Steering behaviors

In C# and C++

Applied in Boxhead

Introduction

In this paper, I will go over the process of research I have done to get to the final result of the artificial intelligence in my game Boxhead.

The game has A* pathfinding implemented but since this paper is all about steering behaviors, I will not go over the concept or implementation of the A* pathfinding algorithm, although this will be used in combination with the steering behaviors.

A* pathfinding is a great pathfinding algorithm but by itself, just having an agent follow the path precisely feels extremely unnatural and limited. My goal is to improve the AI by combining this with steering behaviors and get a more natural and believable result.

Before implementing steering into the game, I decided to experiment in Unity3D first so that the debugging and iterating would go faster. The goal is to research all the separate behaviors first, going from the really simple ones to the more complicated ones and eventually think about a way to combine and apply them in my game.

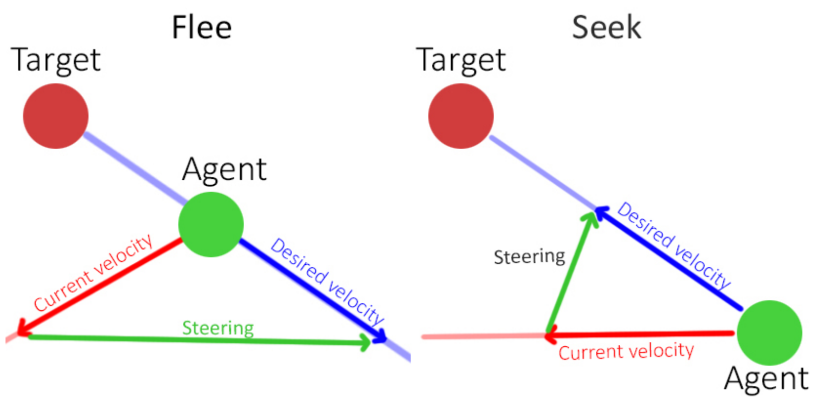
The basics

Since steering behaviors were pretty new to me, it was a good idea to go every separate behavior individually in Unity. When I became familiar with the behaviors and implemented them in Unity, it would be easier to then convert them to C++.

For the agent I use a CharacterController since my game in C++ uses them as well. This may sound wrong but increasing the controller's velocity towards the desired velocity over time results in the same effect.

1. Seek and flee

The very simplest behavior is "Seek" and "Flee". Seek moves the agent towards the target position by calculating the steering acceleration to direct the agent towards the target.



```
Vector3 desiredVelocity = position - position;
float distance = desiredVelocity.magnitude;
desiredVelocity = Normalize(desiredVelocity) * MaxVelocity;
Vector3 steering = desiredVelocity - currentVelocity;
return steering;
```

Flee is the opposite behavior of seeking. It calculates a steering force to move the agent away from the target. This is achievable by simply multiplying the seek with -1.

2. Look where you are going

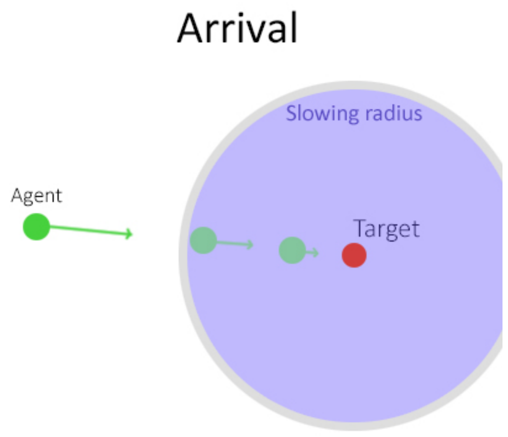
This is not exactly a steering behavior but this behavior rotates the agent towards the direction it is moving in.

```
    lookRotation = LookRotation(agentVelocity)
```

Because steering uses velocity in its calculations, the resulting velocity that gets set to the agent will be gradually changed over time. This way not only the movement will look really natural but also the way the agent looks towards its target.

3. Arrival

Seek behavior, at the moment, has a problem when it reaches its destination. At the destination, the agent will bounce off the target since it actually did not arrive there yet because the target itself is in the way or the agent just moves over it by a little. This together with what is said last topic, the velocity will gradually change over time. But when it reaches its destination, the agent will just move a little too far and then come back again.



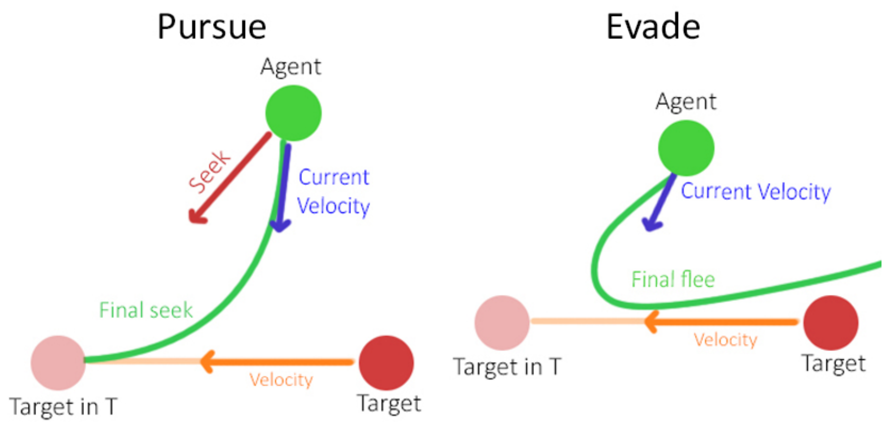
Arrival prevents the agent from moving through the target. For this, the agent has a 'slowing radius'. When the agent is inside this radius, the seeking force is decreased until the agent reaches its destination and the seeking force is zero.

```
desiredVelocity = targetPosition - position;
float distance = desiredVelocity.magnitude;

if (distance < SlowingDistance)
    desiredVelocity = Normalize(desiredVelocity) * MaxVelocity * (distance / SlowingDistance);
else
    desiredVelocity = Normalize(desiredVelocity) * MaxVelocity;
Vector3 steering = desiredVelocity - currentVelocity;
return steering;
```

4. Pursue and evade

Pursue and seek are very alike except that pursue predicts where the target will be in time T so it intercepts the target. For the implementation, the agent must know the velocity of the target to predict where the target will be. To improve the result, the pursue acceleration decreases depending on how close the target is.

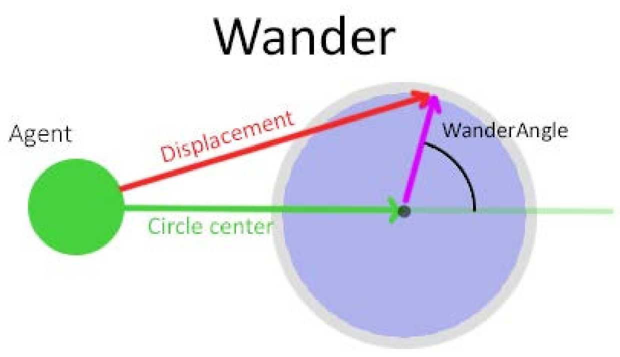


```
float distance = Distance(Target.position, position);
float ahead = distance / 10;
futurePosition = Target.position + Target.velocity * ahead;
return Seek(futurePosition);
```

Evade is like flee the opposite of pursue. It considers the velocity of the target and flees from the position where the target will be in time T.

5. Wander

Wandering is often used in games when the agent does not have a direct task to do and waits for something to happen. A simple implementation is just to have a set interval in where it calculates a random position and seeks to that position. Unfortunately, this results in an unrealistic behavior since every interval, the target position suddenly changes.



A second implementation makes use of a circle that is from a user-set distance from the agent's position. A random direction on the circle is taken and that vector is added to the circle position vector. This result is normalized and multiplied with the avoidance force. Every frame, the wander angle is incremented by a small amount.

6. Flow field pathfinding

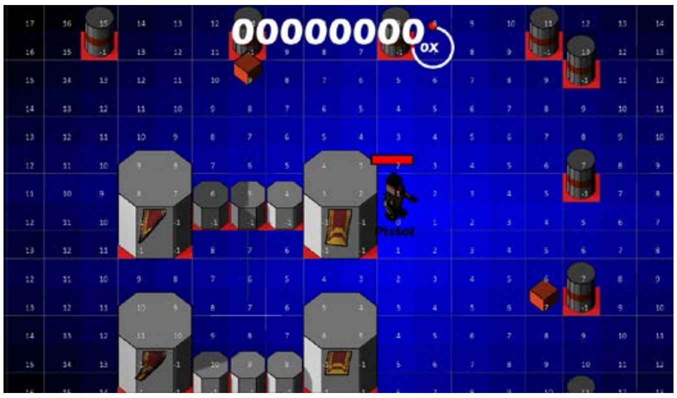
The first time I've heard about flow field pathfinding is during a video of Supreme Commander. The algorithm looked really interesting to research and compare with the most commonly used algorithm A*.

The flow field algorithm uses a graph of nodes that holds the distance from the target node. Before a flow field can be made, a heat map has to be calculated and after that the flow field can be calculated using the node distances from the heat map.

Heat map

The heat map gives every node that distance from the target node. This is done using a breadth first search. This algorithm consists out of three main steps:

1. Start at the goal, set the distance to 0.
2. Get the neighbors of each goal and set their distance to the distance of the previous node plus one and add them to a queue.
3. Dequeue the next node from the queue and redo step 2 until the queue is empty.



```
void CreateHeatmap(FlowFieldNode node, Queue<FlowFieldNode> queue)
{
        var neighbours = GetNeighbours(node);
        foreach (FlowFieldNode n in neighbours)
        {
                n.Distance = node.Distance + 1;
                queue.Enqueue(n);
        }
        if (queue.Count == 0)
                return;
        FlowFieldNode next = queue.Dequeue();
        CreateHeatmap(next, queue);
}
```

Flow field

When the distances for each node are populated, creating the vector field is surprisingly simple. Of course, getting the neighbors has some exceptions.

```
vector.x = left - right;
vector.z = down - up;
vector.normalize();
_grid[i].Vector = vector;
```

The vector field simply stores a vector that points down the gradient of the heat map.

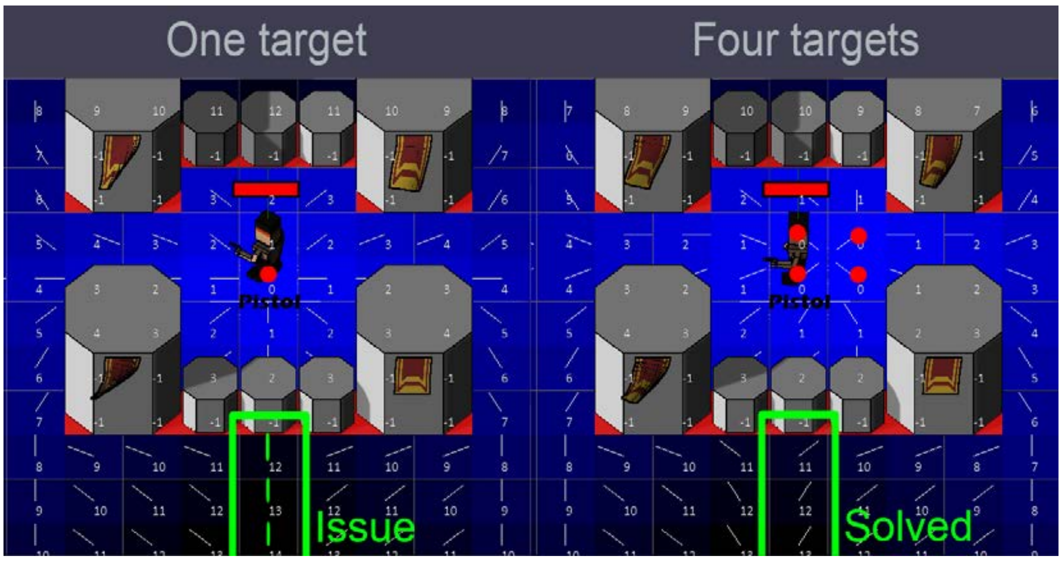
After doing some more research, I found another way to do this that eventually worked better. The first approach has a major downside which is called "local optima", which I will discuss later. With the second approach, you iterate over the whole grid and for each cell, you get the neighbor (can be diagonal), with the lowest distance value. The vector of the current node is the direction to that neighboring node. Even if two neighbors have the same distance value, the node will always point towards one of the two. This approach is a little less efficient but does not have the local optima problem. This is why I chose this approach for my game.

```
int minDistance = numeric_limits<int>::max();
int minDistanceIdx = 0;
for (size_t j = 1; j < neighbors.size(); j++)
{
	if (neighbors[j]->IsWalkable() == false)
		continue;
	if(neighbors[j]->GetDistance() < minDistance)
	{
		minDistance = neighbors[j]->GetDistance();
		minDistanceIdx = j;
	}
}
vec.x = neighbors[minDistanceIdx]->GetPosition().x - m_Grid[i]->GetPosition().x;
vec.z = neighbors[minDistanceIdx]->GetPosition().z - m_Grid[i]->GetPosition().z;
vec.normalize();
```

The result

Now that the flow field is generated, agents can use this flow field to make their way to the target. Using a formula to get the closest node to their world position, the agent can get the node's vector and use it to calculate its steering force.

After a while of testing the method, I noticed a big issue with the flow field. When there are two possible paths from a node that have an equal distance, the weight between these nodes get balanced. This results in an orthogonal vector (x- or y-component is zero). This does not seem like a problem because it is probably just a straight line towards the target. But if there is an obstacle in between the node and the target, the vector of those nodes will have the same direction as the normal of the obstacle. This leads to a huge problem because agents on this particular node will just bump in to the wall. After doing some research on this issue, I found out this was a common problem called "local optima". One of the solutions I've seen the most is to subdivide each node into four nodes and mark the four goal nodes with distance zero. I did not like this idea since it would quadruple the memory usage. The solution I used uses the same principle but does not subdivide the nodes. Having four targets makes sure that neighbors of a node will never outbalance each other so the problem of "local optima" is solved.



In the image, you see the problem solved using four target nodes.

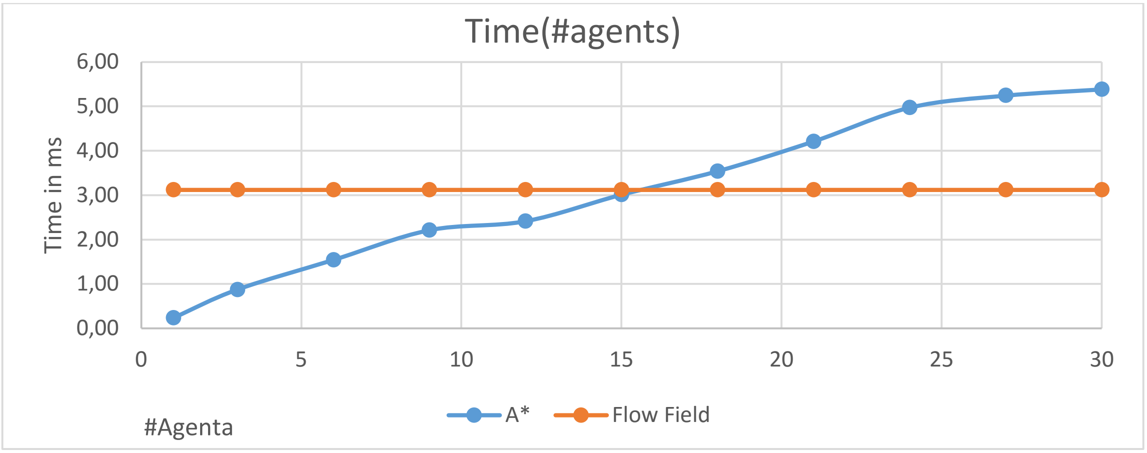
The main properties of using flow field pathfinding is that having multiple agents with the same target is extremely efficient but having a few agents with multiple targets requires the algorithm to calculate a heat map and a flow field for every target which is quite inefficient.

Flow field pathfinding vs. A* pathfinding

Comparing A* star pathfinding with flow field pathfinding shows that flow field pathfinding path finding is extremely efficient when there is only one target and multiple agents since it only needs to recalculate the vector field each frame instead of finding a path for each agent with A* pathfinding.

Each has their advantages and disadvantages. The reason to use A* above flow fields is precision. If there would be areas that are not walkable but are not physical obstacles, the agent would be able to run through it (see image for clarification). This is a big issue with flow field pathfinding that A* does not have at all.

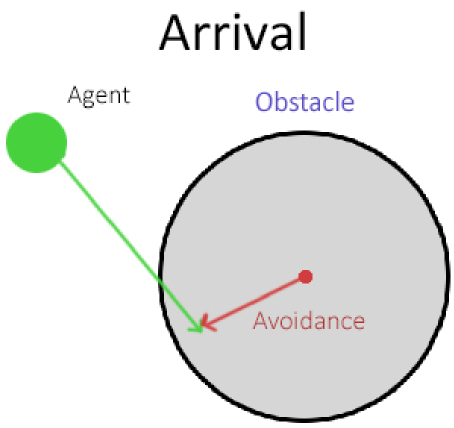
On the other side, if you have hundreds of agents and one target, flow field pathfinding is the most efficient choice. Flow field pathfinding works with velocity and thus works extremely well together with other steering behaviors.



In my situation, combining an A* grid and a flow field together is very convenient since a flow field node just requires adding a distance and a vector variable. Other than this, the node structure of both algorithms are very much alike. After this, the calculation of a flow field can be optional.

7. Obstacle avoidance

Obstacle avoidance is a useful way for agents to locally prevent it from colliding in to obstacles or other agents. This is no solution to pathfinding by its own but it helps making the movement more realistic in a local area.



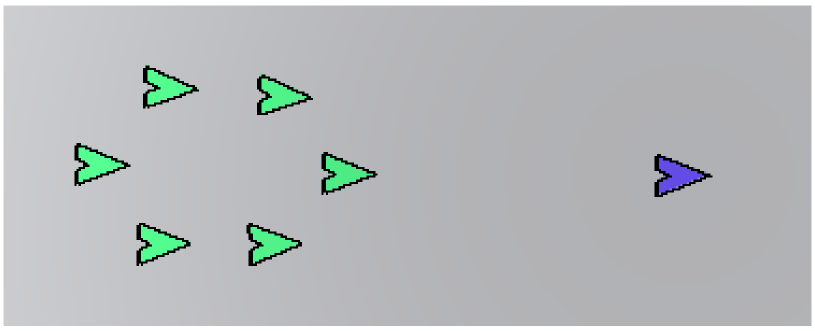
I have read many ways to handle obstacle avoidance and the most common one is doing a raycast towards the direction where the agent is going with the length of the agent's 'vision'. When an object is hit, the steering vector is calculated by subtracting the endpoint of the ray by the center point of the obstacle like shown in the image.

This seemed to be working quite alright but I thought, having Nvidia physX at my disposal on both platforms, it would maybe be better to use the normal of the surface that has been hit and reflect the forward direction of the agent with that normal. This idea unfortunately did not work out that well.

I can conclude that obstacle avoidance is definitely no replacement for actual pathfinding. It is a great way for local avoidance but when it comes to finding a way to a target is too much without using a pathfinding algorithm.

8. Flocking

Flocking is one of the most interesting steering behavior besides flow field following. It is a balance between three different forces: cohesion, separation and alignment. I will not go in to flocking in this paper but I will eventually use it in my application. The basic idea is that, when agents move in a group, cohesion will keep them together, separation will make them keep their distance from each other and alignment makes sure that they are all moving in the same direction. The result is a behavior like a flock of birds (That is where the name comes from). Another term for agent that flock is boids (bird-oids).



Combining steering behaviors

The most important advantage of steering behaviors is that they can be easily combined by just adding all the elements together. After adding them all, you truncate the result so it does not exceed the agent's max velocity and you got your steering force. Another very advantageous thing is that every steering force can have its own weight in the resulting force. By just multiplying a certain behavior with a weight, that specific behavior can influence the resulting force more or less to achieve many different results. Because of this, changing the weights dynamically, depending on different events, is a really interesting thing to play with.

The integration goes as follows (pseudo-code):

```
force = SteeringForceSum;
acceleration = force / mass; //Optional
velocity = acceleration * deltatime;
speed = velocity.length;
agent.move(velocity);
if(speed > lookAtThreshold)
       lookDirection = velocity;
```

Implementation in Boxhead (C++)

I won't go too far in depth on how I implemented this in C++ since the algorithms obviously stay the same no matter what programming language that is used. Only some datatypes differ. Here a few examples:

- Vector3 → XMFLOAT3 / XMVECTOR / PxVec3
- Queue<T> → std::queue<T> (Enqueue / Dequeue → push() / front() & pop())
- List<T> → std::vector<T>

I have two main classes to maintain the pathfinding: Grid and Pathfinding. The grid is created at the start of the program and holds all the node data. The pathfinding class uses the singleton pattern so it can be used everywhere. This class mostly handles all my A* pathfinding but also serves as an accessor for the grid. This way enemies can both access the Pathfinding class for A* pathfinding as well as access the grid to get the node data for flow field pathfinding.

I will divide the implementation in two parts, the player and the enemies.

The player

The player uses a RTS-like point and click system. Left click to shoot in the mouse direction and right click to move to the mouse position. For this I used A* in combination with some steering behaviors.

The first thought was just to, when the player clicks on a location, calculate a path using A* and just make the player follow the path. The result was alright but felt really unnatural especially when the player saw the target. The first improvement was to use path follow to move along the path in a more natural way.

```
PxVec3 currPos = GetTransform()->GetWorldPositionPcVec3();
PxVec3 targetPos = ToPxVec3(m_Path[m_CurrentPathIndex]->GetPosition());
if(DistanceXZ(currPos, targetPos) <= m_NodeRadius)
	++m_CurrentPathIndex;
PxVec3 direction = targetPos - currPos;
direction.normalize();
return direction;
```

The code is simplified because normally it would require you to handle some exceptional situations.

As you can see, like every steering behavior, it returns a vector. This way it's really convenient to move the player over time, giving it a natural ease-in and –out feel.

A second thing I realized is that when the player sees the target, meaning that there are no more obstacles in the way, it is not necessary anymore for the player to follow the path. Instead the player can just do a simple **seek** together with an **arrival** behavior towards the target resulting in a more desirable movement pattern.

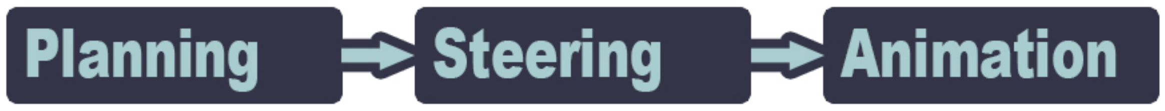
The enemies

The enemies are zombies and most of the time move in groups. So I thought it was the perfect opportunity to make use of the flow field and some other steering behaviors. I noticed that it was quite the challenge to make use of as many steering behaviors as possible since steering is better used in RTS games or crowd simulations. Nevertheless, experimenting with it eventually gave some good results.

The basic idea is using flow pathfinding to have the zombies make their way to the player. When the zombies are in line of sight of the player, the would do a pursue towards the player while combining separation and cohesion in a small range to have the surround you when they reach you instead of them standing in one line. Since I use a behavior tree, it is really convenient since it is easier to focus on each behavior separately and them bringing them together.

Conclusion

The reason why steering behaviors are so good is that they are not based on complex strategies involving path planning or global calculation but still create a very believable result when combined. The implementation is easy to understand and combining the behaviors can produce complex movement patterns. Combining the behaviors is extremely intuitive since they all rely on a desired velocity. This means you can just add all the desired behaviors together, each having a certain 'weight' in the resulting calculation.



The main idea behind the whole artificial intelligence is first **selecting the action** by strategy, planning, decisions, … and **calculating steering** forces accordingly. After the path is determined by the steering, the locomotion (animations) can be applied and modified according to the applied forces/velocity.

Steering behaviors are meant to be used when having multiple agents that move in groups like in real time strategy games like Supreme Commander and Planetary Annihilation.

Using steering behaviors in my game was a challenge because at first sight, A* pathfinding combined with some own behaviors in a behavior tree seemed like the best solution. The pathfinding is more precise and robust but on the bad side, the performance is lower. For demo purposes I did not combine A* with any steering behaviors but created enemies that use A* and enemies that use several steering behaviors.

References and sources:

Introduction to steering behaviors:

http://www.gamasutra.com/blogs/JuanBelonPerez/20140724/221421/Introduction_to_Steering_Behaviours.php

Paper of Craig Reynolds on "Steering behaviors for autonomous characters"

http://www.cs.uu.nl/docs/vakken/mcrs/papers/8.pdf

Steering Behaviors overview

http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732

Flocking basics:

http://gamedevelopment.tutsplus.com/tutorials/the-three-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444

Overview of different steering behaviors

https://github.com/libgdx/gdx-ai/wiki/Steering-Behaviors#independent-facing

Pentheny Graham – The Next Vector presentation at GDC 2013

http://gdcvault.com/play/1018230/The-Next-Vector-Improvements-in

Battle circle AI

http://gamedevelopment.tutsplus.com/tutorials/battle-circle-ai-let-your-player-feel-like-theyre-fighting-lots-of-enemies--gamedev-13535

Intro to flow field pathfinding

http://gamedevelopment.tutsplus.com/tutorials/understanding-goal-based-vector-field-pathfinding--gamedev-9007

The breadth first search algorithm

http://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm

Flow field generation

https://howtorts.github.io/2014/01/04/basic-flow-fields.html

All images are created by myself.

Paper written for Digital Arts & Entertainment, Howest University.

Gameplay Programming, 2016