Report - Final Assignment v1.0
Game AI.

HBO-ICT, Games Programming
Windesheim University of Applied Sciences
Lecturer: Gido Hakvoort

Windesheim, Zwolle
Version: 1.0
May 29, 2018
Authors : Gijs Alberts, Anne Zweers

Revision History

Revision	Date	Author(s)	Description
1.0	12-04-2018	Gijs Alberts Anne Zweers	Initial version of the document.

Contents

1 Introduction

1.1 Tussenkopje

citatie van iets [1] zie voor bibid bibliography in FinalAssignment

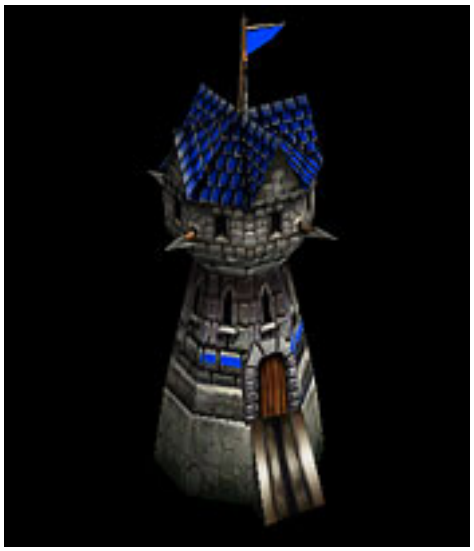


Figure 1: Example of the summary of a method.

1.1.1 tussentussenkopje

paragraaf

subparagraaf

2 Steering

In this chapter we will describe:

- The steering behaviors we implemented.
- How we implemented these steering behaviors.
- The issues and problems we faced while implementing the steering behaviors and how we solved these issues.
- How steering behaviors can be combined within our game.

A class diagram about all of the classes that are relevant to the steering behaviors can be found at the bottom of this chapter.

2.1 Behaviors

We implemented the following steering behaviors:

- Seek.
  Will steer the agent towards a target within the game.
- Flee.
  Will steer the agent away from a target within the game.
- Arrive.
  Will steer the agent towards a target within the game and will decelerate as the agent gets closer to the target position.
- Follow Path.
  Will steer the agent along a path. Uses seek to reach all the waypoints but the last waypoint within the path. Arrive is used for the last waypoint.
- Pursuit.
  Will steer the agent towards a target agent.
- Evade.
  Will steer the agent away from a target agent.
- Offset Pursuit.
  Will steer the agent towards a target agent while keeping a distance between them.
- Explore.
  Will make the agent explore a queue of Powerups.

Our implementation supports the following summing methods:

- Weighted truncated sum.
- Prioritization.
- Prioritized dithering.

2.2 Issues and Solutions

We mainly just rewrote the source from the Programming Game AI by Example book by Mat Buckland into C# so we didn't have too many problems. We had some issues with updating the Velocity/Heading of the agent's and calculating the steering force based on the elapsed time. The measurement of elapsed time that Mat Buckland uses didn't seem to be documented within the source so it took some adjustments to get it about right.

We used a different implementation for the Offset Pursuit than the book did. The reason for this is that this seemed like a simpler implementation with the same effect.

```
//------------------------ Offset Pursuit -----------------------------
//
//  Produces a steering force that keeps a vehicle at a specified offset
//  from a leader vehicle
//--------------------------------------------------------------------------
Vector2D SteeringBehavior::OffsetPursuit(const Vehicle*  leader,
                                         const Vector2D offset)
{
  //calculate the offset's position in world space
  Vector2D WorldOffsetPos = PointToWorldSpace(offset,
                                              leader->Heading(),
                                              leader->Side(),
                                              leader->Pos());

  Vector2D ToOffset = WorldOffsetPos - m_pVehicle->Pos();

  //the lookahead time is propotional to the distance between the leader
  //and the pursuer; and is inversely proportional to the sum of both
  //agent's velocities
  double LookAheadTime = ToOffset.Length() /
                         (m_pVehicle->MaxSpeed() + leader->Speed());

  //now Arrive at the predicted future position of the offset
  return Arrive(WorldOffsetPos + leader->Velocity() * LookAheadTime, fast);
}
```

Figure 2: Original Offset Pursuit.

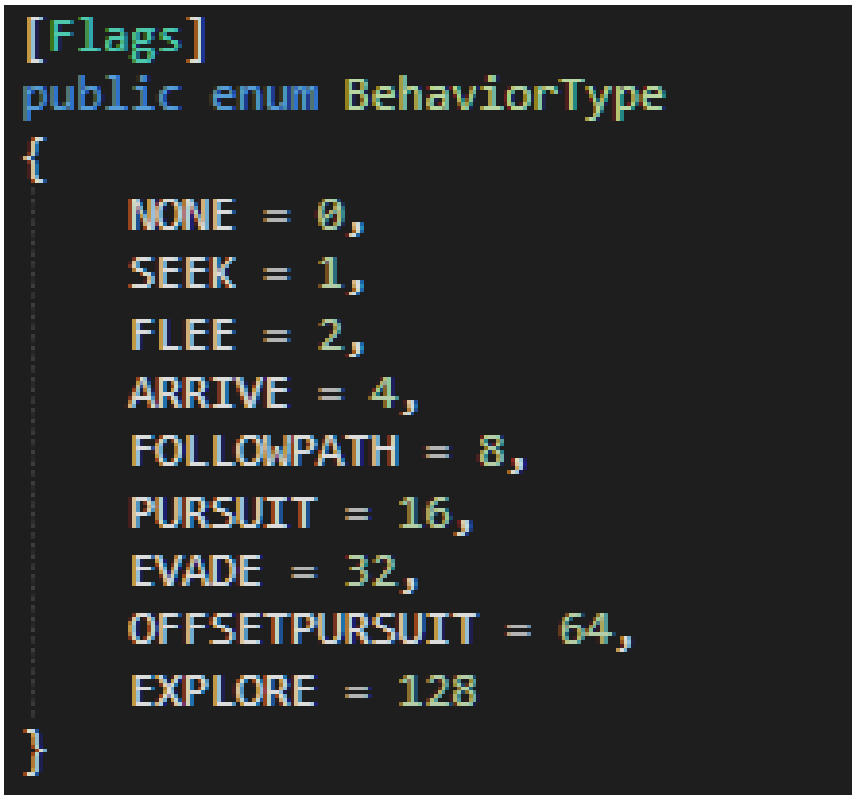
```
/// <summary>
/// Produces a steering force that keeps a FlyingEntity at a specified offset
/// from a leader FlyingEntity.
/// </summary>
/// <param name="leader"></param>
/// <param name="offset"></param>
/// <returns></returns>
public Vector2D OffsetPursuit(FlyingEntity leader, int offset)
{
    return Arrive(leader.Pos - leader.Heading * offset, DecelerationRate.FAST);
}
```

Figure 3: Implementation of Offset Pursuit.

We also had some issues with oscillation which were solved by creating a threshold. We basically made it so that if the agent is within a certain very close distance to the target position the agent's position will be transformed into the target position after finding an article about the issue. [2].

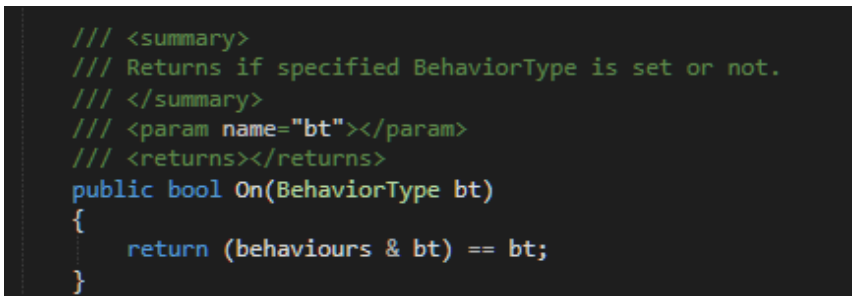
2.3 Combining Steering Behaviors

Combining multiple steering behaviors is very easy since we use a bit operator system for it (just like the book did).



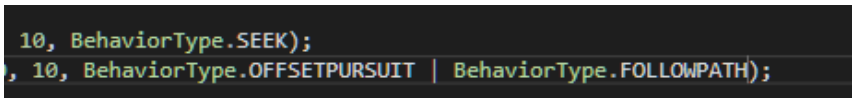
```
[Flags]
public enum BehaviorType
{
    NONE = 0,
    SEEK = 1,
    FLEE = 2,
    ARRIVE = 4,
    FOLLOWPATH = 8,
    PURSUIT = 16,
    EVADE = 32,
    OFFSETPURSUIT = 64,
    EXPLORE = 128
}
```

Figure 4: Behaviour flags.



```
/// <summary>
/// Returns if specified BehaviorType is set or not.
/// </summary>
/// <param name="bt"></param>
/// <returns></returns>
public bool On(BehaviorType bt)
{
    return (behaviours & bt) == bt;
}
```

Figure 5: Method for asserting if a behavior is enabled or not.



```
10, BehaviorType.SEEK);
, 10, BehaviorType.OFFSETPURSUIT | BehaviorType.FOLLOWPATH);
```

Figure 6: Combining steering behaviors.

2.4 Class Diagram

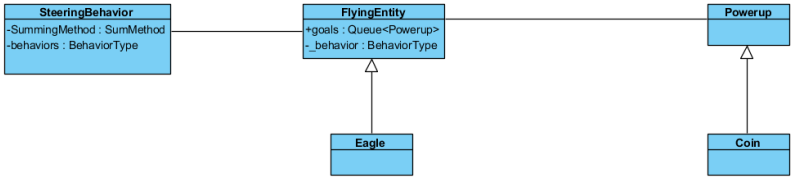


Figure 7: Class diagram of classes that are relevant to steering.

3 Path Planning

In this chapter we will describe:

- How the graph representing the environment is generated.
- The heuristic we use in A*.

A class diagram about all of the classes that are relevant to the path planning can be found at the bottom of this chapter.

3.0.1 Generation of the graph

Our game consists of a tile system. The Game world is divided into a 40x40 grid of tiles and each buildable tile has it's own vertex. The vertex of a tile is connected to the vertexes of neighbouring tiles by Edges, creating a graph.

```
/// Initialzes Graph of Gameworld.
public void InitializeGraph()
{
    // Destroys all Vertices.
    for (int i = 0; i < GameWorld.Instance.tiles; i++)
    {
        GameWorld.Instance.tilesList[i].DestroyVertex();
    }

    /*
     * Creates Vertex on every buildable Tile.
     * Connects this Vertex with all neighbouring Vertices (corresponding to other buildable Tiles)
     * by adding Edges between them.
     */
    for (int i = 0; i < GameWorld.Instance.tiles; i++)
    {
        BaseTile tile = GameWorld.Instance.tilesList[i];
        if (tile.buildable)
        {
            CreateGraph(tile);
        }
    }
}
```

Figure 8: How the graph of the Gameworld is generated.

```
/// Creates vertex that corresponds to Tile if Tile is not null.
/// Gets all availabile neighbouring tiles of this Tile.
/// Connects the Vertex of the current Tile with the vertex of each available (for building) neighbouring Tile
/// if they aren't already connected. This is done by adding an Edge between them.
public void CreateGraph(BaseTile tile)
{
    if (tile == null) return;
    tile.CreateVertex();

    List<BaseTile> neighbours = GameWorld.Instance.GetAvailableNeighbours(tile);
    foreach (BaseTile neighbour in neighbours)
    {
        if (tile.IsConnected(neighbour)) continue;
        neighbour.CreateVertex();
        AddEdge(tile, neighbour, cost);
    }
}
```

Figure 9: CreateGraph method.

3.0.2 Heuristic

We use the Manhattan heuristic for calculating the distance between two tiles.

```
/// Returns a estimate of the distance in Tiles between 2 points.
private static int Heuristics(Vector2D a, Vector2D b)
    => (int)Math.Abs((a.x / BaseTile.size) - (b.x / BaseTile.size)) + (int)Math.Abs((a.y / BaseTile.size) - (b.y / BaseTile.size));
```

Figure 10: Function for calculating Manhattan distance between two points.

3.0.3 Class Diagram

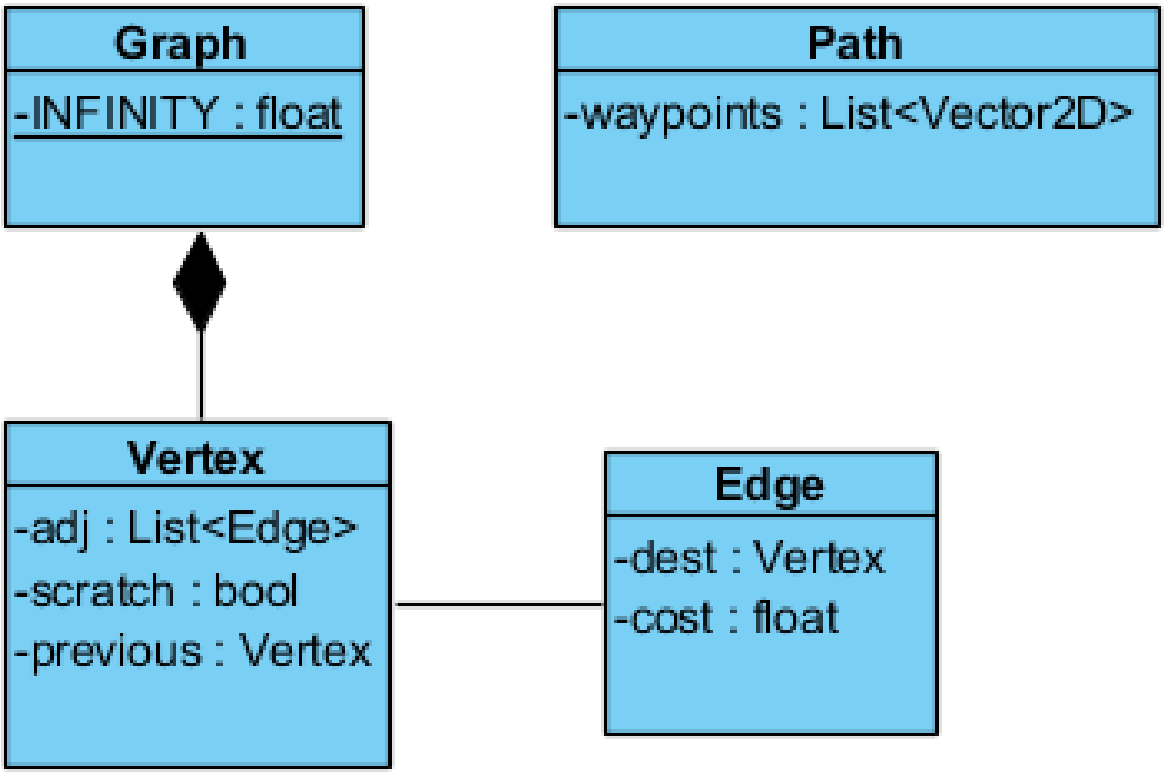


Figure 11: Class diagram of classes that are relevant to path planning.

4 Behavior

5 Fuzzy Logic

6 Conclusion

We would like to think we did ok for this project, although there are many things that could be improved upon.

Some examples of what we think could be improved upon are the implementation of fuzzy logic. We currently have strangely formed FLV's that do however seem to work. When changing the graphs to be a normal format we have a bug where the defuzzification of the distance to the enemy always seems to return 0 and therefore we decided to keep the graphs the way they were.

We also think the enemy/entity implementation we have could be improved on. We only later discovered the source found within the book and we already had the enemy-hierarchy implemented. after that we decided to implement the entity-hierarchy like the book did but the combination of both creates awkward situations such as the eagle not being an Enemy.

We think the main difficulties we had with the project were finding about the book source too late and partly because of that, not being ready for the deadline. After the deadline we both were quite busy with the game project and had to work on the Tower Defense game during the weekends mainly which isn't a great excuse but it lead to us taking so long to hand this in.

On the more positive side we think we implemented most required features or atleast tried to implement them, We feel like we have some good parts within the code unlike the enemy-entity system. Such as the Powerup hierarchy for the explore and being able to specify a Queue of Powerups for the Flying Entity to explore, it would be very easy to expand this to include different kinds of Powerups. We also think we did a decent job on commenting the code and staying consistent with our conventions.

References

[1] *GitLab: Issue Board.*
https://docs.gitlab.com/ee/user/project/issue_board.html

[2] *StackExchange.*
https://gamedev.stackexchange.com/questions/44400/arrive-steering-behavior