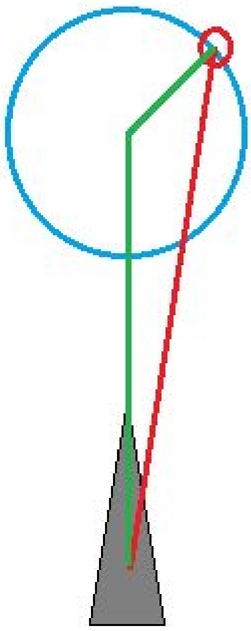


```
public rotate(n: number): Vector {
    n = n * Math.PI / 180;
    let ca: number = Math.cos(n);
    let sa: number = Math.sin(n);
    let x: number = ca * this.x - sa * this.y;
    let y: number = sa * this.x + ca * this.y;

    this.x = x;
    this.y = y;
    return this;
}
```

Wander behaviour

To implement the wander behaviour we used the method that calculates a circle in front of the agent and generates a random point on that circle for the agent to move to, this behaviour was pretty easy to implement and can also be easily adjusted to change the size of the circle and the distance between the agent and the circle.



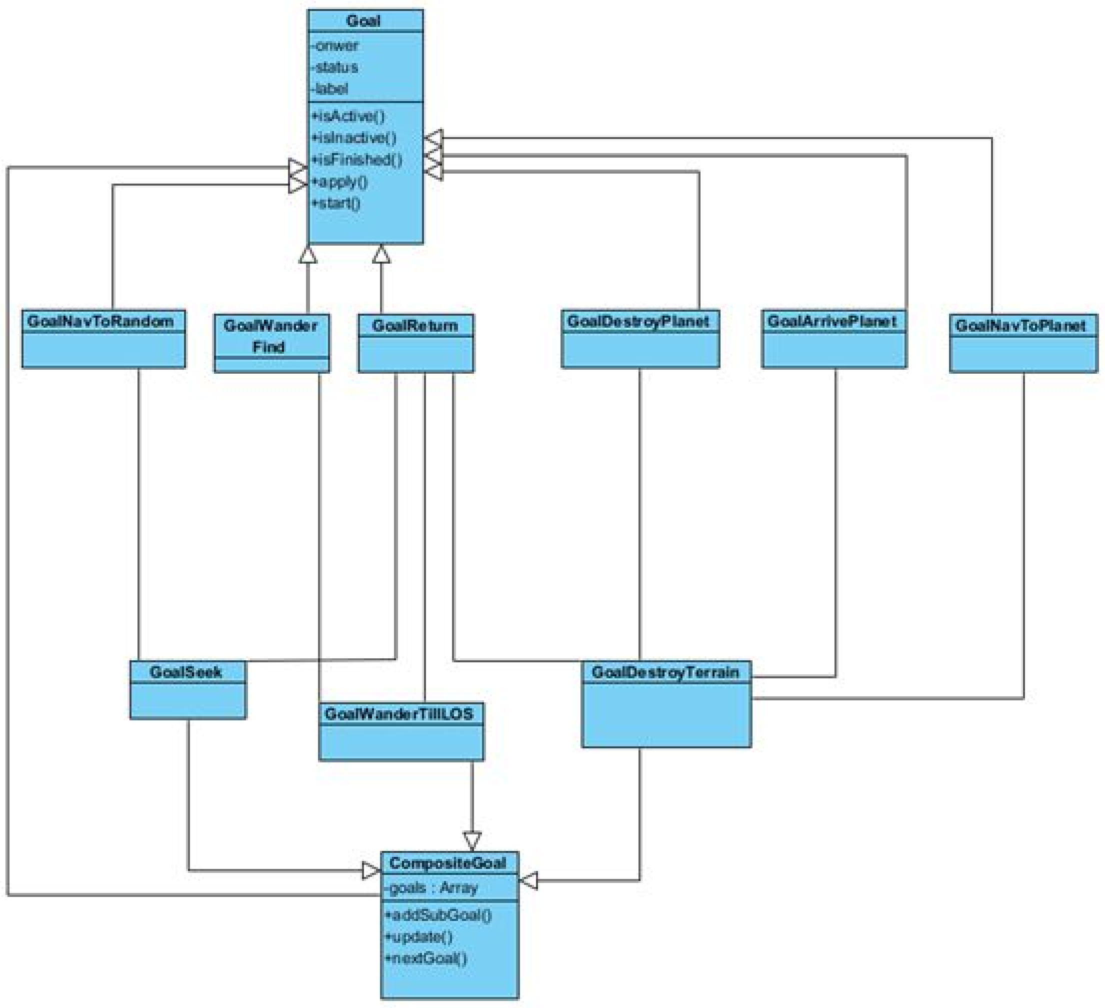
Graph and Pathfinding

We implemented pathfinding in our simulation using the a* algorithm. Our implementation of this algorithm used a graph with horizontal, vertical and diagonal edges. First we filled the world with a flood fill algorithm, after which we connected these nodes with each other so that a graph was created. We opted to use a recursive function to do the flood fill with three stop conditions: There is already a node in the current position, there is a planet in the current position and the current position is not inside of the canvas area.
The only problem we faced is that if a planet is relatively small an edge could be generated that went through the planet. We decided that this was not worth fixing and just opted to not make small planets.

The pathfinding was done by first taking the starting position of the agent and finding the closest node on the graph to that position and moving the agent to that node. After that we used a* by making two arrays, open and closed. Open is an array that holds all the nodes that have been seen but not checked yet and close is an array that holds all the nodes that have been seen and been checked. On a check we check if the score a node has is higher or lower than the new score + heuristic. If the existing score is lower nothing happens, and if the existing score is higher we update the score to the new score and save the current node as the checked nodes previous node. By doing this we can easily get the path at the end by just recursively checking the nodes previous node until it is null. At the end all the graph nodes get converted to vectors so that we can apply seek behaviours to them so the agent can follow the path, by using our existing seek behaviour the agent follows the path smoothly instead of making impossibly sharp turns.

Goals

Our game has 3 goals, which in our case means there is 3 composite goals, which have their own parts, or atomic goals. All three make use of the atomic goal GoalReturn, which returns the owner to following the Big Black Triangle. **GoalSeek** makes use of the pathfinding behaviour we made to go to a random node in the graph and then return to Big Black Triangle.
GoalWanderTillLOS makes use of the wander behaviour previously made, wanders around until the small blue circle is within line of sight then returns to the Big Black Triangle.
GoalDestroyTerrain uses the node pathfinding behaviour to get close to a randomly selected Planet, uses the arrive behaviour to dig into the planet to destroy it, then returns to the Big Black Triangle.
Possible future implementations can solve the gaps this causes in the grid for pathfinding.
The behaviours have conditions which need to be met to finish, then the above composite goal will know to move on to the next composite or atomic goal in the list.
In this example the Goals are randomly selected for each Small Black Triangle, once they are done they will return to leader following behaviour which has no end.
It is very easy to implement new goal trees in this system to suit the needs of the simulation.
This class diagram clarifies which goals are composite and which goals are atomic, and which composite goals use which atomic goals. Note compositeGoal extends goal.


Goal
-onwer
-status
-label
+isActive()
+isInactive()
+isFinished()
+apply()
+start()
GoalNavToRandom
GoalWander Find
GoalReturn
GoalDestroyPlanet
GoalArrivePlanet
GoalNavToPlanet
GoalSeek
GoalWanderTillLOS
GoalDestroyTerrain
CompositeGoal
-goals : Array
+addSubGoal()
+update()
+nextGoal()

Line of Sight detection

Even though it was not in the assignment we wanted to include Line of Sight detection into the project because it seemed to make sense that the triangles wanted to see the blue circle at some point, and the math behind it is interesting. For this we applied the maths detailed here; http://paulbourke.net/geometry/pointlineplane/.

Point d is calculated this way. Then it’s a matter of checking if the distance between point d and c is bigger than the radius of c, if so there is no intersection.