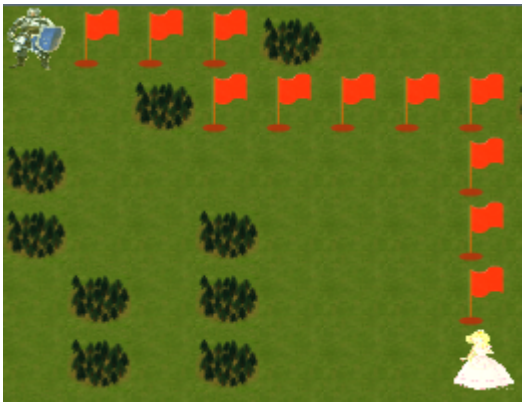


AI - Opdracht 1

A* in XNA

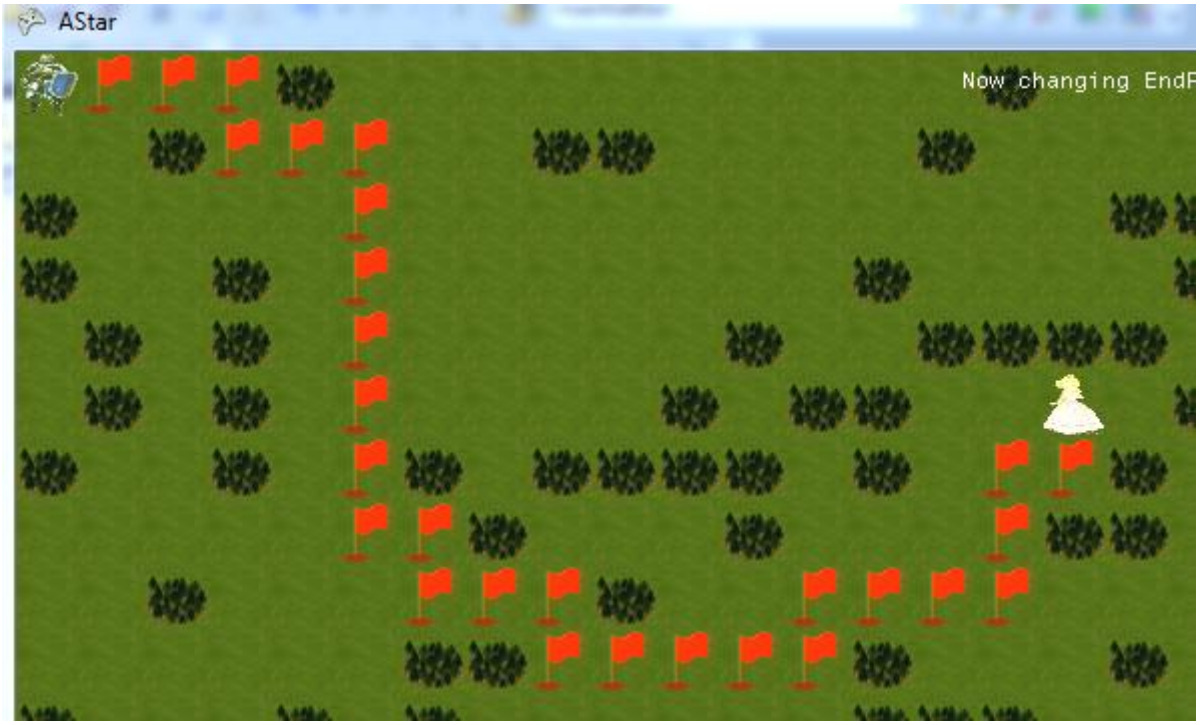
Simon Karman - 500621839
Jorn Theunissen - 500621813
8-2-2012

Inhoud

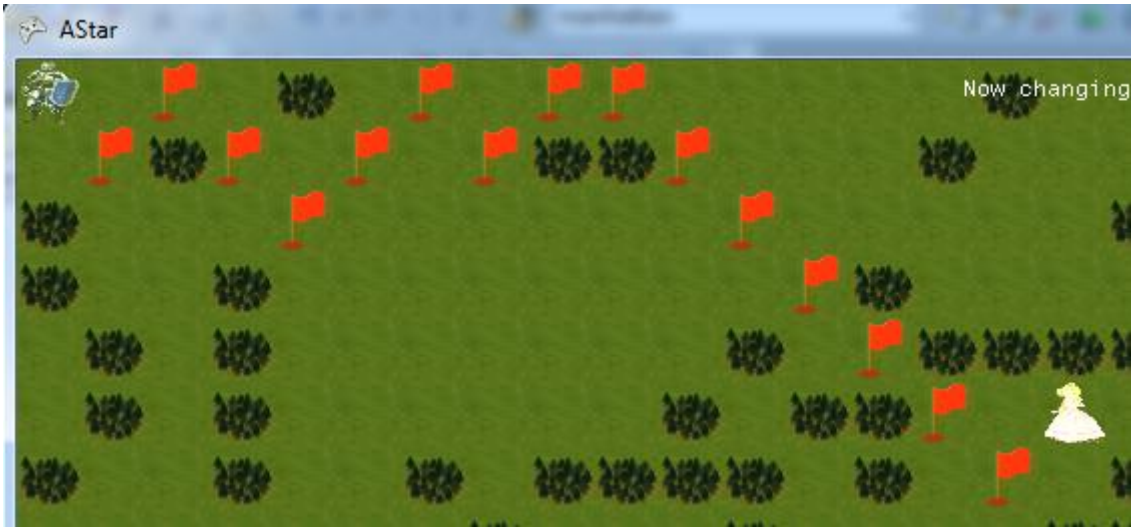
Deel 1

De applicatie die wij hebben gemaakt is gecreëerd in XNA. In de applicatie is het de bedoeling dat de ridder het kortste pad naar de prinses vind. Dit doet hij door middel van A* pathfinding. We hebben gebruiken 2 soorten datastructuren om gegevens bij te houden tijdens het pad zoeken. Daarnaast hebben we zowel horizontaal-verticaal als horizontaal-verticaal-diagonaal er ingebouwd. hieronder staan 2 plaatjes. Beide plaatjes hebben het zelfde eind als start doel.

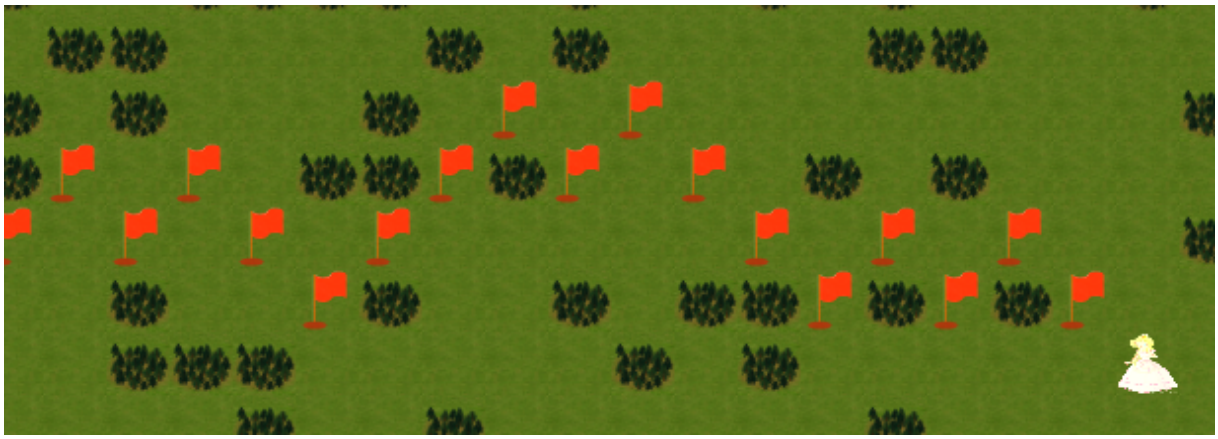
Alleen horizontaal en verticaal:



Horizontal, verticaal en diagonaal:



Bij het diagonaal zoeken komen we wel nog bij een nadeel van A* uit: zig zaggen



Deel 2

Vraag 1

De gebruikte data structuren zijn:

- Heap
- Unsorted List

Vraag 2

Manhattan uitgelegd:

```
private float ManhattanDistance(int index1, int index2)
{
    Vector2 pos1 = IndextoXY(index1);
    Vector2 pos2 = IndextoXY(index2);
    return (Math.Abs(pos1.X - pos2.X) + Math.Abs(pos1.Y - pos2.Y)) * 0.1f;

}
```

We geven de huidige en eind positie mee aan de functie. We converteren de variabelen om naar 2d vectoren. Vervolgens trekken we de x position van elkaar af. Dit doen we ook voor de y posities. Vervolgens tellen we deze 2 berekeningen bij elkaar op. Met Math.abs zorgen we er voor dat er altijd een positief getal is.

Vraag 3

De Lists:

```
////////////////////////////////////////////////
Boolean found = false;
List<int> open = new List<int>();
List<int> closed = new List<int>();
open.Add(indexStart);
int currentNode = -1;
while (open.Count != 0)
{
    currentNode = GetLowestCost(open.ToArray());
    if (currentNode == indexEnd)
    {
        found = true;
        break;
    }
    else
    {
        open.Remove(currentNode);
        closed.Add(currentNode);
        int[] adjacentNodes = GetAdjacentNodes(currentNode);
        foreach (int adjacentNode in adjacentNodes)
        {
            if ((adjacentNode >= 0) && (adjacentNode < Width * Height))
            {
                if (pathTiles[adjacentNode].walkable)
                {
                    if (!open.Contains(adjacentNode))
                    {
                        if (!closed.Contains(adjacentNode))
                        {
                            open.Add(adjacentNode);
                            pathTiles[adjacentNode].cost = pathTiles[currentNode].cost + 1;
                            pathTiles[adjacentNode].heurestic = ManhattanDistance(adjacentNode, indexStart);
```

De Lists 'Open' en 'Closed' zijn heel gemakkelijk om bij te houden te gebruiken bij A*. Dit komt vooral om de lists zich niks aantrekken van de aantallen die worden opgeslagen in de list. Zodra de start tile is toegevoegd aan de list, begint de while die er voor zorgt dat er een pad gevonden gaat worden. Ook bij het vinden van de adjacent Nodes gebruiken we een list.

De heap:

```
Boolean found = false;
Heap<PathTile> open = new Heap<PathTile>();
List<int> closed = new List<int>();
open.Push(pathTiles[indexStart]);
int currentNode = -1;
while (open.Count != 0)
{
    PathTile shortest = open.Pop();
    currentNode = shortest.GetIndex();
    if (currentNode == indexEnd)
    {
        found = true;
        break;
    }
    else
    {
        closed.Add(currentNode);
        int[] adjacentNodes = HeapGetAdjacentNodes(currentNode);
```

Veel veranderd er eigenlijk niet in de code. De Heap pathfinder heeft van alle methoden zijn eigen kopie (alleen met het woordje 'heap' ervoor.) Het grote voordeel van de heap is dat de hoeveelheid zoektijd veel meer beperkt blijft. Dit om dat er een stuk minder gezocht hoeft te worden in de unsorted list maar gewoon de bovenste waarde uit de array gepopt kan worden.

Vraag 4

Het verschil tussen de Big O van een binary heap implementatie en de implementatie van een unsorted list is dat bij gebruik van een implementatie van een unsorted list de hele unsorted list moet worden doorzocht. Het ophalen van de korste node in deze implementatie van een unsorted list kost Big O (N) tijd. Het toevoegen is echter heel snel omdat dit aan het einde van de array kan en kost dus Big O (1) tijd.

Nu hebben wij (geheel toevallig) ook een binary heap geimplementeerd. In die binary heap kan je met Log N toevoegen en met Log N ophalen. De binary heap implementatie is daarom dus iets sneller dan de implementatie van de Unsorterd List.

Hieronder is dit nog eens verwerkt in een tabel.

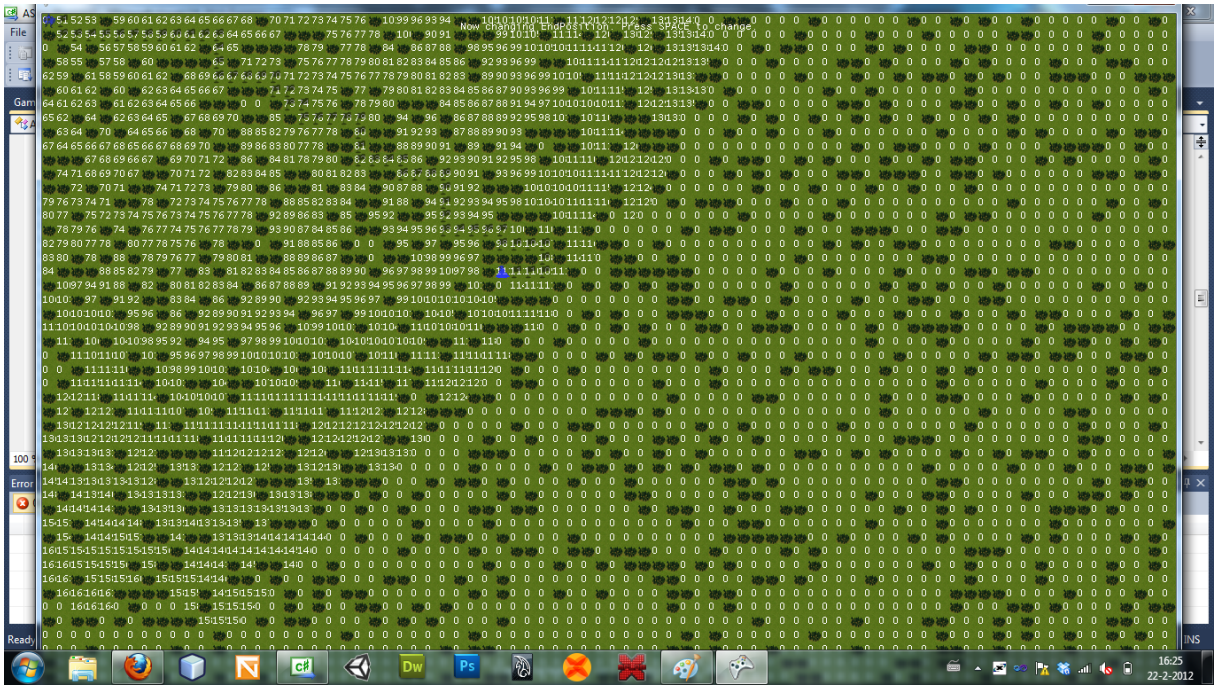
Taak	**Unsorted List**	**Binary Heap**
Toevoegen	Big O (1)	Big O (Log N)
Ophalen van de korste node	Big O (N)	Big O (Log N)

Vraag 5

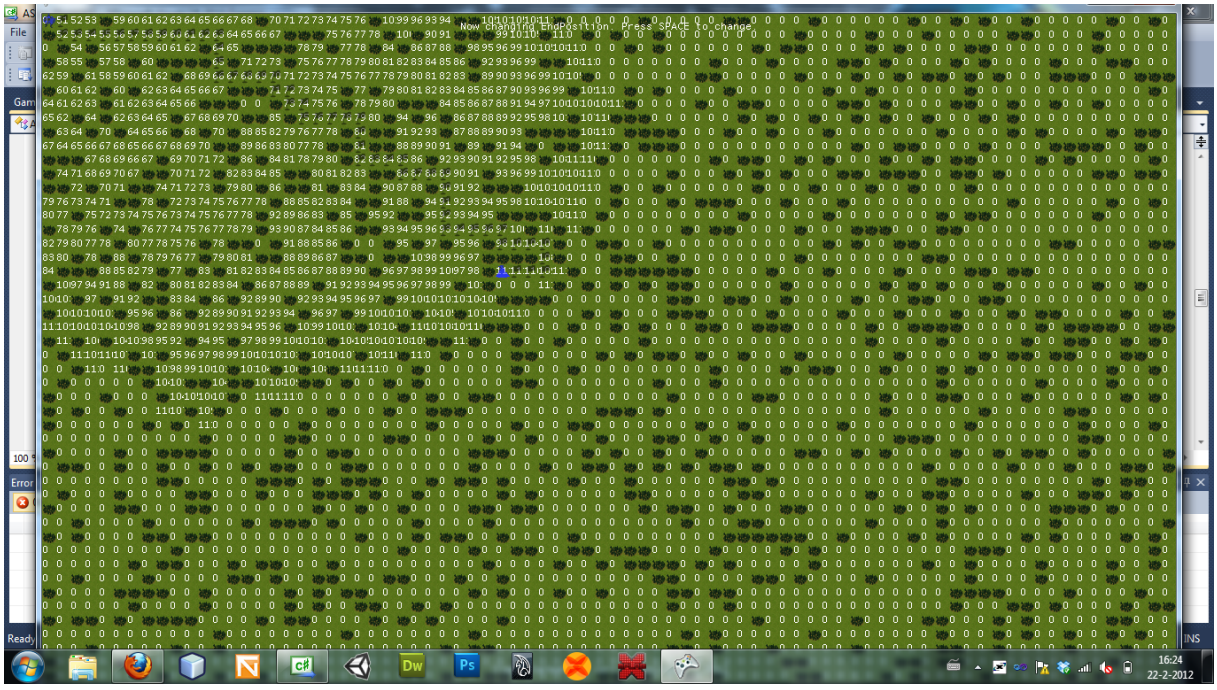
A* en dijkstra in een vergelijking. De start en eindpunten zijn hetzelfde. De getallen per veld laten zijn wat de total kosten zijn. Velden met een 0 zijn niet door het algoritme aangeraakt.

Test 1

Dijkstra:

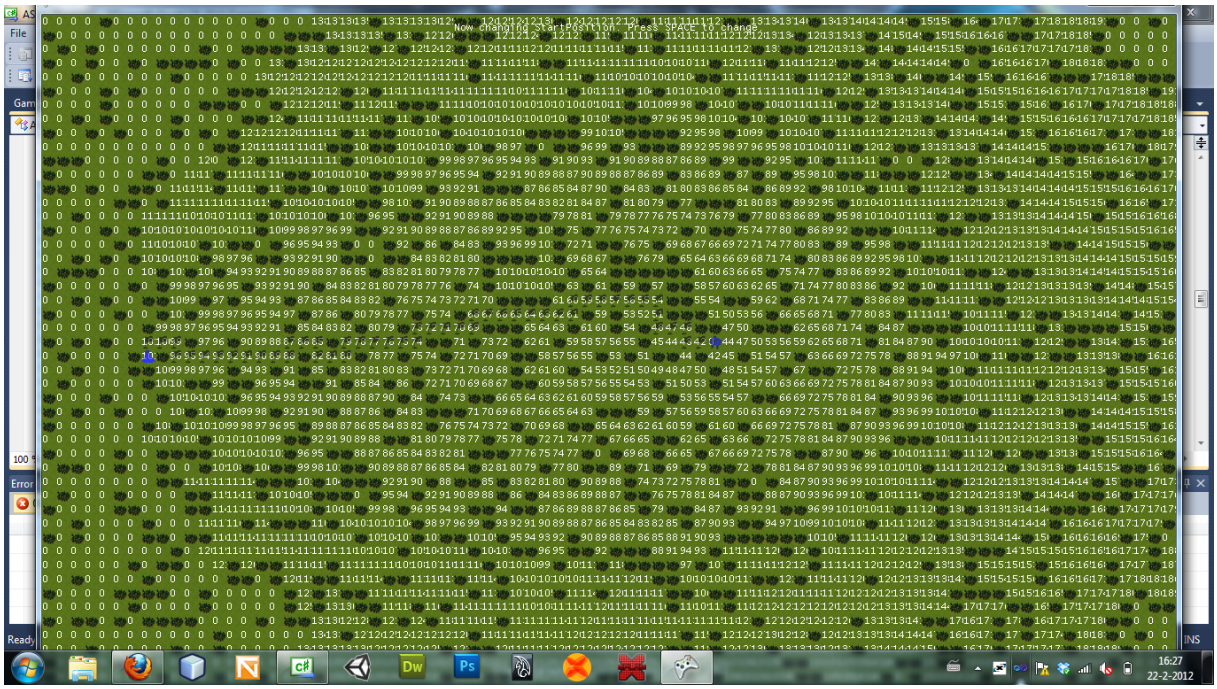


A*:

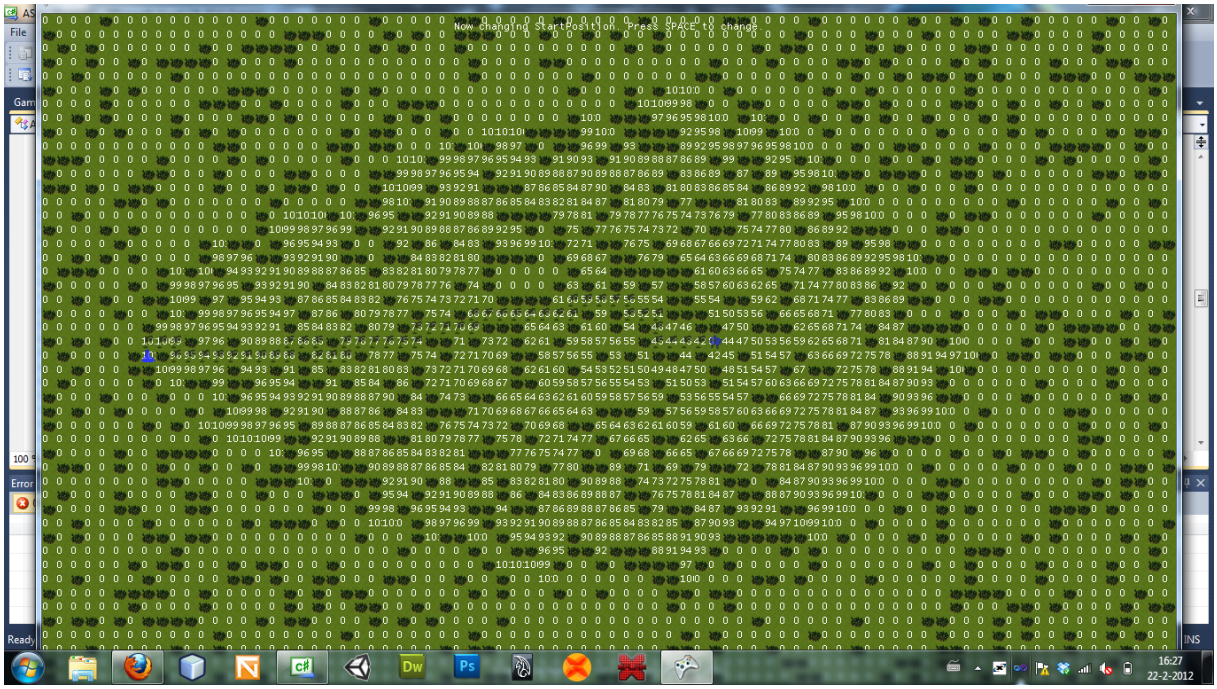


Test 2

Dijkstra:



A*:



Test 3

Diagonal check. Deze screenshot laat zien hoe goed het A* algoritme werkt wanneer diagonal movement is toegestaan.

Vraag 6

Wereld	Heap	Array/List
20x20	1 ms	1 ms
40x40	16 ms	13 ms
80x80	194 ms	69 ms
160x160	208 ms	520 ms
320x320	221 ms	6347 ms