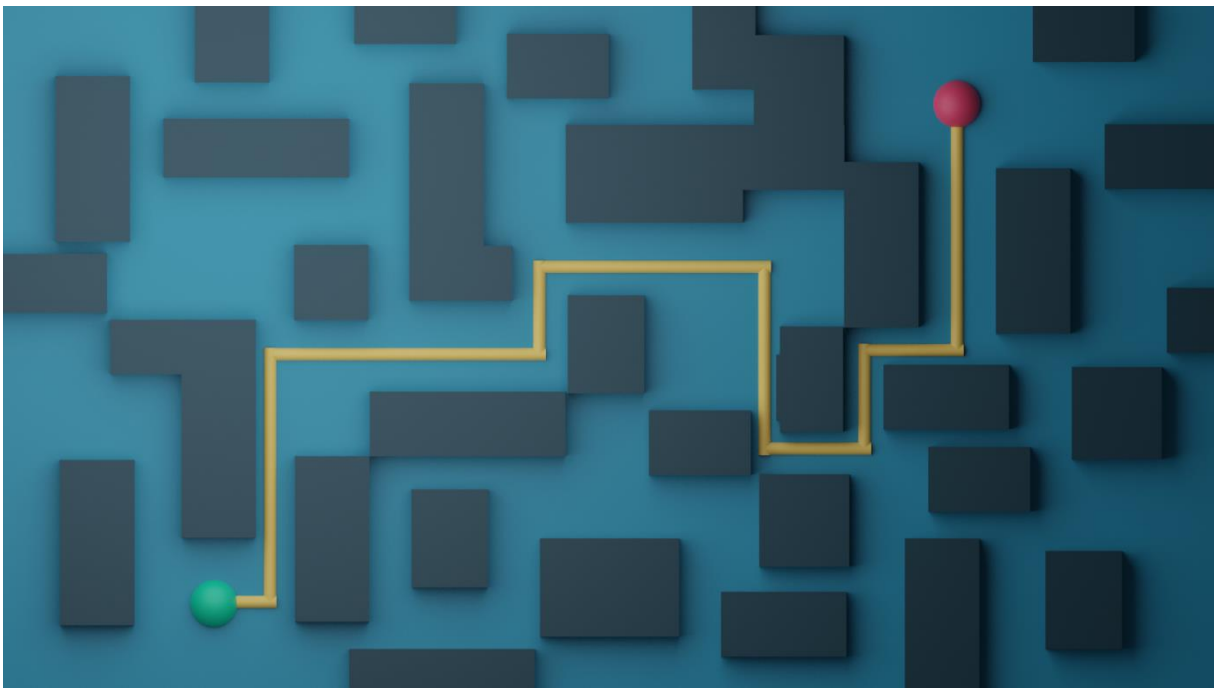
Hoe wordt pathfinding toegepast in videospellen?



Robin Schellemans
6D – Wetenschappen
Wiskunde
Scheppersinstituut Mechelen
2020-2021
Promotor: Joke Van Nuffel

Hoe wordt pathfinding toegepast in videospellen?

Robin Schellemans
6D – Wetenschappen
Wiskunde
Scheppersinstituut Mechelen
2020-2021
Promotor: Joke Van Nuffel

Voorwoord

Ik ben op zoek gegaan naar een manier om pathfinding toe te passen op videospellen omdat ik dit zelf nodig had. Ik ben namelijk een game aan het maken waar je vijanden moet verslaan. Deze vijanden moeten de weg vinden naar de speler. Door dit eindwerk heb ik veel bijgeleerd over artificiële intelligentie en algoritmes.

Ik wil mijn promotor bedanken die enthousiast was toen ze hoorde wat ik van plan was en mij steunde door dit hele proces, door opbouwende feedback en mij in de goede richting te leiden.

Ik wil ook mijn familie bedanken die altijd aandachtig naar mij hebben geluisterd toen ik bezig was over al het complexe van programmeren. Zelfs als ze hier niets van begrepen, waren ze een goed luisterend oor. Ik wil ze ook bedanken dat ze me bleven steunen en aanmoedigen. Ook wil ik ze bedanken om mijn eindwerk na te lezen en mijn kleine foutjes er uit te halen.

Inhoudsopgave

Inleiding

In dit eindwerk ga ik kijken hoe pathfinding wordt toegepast in videospellen. Pathfinding is belangrijk in veel computerspellen, omdat de niet speelbare karakters een weg moeten vinden in het spel. Ik ga onderzoeken wat pathfinding is en welke andere toepassingen het heeft buiten videogames. Ik ga nagaan of pathfinding een vorm van artificiële intelligentie is, hoe het in elkaar zit en op welke manieren dit geprogrammeerd kan worden en de logica hierachter. De focus van dit eindwerk ligt op de toepassing van pathfinding in videospellen en welke manier hier de beste voor is. En de verbeteringen die er op het basis pathfinding-algoritme gemaakt kunnen worden.

Tenslotte ga ik pathfinding toepassen op een computerspel dat ik aan het maken ben. Door dit te doen ga ik een beter beeld krijgen over hoe het toegepast wordt en welke uitdagingen hierbij komen kijken. Voor het pathfinding algoritme gebruik ik het algoritme van Sebastian Lague, hij is dan ook mijn grootste bron. Ik ga dit toepassen door Unity te gebruiken en C#, want dit zijn de engine[1] en taal die ik gebruik voor mijn spel.

[1] Een game engine, hier kan een game in gemaakt worden.

Hoofdstuk 1 Wat is pathfinding en voor wat wordt het gebruikt?

In dit hoofdstuk leg ik uit wat pathfinding is en of het een vorm van artificiële intelligentie is. Ik ga de toepassingen van pathfinding bespreken en waarom het gebruikt wordt in videospellen.

1.1. Wat is pathfinding?

“The plotting by a computer application of the best route between two points” (pathfinding, 2019) is een definitie voor pathfinding. Pathfinding is dus het bepalen van de beste route tussen twee punten m.b.v. een computerapplicatie. De beste route is afhankelijk van de situatie, bijvoorbeeld de kortste route of de route met de minste obstakels. Met menselijke intelligentie is het een makkelijke opdracht om een pad te vinden, maar voor een computer is het een moeilijke opdracht. Bij pathfinding gaat de computer dus iets doen waar menselijke intelligentie voor nodig is. Is het dan een vorm van artificiële intelligentie? Een definitie van artificiële intelligentie is:

> Kunstmatige intelligentie (KI) of artificiële intelligentie (AI) is de wetenschap die zich bezighoudt met het creëren van een artefact dat een vorm van intelligentie vertoont. (Kunstmatige intelligentie, 2020)

Dit kan terugggevonden worden in een computerprogramma dat iets uitvoert dat normaal gezien alleen kan uitgevoerd worden via menselijke intelligentie. Er zijn vier verschillende types van AI, namelijk reactieve AI, beperkt geheugen, theory of mind en zelfbewustzijn (Bauvois, 2017).

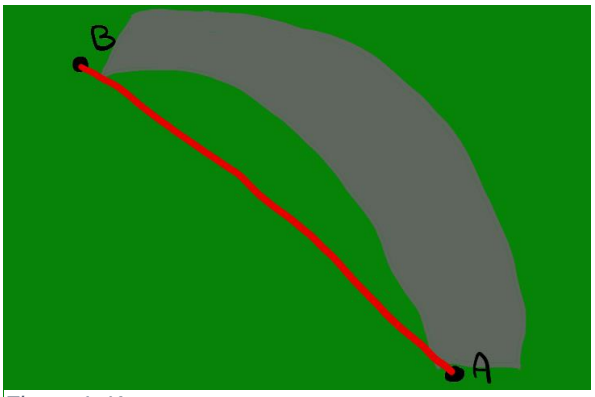
Een voorbeeld van AI is rijpe aardbeien onderscheiden van onrijpe aardbeien. Vroeger konden alleen mensen dit. Maar met artificiële intelligentie kan een computer dit nu ook. Om een computer dit te laten doen, moet het eerst leren wat een rijpe en een onrijpe aardbei is. Voor een computer duurt het langer om dit aan te leren dan voor een mens. Ten eerste moet een computer een aardbei kunnen herkennen in een foto voordat het kan zien of de aardbei rijp of onrijp is. Een computer is een complexe wiskundige machine. Wiskunde is een exacte wetenschap en de natuur is niet exact. De computer kan het leren door verschillende foto’s van aardbeien te analyseren

waarbij de aardbei is aangeduid. Hoe meer foto's de computer ziet, hoe beter hij op een nieuwe foto een aardbei zal kunnen aanduiden. Deze vorm van artificiële intelligentie wordt Machine learning genoemd en wordt vaak gebruikt voor toepassingen waarbij de computer iets moet kunnen herkennen. Dit voorbeeld is een vorm van AI met een beperkt geheugen (Puttemans, 2017).

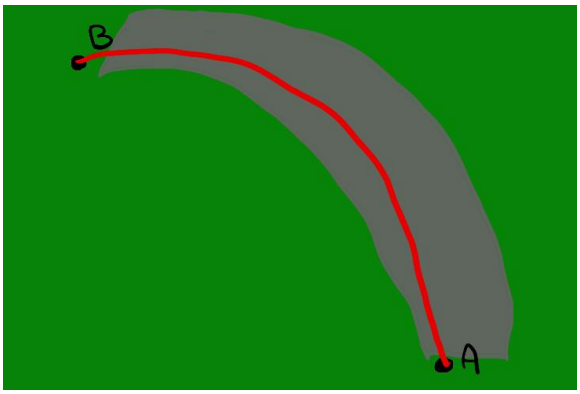
Bij pathfinding daarentegen gebruikt de computer geen geheugen maar analyseert het de situatie en neemt dan een beslissing. Dit is een vorm van reactieve AI. Een voorbeeld hiervan is een computer die schaakt tegen een mens. De computer heeft geleerd wat de verschillende schaakstukken zijn en hoe die mogen bewegen. De computer analyseert het schaakbord elke beurt en gaat per beurt beslissen wat de beste zet is, door uit te rekenen met welke zet hij het meeste kans heeft om te winnen (Knapton, 2017). Dit is ook een vorm van AI, want de computer moet denken zoals een mens en moet kunnen reageren op verschillende situaties.

Pathfinding gebruikt deze vorm van AI, want de computer zal afhankelijk van een situatie moeten beslissen welke route hij best kan nemen. In sommige situaties willen we dat we de kortste weg vinden van punt A tot punt B. In andere situaties is de kortste weg niet altijd de beste. In een spel waarbij een team strategisch moet bewegen, zullen er andere wegen gevolgd moeten worden. Hierbij gaan er ook andere factoren de paden beïnvloeden, dit maakt het dan complexer. Door het complexer te maken gaat het ook meer beginnen lijken op hoe een mens denkt. We kunnen het niet alleen strategischer maken, maar ook natuurlijker. Bij een weg door een grasgebied is het logisch de weg te volgen, ook al is dit niet de kortste weg.

Zoals je ziet op figuur 1 is het kortste pad van A naar B door het gras, maar in een natuurlijke situatie volg je de weg. We kunnen dus een factor bijvoegen en de computer zeggen dat het beter is om wegen te nemen zoals op figuur 2 (Lague, 2015).



Figuur 1: Kortste weg



Figuur 2: Efficiëntste weg

Pathfinding is een algoritme, maar wat is dit? Een algoritme is een lijst met stappen die de computer moet uitvoeren, zoals een recept. Een algoritme wordt geschreven door een programmeur, deze geeft de computer verschillende opdrachten of commando's. Dit doet een programmeur aan de hand van een taal die de computer kan begrijpen. Programmeurs gaan hun algoritme niet in deze taal schrijven, maar in een taal die dan wordt omgezet in een lijst met stappen die de computer wel begrijpt. Er zijn verschillende soorten programmeertalen met elk hun eigen toepassingen. In dit eindwerk ga ik werken met C#, een taal die onder anderen gebruikt wordt om desktop applicaties en videospellen te maken. Pathfinding is dus een stappenplan om een weg te vinden. Hoe dit stappenplan in elkaar zit ga ik in hoofdstuk 2 uitleggen. Hieronder zie je een voorbeeld van hoe deze taal eruit ziet.

```
namespace HelloWorld
{
   class Program
   {
      static void Main(string[] args)
      {
         string message = "Hello World!!";

         Console.WriteLine(message);
      }
   }
}
```

1.2. Voor wat wordt pathfinding gebruikt?

Pathfinding wordt onder andere gebruikt door robots. Hier wordt het gebruikt zodat de robot zelf een weg kan vinden en niet alleen bestuurd moet worden door een mens.

Een robotstofzuiger gebruikt pathfinding om zijn weg te vinden door een huis. Hierbij onthoudt de robot ook wat hij al heeft gestofzuigd en wat niet. Bij deze stofzuigers zijn er ook apparaten die een zone kunnen afbakenen, de robot houdt hier dan rekening mee en vermijdt de zone (Edwards, 2018).

Pathfinding wordt ook gebruikt voor applicaties zoals gps-systemen. Google Maps gebruikt Dijkstra's Algoritme om de snelste weg te vinden. Het houdt rekening met verschillende factoren. Files en wegenwerken worden mee in het algoritme berekend, zodat de applicatie de snelste weg kan voorzien (Kim, 2019). Pathfinding wordt in ons dagelijks leven gebruikt, maar het kan ook gebruikt worden voor recreatie, namelijk in videospellen.

In dit werk ga ik me vooral focussen op pathfinding in videospellen omdat ik pathfinding in mijn eigen game wil gebruiken. Pathfinding wordt tegenwoordig bijna in alle spellen gebruikt, de meest voorkomende toepassing hierbij is pathfinding bij de vijand. De vijand moet namelijk een weg kunnen vinden naar de speler. Bij sommige spellen is het voldoende dat de vijand de kortste weg vindt naar de speler, maar bij andere spellen moet het vijandelijk team strategisch naar de speler gaan. Hoe beter de strategische bewegingen zijn hoe uitdagender en leuker het spel wordt.

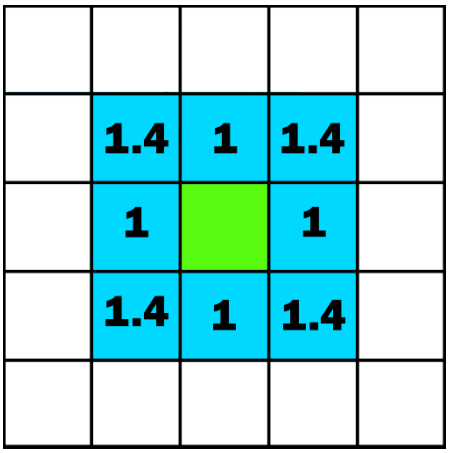
1.3. Soorten pathfinding algoritmes

1.3.1 Dijkstra's algoritme

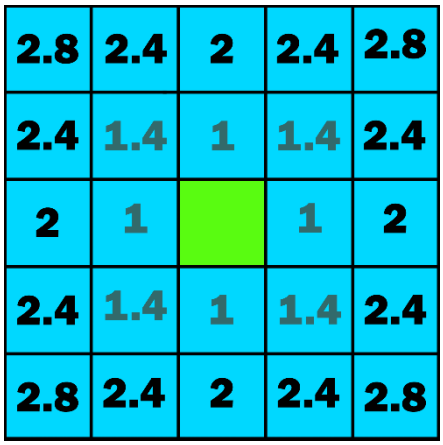
Er zijn verschillende soorten algoritmes voor pathfinding, sommige zijn sneller dan andere. Het meest gebruikte algoritme is het A* algoritme, maar dit algoritme is gebaseerd op een ander pathfinding algoritme genaamd Dijkstra's algoritme.

Om een algoritme snel te laten werken, kan een rooster gebruikt worden. De grootte van de vakken van het rooster wordt vooraf bepaald, afhankelijk van de grootte van de oppervlakte waar pathfinding op wordt toegepast. Dijkstra's algoritme gaat voor elk vak rond het startpunt een waarde toevoegen. Voor de vierkanten die direct grenzen aan het beginpunt wordt er plus 1 gedaan en voor de diagonale vakken plus $\sqrt{2}$ (stelling

van Pythagoras). Om het gemakkelijker te maken ga ik 1,4 gebruiken in mijn voorbeelden, zoals te zien is op figuur 3. Dit proces wordt dan herhaald voor elk nieuw vak, zoals te zien is op figuur 4.

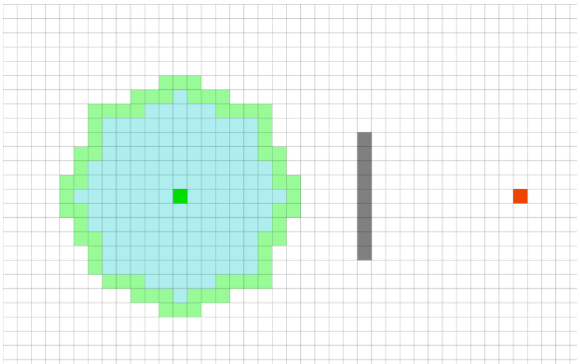


Figuur 3: Dijkstra's algoritme eerste iteratie

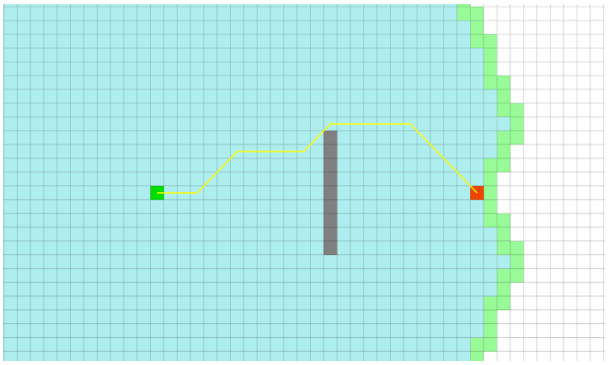


Figuur 4: Dijkstra's algoritme tweede iteratie

Er kan op verschillende manieren naar een vak gegaan worden, maar een vak krijgt altijd de kleinst mogelijke waarde. Als er dus op een nieuwe manier naar een vierkant kan gegaan worden waarbij de waarde kleiner is, wordt de waarde veranderd. De route naar elk vak wordt ook bijgehouden. Als er een vak berekend is dat overlapt met de eindbestemming, wordt de weg met de laagste waarde gekozen. Op figuur 5 en 6 is Dijkstra's algoritme voorgesteld op grote schaal. De blauwe zijn de berekende vakken en de groene vakken zijn de vakken waarrond er berekend wordt.(Lague, 2015).



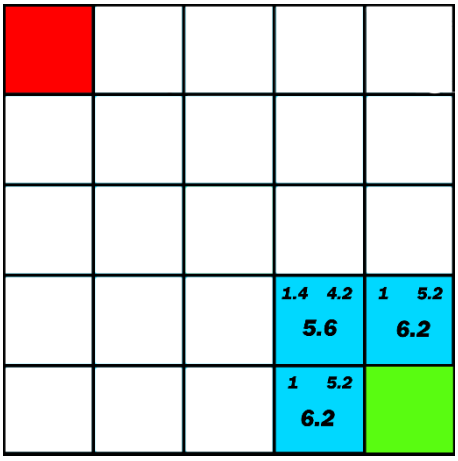
Figuur 5: Dijkstra's algoritme op grote schaal



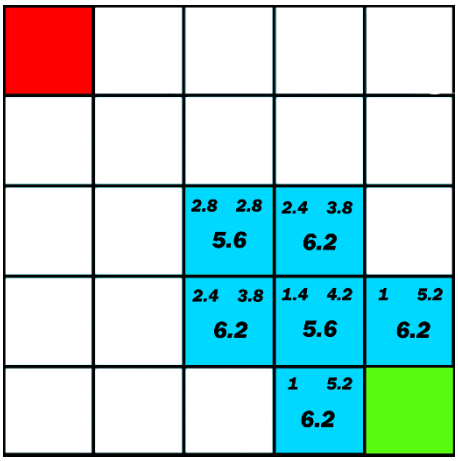
Figuur 6: Dijkstra's algoritme op grote schaal

1.3.2 A* algoritme

A* is gebaseerd op Dijkstra's algoritme maar werkt efficiënter. Dijkstra's algoritme berekent alle vakken en dit vergt veel tijd. A* gaat zo min mogelijk vakken berekenen. Alleen vakken rond het vak dat het dichtst bij het begin- en eind vak ligt, worden berekend. Er worden per vak drie waarden bepaald, namelijk de afstand tot het beginpunt, tot het eindpunt en de som van deze twee getallen. De afstand tot het beginpunt wordt op dezelfde manier berekend als bij Dijkstra's algoritme. De afstand tot het eindpunt wordt in een rechte lijn berekend, niet rekening houdend met obstakels. Op figuren 7 en 8 is de waarde linksboven de afstand tot het beginpunt en rechtsboven de afstand tot het eindpunt. In het midden zien we de twee waarden opgeteld. Het vierkant met de laagste waarde in het midden wordt gekozen om verder rond te rekenen.

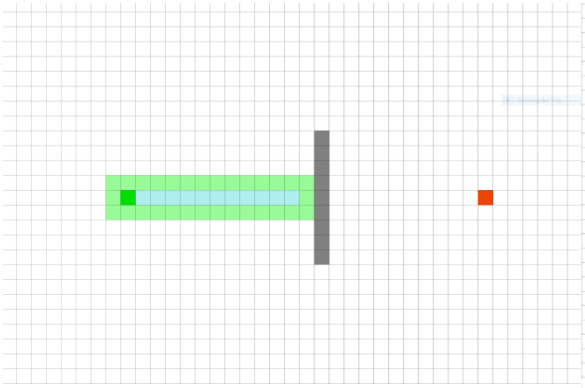


Figuur 7: A eerste iteratie*

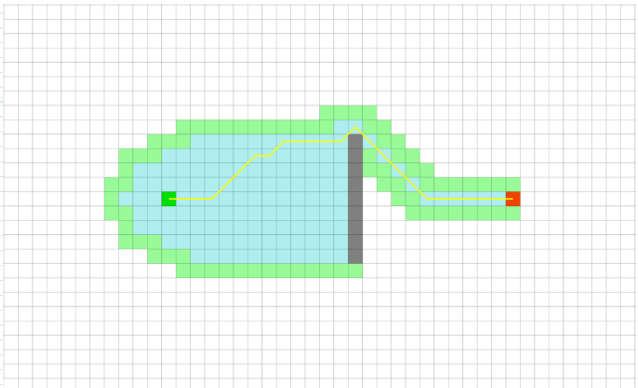


Figuur 8: A tweede iteratie*

Op grote schaal ziet dit er dan uit zoals op figuur 9 en 10. In vergelijking met figuren 5 en 6 is te zien dat bij A* veel minder vakken berekend worden (Lague, 2015).



Figuur 9: A op grote schaal*



Figuur 10: A op grote schaal*

1.4. Hoe maakt pathfinding videospellen beter?

Het A* algoritme kan gebruikt worden in een spel waarbij de speler verschillende levels moet voltooien, met in elk level vijanden die verslagen moeten worden. In de meeste situaties gaan de vijanden de speler aanvallen. Om dit te kunnen doen, moeten ze een weg vinden naar de speler. Hier komt het A* algoritme dus goed van pas. Maar omdat dit algoritme altijd de kortste weg gebruikt, maken de vijanden altijd dezelfde bewegingen. Als de speler het level een paar keer opnieuw heeft gespeeld, zal de speler zich al rap de exacte bewegingen van de vijanden herinneren. Dit zorgt ervoor dat de speler het level uitspeelt omdat hij de paden van de vijand kent en niet omdat hij beter is geworden in het spel.

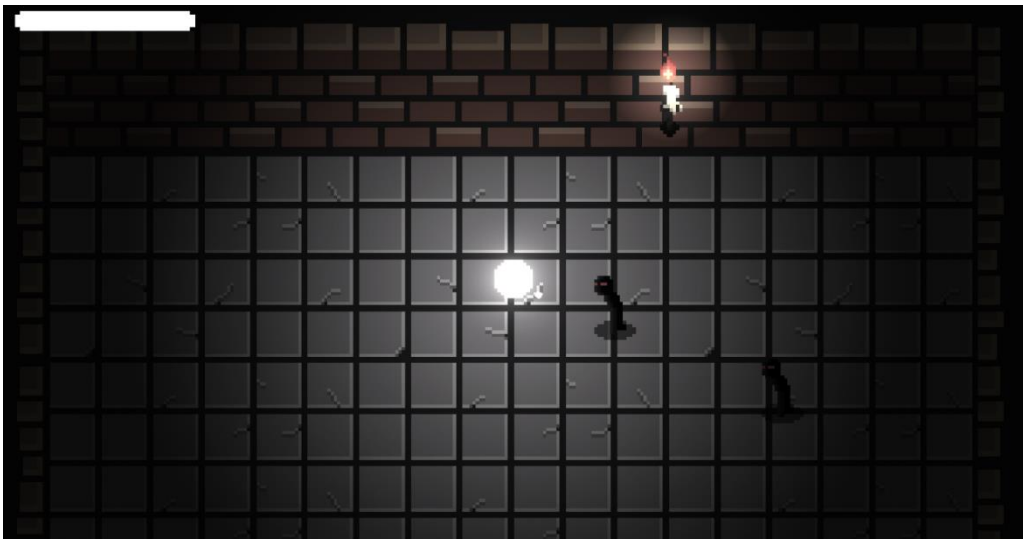
Dit probleem kan opgelost worden door niet enkel de kortste weg te berekenen, maar ook andere paden die ook snel zijn. Dit zorgt ervoor dat er altijd variatie is in een level, waardoor de speler beter in het spel moet worden om een level te verslagen. Deze variatie zorgt er ook voor dat de levels opnieuw gespeeld kunnen worden, waardoor de speler langer van het spel kan genieten (John, 2008).

Hoofdstuk 2 Onderzoek

In dit hoofdstuk ga ik uitleggen hoe ik pathfinding heb toegepast op mijn eigen videospel. Door dit onderzoek te doen kreeg ik een beter beeld over hoe pathfinding nu eigenlijk wordt toegepast in videospellen. Omdat het voor mij te moeilijk is om zelf een algoritme te schrijven, ga ik het algoritme van iemand anders gebruiken. Dit algoritme heeft de auteur openlijk op het internet gezet voor iedereen om gebruik van te maken. Zijn code gebruiken en op mijn game toepassen is een beetje te gemakkelijk voor een onderzoek. Dus ga ik onderzoeken hoe dat zijn algoritme in elkaar zit. Elk spel is anders en er gaan dus verschillende toepassingen zijn per videospel. Dus is het goed om te weten hoe de algoritmes in elkaar zitten om daar dan nog verbeteringen op te maken specifiek aan een spel. Mijn videospel is gemaakt in een computerapplicatie om spellen in te maken, namelijk Unity.

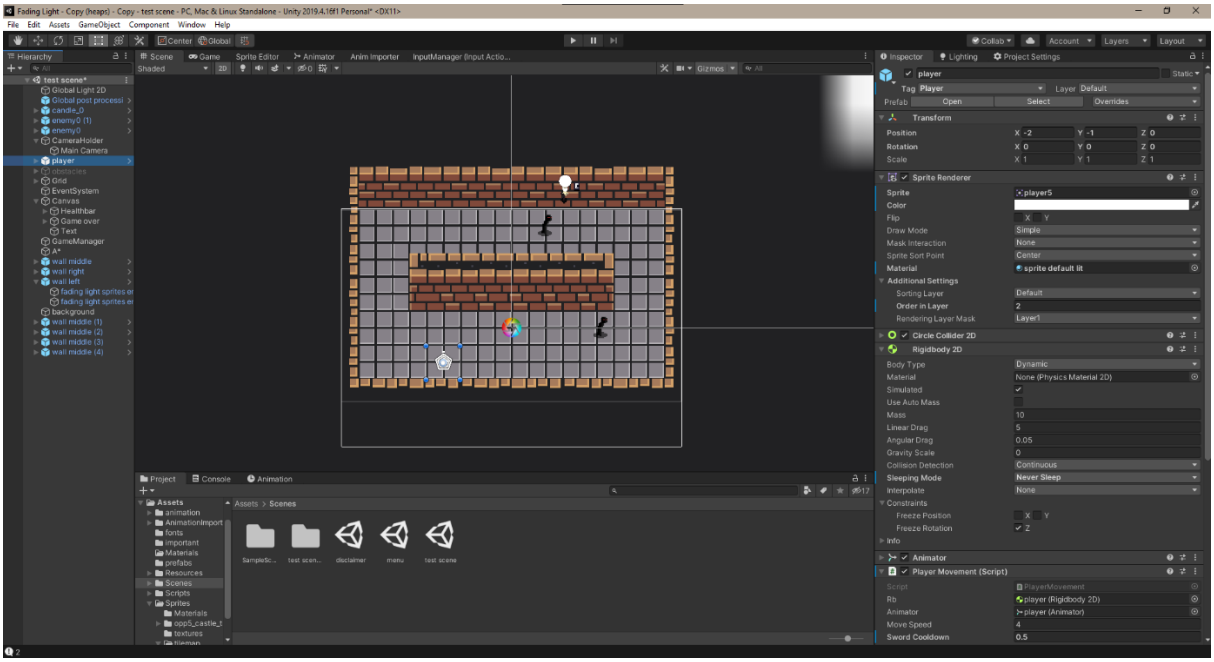
2.1. Mijn videospel

Vooraleer ik ga uitleggen hoe ik pathfinding toepas op mijn eigen videospel, zal ik uitleggen wat mijn spel inhoudt. Mijn spel is nog niet af maar wel klaar en geschikt om pathfinding op toe te passen. Ik ben een 2D spel aan het maken waarbij de speler het laatste bolletje licht is in een wereld waar schaduwen de wereld donker hebben gemaakt. Je moet als speler deze schaduwen bevechten met een lichtzwaard. Als de vijanden gewoon stil staan, is dit niet moeilijk. Dus heb ik pathfinding nodig om de vijanden naar de speler toe te laten gaan. Op figuur 11 zie je mijn spel met het lichtbolletje als speler en twee zwarte wezentjes als de vijanden.



Figuur 11: Mijn videospel

Deze game is gemaakt in een game engine: dit is een software die je kan gebruiken om games te maken. In een game engine zijn er al veel dingen ingebouwd waardoor het makkelijker wordt om een game te maken. Bekende game engines zijn Unreal Engine en Unity, deze laatste heeft veel functies voor 2D spellen vandaar dat ik voor deze engine gekozen heb. Op figuur 12 kan je zien hoe Unity eruit ziet. Je kan ook zelf



Figuur 12: Unity

een game Engine maken dit vergt veel tijd maar je hebt uiteindelijk wel controle over alles.

In het midden van het scherm zie je de objecten in het spel zoals de speler, de vijand en de omgeving. Links kan je alle objecten zien die er in de game zijn, en rechts zie je de scripts die van toepassing zijn op een object. Scripts zijn bestanden met code. Unity heeft zelf scripts zoals basis fysica scripts die je kan gebruiken om bijvoorbeeld licht of zwaartekracht te simuleren. Je kan ook je eigen code schrijven en die dan op een object toepassen. De programmeertaal die Unity gebruikt is C#. Onderaan kan je al de bestanden in je game zien zoals de scripts en foto's.

2.2. Pathfinding Algoritme

Zoals eerder vermeld, is mijn kennis nog te beperkt om zelf een algoritme te verzinnen, dus ga ik het algoritme van Sebastian Lague gebruiken. Veel programmeurs gaan

eerst een zogenaamde pseudocode schrijven, dit lijkt op code maar werkt niet. Dit is een soort van schema zodat de programmeur kan zien wat hij moet doen. De pseudocode van dit algoritme kan je hieronder zien.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
    current = node in OPEN with the lowest f_cost
    remove current from OPEN
    add current to CLOSED

    if current is the target node //path has been found
        return

    foreach neighbour of the current node
        if neighbour is not traversable or neighbour is in CLOSED
            skip to the next neighbour

        if new path to neighbour is shorter OR neighbour is not in OPEN
            set f_cost of neighbour
            set parent of neighbour to current
            if neighbour is not in OPEN
                add neighbour to OPEN
```

Er is een lijst met nodes die moeten geëvalueerd worden (OPEN) en een lijst met nodes die al geëvalueerd zijn (CLOSED). Nodes zijn variabelen die gelinkt zijn: elke node houdt zijn eigen data bij en ook informatie over de volgende node. Omdat we in ons rooster vakjes moeten berekenen aan de hand van andere vakjes gebruiken we nodes. Je kan dus zo een vakje in het rooster voorstellen met een node.

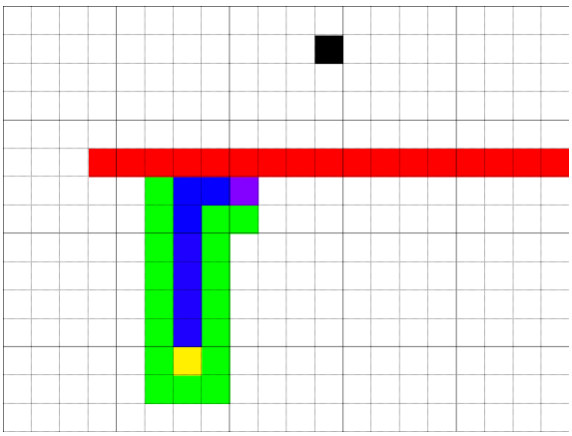
Eerst wordt de startpositie toegevoegd aan de lijst OPEN. Dan moeten we een cyclus maken met een aantal stappen. Ten eerste moeten we de node nemen met de laagste f_cost[2]. De f_cost van een node is de som van de afstand tot de startpositie en de afstand tot de eindpositie. De afstand tot het eindpunt wordt berekend in een rechte lijn zonder rekening te houden met obstakels. Dit is het verschil tussen de positie van die node en de eindpositie, dit zijn twee vectoren en kunnen we dus van elkaar aftrekken. In het begin wordt de startpositie dus gekozen als node. Dan wordt deze node verwijderd van de OPEN lijst en toegevoegd aan de CLOSED lijst, omdat de node geëvalueerd is. Nadien wordt nagekeken of we niet al bij onze bestemming zijn en als dat zo is, stopt het algoritme en geeft het pad. Dan wordt er naar elke aanliggende node gekeken als er een node is waar je niet op kunt wandelen of indien de node al geëvalueerd is, wordt deze overgeslagen. Voor de andere nodes wordt er gekeken of het nieuwe pad naar de aanliggende node kleiner is, of dat de node nog niet in de OPEN lijst is. Als het één van deze twee is, wordt de f_cost berekend voor deze node en wordt de huidige node de “ouder” van de aanliggende. Dit wordt gedaan zodat het algoritme op het einde nog kan weten hoe het pad verloopt. Als deze node nog niet in de OPEN lijst zit, wordt die ook toegevoegd. Dan begint het proces opnieuw en wordt opnieuw de node met de kleinste f_cost gekozen.

2.2.1 Voorbeeld

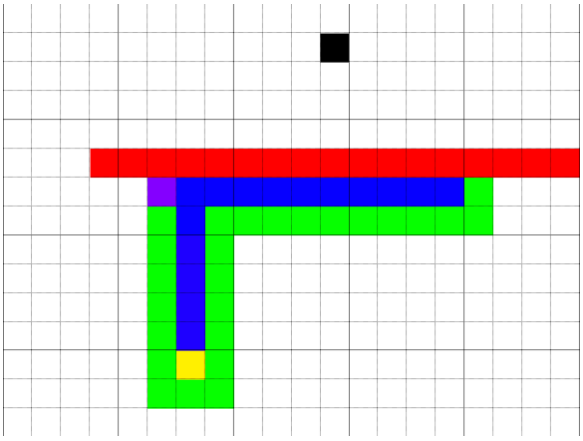
Met een voorbeeld gaat dit duidelijker worden. Op figuren 13 en 14 is het gele vakje het startpunt en het zwarte vakje het eindpunt. De groene vakjes zijn de nodes in de OPEN lijst en de blauwe in de CLOSED lijst. De rode vakjes zijn een obstakel en er

[2] f_cost is de som van de afstand tot het begin punt tot een node (g_cost) en de afstand tot het eindpunt tot die node (h_cost).

kan dus niet overgegaan worden. De huidige node wordt voorgesteld met een paarse kleur. Op figuur 13 werd de huidige node gekozen omdat die het dichts bij het eindpunt is en gaat dus de laagste f_cost hebben. Dan wordt er rond deze node gekeken. De nodes boven de huidige node zijn rood dus die worden overgeslagen en de twee



Figuur 13: A voorbeeld 1*



Figuur 14: A voorbeeld 2*

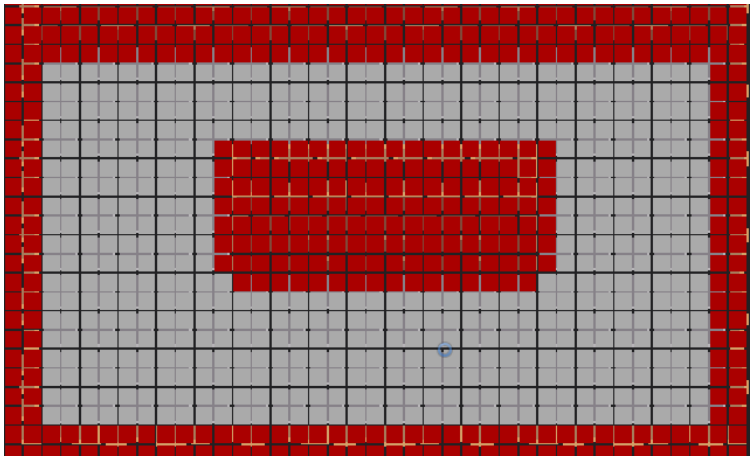
groene nodes onder de huidige worden ook overgeslagen want die zitten al in de OPEN lijst. De node rechts en rechtsonder zijn nog niet in de OPEN lijst. Dus hun f_cost wordt berekend, ze worden toegevoegd aan de OPEN lijst en de huidige node wordt hun ouder. Als dit herhaald wordt en het pad naar rechts blijft gaan, gaat de f_cost bij die nodes hoger en hoger worden, tot op het moment dat de f_cost van de groene node links boven een kleinere f_cost heeft. Zoals te zien op figuur 14 wordt dan deze node gekozen als huidige node en zo gaat het verder.

2.3. Toepassing

2.3.1 Rooster

Het pathfinding algoritme werd in vijf verschillende scripts gemaakt. Om te beginnen moet ik de scripts in mijn project toevoegen. Ik zal het eerst hebben over het script voor het rooster want dit is de eerste stap (zie bijlage 1). Dit script heeft drie waarden nodig die je op voorhand moet instellen, namelijk de hoogte en de breedte van het rooster en de grootte van een node. Dan gaat dit script berekenen hoeveel nodes er in dit rooster passen. Dit script zorgt ervoor dat we de aanliggende nodes van een node kunnen aanvragen. Mijn game is gemaakt met kleine afbeeldingen omdat ik ze getekend heb met pixels, ik moest dus de grootte van mijn nodes klein nemen zodat de pathfinding accuraat kan zijn. Op figuur 15 kan je het rooster zien met als rode vakjes de vakjes waar je niet op kan lopen. Voor dit script moest ik een nieuw object maken in mijn game, op dit object moest ik dan dit script toepassen. Zonder dit object

kan er geen rooster zijn in de game, de positie van dit object is ook belangrijk want dit is het middelpunt van het rooster. Ook moest ik definiëren wat obstakels waren door een categorie te maken en de obstakels in deze categorie te zetten. En dan deze categorie te vernoemen als onbereikbaar in het rooster script.



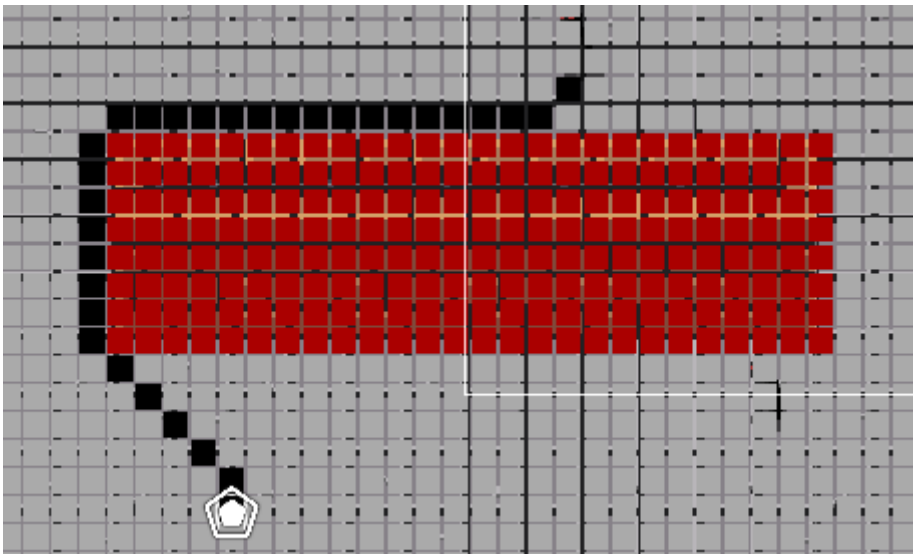
Figuur 15: Rooster

2.3.2 Nodes

Dit rooster is opgebouwd uit nodes, dus moeten we definiëren welke waarden deze nodes hebben. Dit wordt gedaan in een ander script (zie bijlage 2). In dit script wordt opgeslagen waar de node in het rooster ligt door de x en y coördinaat bij te houden. De g_cost, h_cost worden hier bijgehouden en de f_cost wordt hier berekend. Hier wordt ook de ouder van de nodes bijgehouden. De ouder van een node is de vorige node in een pad, zodat het algoritme kan bijhouden hoe het pad verloopt. Dit script wordt dus gebruikt in andere scripts, om gegevens van de nodes op te vragen. Dit script wordt alleen gebruikt door andere scripts en er moet dus niets speciaal mee gebeuren.

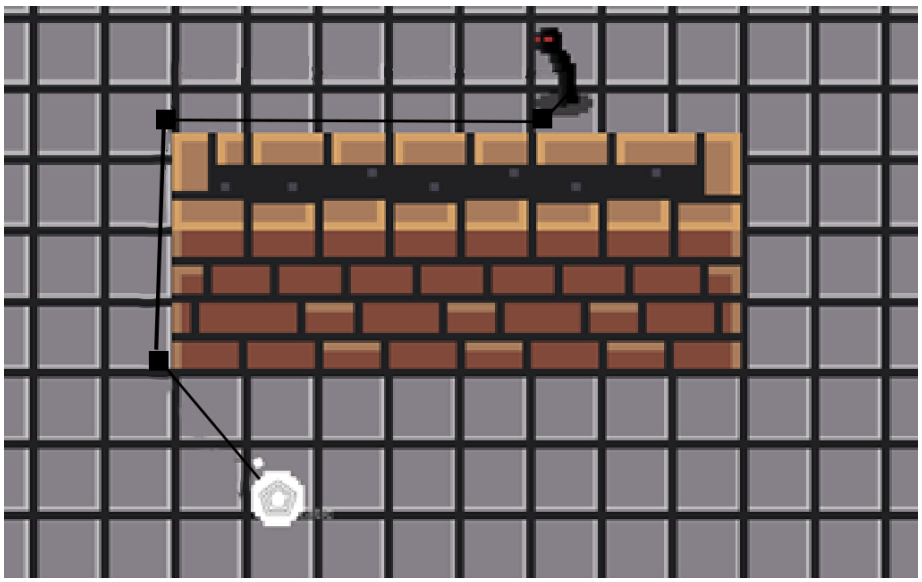
2.3.3 Pathfinding

Vervolgens zal ik het hebben over het script waar de pathfinding zelf wordt gedaan (zie bijlage 3). In dit script zit het algoritme dat ik eerder al heb uitgelegd met de pseudocode. Hier zit ook een functie in die het pad terug natrekt wanneer er een node in de OPEN lijst zit die overeenkomt met de eindbestemming. Op figuur 16 kan je zien dat het pad gevormd is.



Figuur 16: Pad

In het pathfinding script wordt het pad ook simpeler gemaakt, het heeft namelijk geen zin om elke node in een rechte lijn op het pad bij te houden. Dus worden er waypoints gezet waar het pad van richting veranderd en wordt de vijand naar die waypoint gestuurd. Het simpeler pad ziet er dan zo uit als de zwarte lijn op figuur 17.



Figuur 17: Verbeterd pad

Dit script moet op hetzelfde object geplaatst worden als het rooster, want het startpunt en het eindpunt van het pad worden berekend ten opzichte van de positie van dit object. Als dit object niet in het midden staat, gaan de paden niet correct zijn.

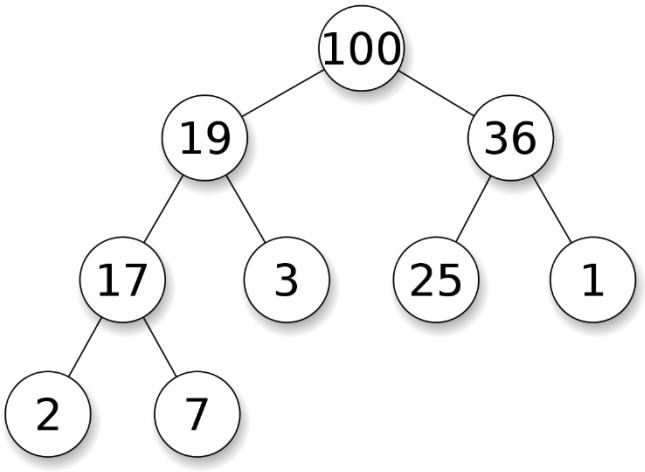
2.3.4 Units

Er is ook een script dat op de vijanden wordt geplaatst, dit script zorgt ervoor dat een vijand een pad aanvraagt en dit pad ook volgt (zie bijlage 4). Het gaat ook checken of de speler nog niet verplaatst is en als dit wel zo is, vraagt het dit script een nieuw pad

aan. Bij dit script moet ook gedefinieerd worden wat het doelwit is, hier heb ik dus de speler gekozen als doelwit.

2.3.5 Heap

Ten laatste is er ook nog een script om het algoritme te optimaliseren, dit doe ik met een heap (zie bijlage 5). Een heap is een datastructuur in de vorm van een boomdiagram zoals te zien op figuur 18. Een heap is handig als er meerdere keren de belangrijkste of minst belangrijke node verwijderd moet worden. In een heap wordt een



Figuur 18: Heap (Heap (data sctructure), 2021)

node alleen vergeleken met zijn ouders en grootouders en niet met broers, zussen of neven en nichten. De heap die ik gebruik hier is een binary heap hier kan elke node tot 2 kindnodes[3] hebben. Er zijn twee soorten heaps: een Maximum en een minimum heap, bij de maximum heap is elke oudernode[4] groter dan de kindnode. Bij een minimum heap exact het omgekeerde. Ik ga een minimum heap gebruiken, want bij pathfinding moet de oudernode kleiner zijn dan zijn kindnodes. Als er een nieuwe node inkomt wordt die vergeleken met de parent node is die kleiner, dan wisselen deze twee nodes. Dit wordt herhaald totdat alle oudernodes kleiner zijn dan de kindnodes. Op deze manier wordt er alleen vergeleken met oudernodes en is het dus efficiënter (Heap (data structure), 2021).

[3] Child nodes van een node zijn de nodes na die node.
[4] Parent node is de node die voor een node komt.

2.3.6 Problemen

Eerst had ik een pathfinding algoritme genomen dat bedoeld was voor een 3D game, maar ik kon dit zo aanpassen dat dit ook werkte in een 2D game. Bij de 3D game werden de x-as en de z-as gebruikt voor de oriëntatie van het rooster. Omdat de grond gelijk is aan de z-as. Maar bij mijn game is de grond gelijk aan de y-as dus veranderde ik dit zo in de scripts. Ook bij het definiëren moest ik iets aanpassen, het algoritme herkende namelijk niet mijn afbeeldingen als objecten en moest ik dus onzichtbare kubussen maken waar een obstakel was. Toen ik op zoek was naar een betere oplossing hiervoor, vond ik dat de maker van het algoritme een versie voor 2D games had gemaakt. Ik heb dan deze scripts toegepast en de problemen waren er niet meer. Er was nog een laatste probleem, de vijanden gingen naar de positie waar de speler was bij het begin van het spel. Dit kwam omdat het aanvragen van het pad in een functie stond die alleen werd geactiveerd bij het starten van de game. Ik heb dit opgelost door het aanvragen van het pad in een functie te plaatsen die elke keer dat er een update is in de game geactiveerd wordt. Wat ik bedoel met updates is elke keer dat het beeld vernieuwt. Dit is gelijk aan de frames per seconde, dit is de hoeveelheid afbeeldingen er per seconden worden afgebeeld. Dus ook als er niets is veranderd wordt de update functie nog steeds geactiveerd. Er kan alleen iets gebeuren op een update maar dit moet niet.

Ik heb ook zelf aan verbeteringen gedacht die ik in de toekomst zou kunnen toepassen. Bijvoorbeeld ervoor zorgen dat de vijanden plaatsen met licht vermijden maar er wel door kunnen. Een systeem waardoor de vijanden niet allemaal uit één richting naar de speler komen. Hierbij zou ik de vijanden met elkaar moeten laten communiceren.

Besluit

Pathfinding is een vorm van reactieve AI waarbij de computer het beste pad vindt tussen twee punten. Dit kan gedaan worden op verschillende manieren en kan op verschillende manieren toegepast worden. Van een stofzuiger tot de vijanden in een videospel. Bij de meeste algoritmes wordt dit gedaan door nodes te gebruiken en de afstand te berekenen. Zoals bij Dijkstra's algoritme en het A*algoritme, hier is het eerste nuttig om meerdere paden tegelijk te vinden, zoals alle beste wegen naar alle omringende steden vinden. A* wordt eerder gebruikt om één specifiek pad te vinden tussen twee punten. Op deze algoritmes kunnen ook verbeteringen geplaatst worden zodat het pad natuurlijker wordt of strategischer. Hierbij is A* een verbetering van Dijkstra's algoritme om het efficiënter te maken.

Pathfinding wordt toegepast op videogames door eerst een pseudocode te schrijven en dan aan de hand van deze pseudocode het algoritme te schrijven. Als je het algoritme niet zelf wilt maken, kan je het algoritme van iemand anders gebruiken, die dit vrijwillig ter beschikking stelt. Hierbij is het handig om uit te zoeken hoe deze algoritmes in elkaar zitten zodat je zelf nog verbeteringen kunt invoeren. Het algoritme wordt bepaald door de game engine waarin het spel is gemaakt en in welke taal het spel is geschreven. In dit eindwerk was de game engine Unity en de taal C#. Als je weet hoe je een game engine moet gebruiken is het niet moeilijk om pathfinding toe te passen, want er zijn al genoeg algoritmes online die je kan gebruiken. De algoritmes volledig begrijpen is een grotere uitdaging, zeker als je niet alle kennis hebt over de taal van het algoritme. Dit komt omdat de makers van algoritmes de code zo compact mogelijk proberen te maken. Deze compacte vorm is moeilijker om te begrijpen.

Er kan nog verder onderzocht worden op welke manier de verschillende eenheden met elkaar kunnen communiceren en dus rekening houden met de andere paden. Voor mijzelf wil ik hier later op terugkomen en zelf mijn eigen algoritme schrijven als ik meer kennis heb van programmeren.

Je kunt mijn computerspel met pathfinding uittesten op schellemans.be.

Literatuurlijst

Bauvois, V. (2017, 5 oktober). *Artificiële intelligentie: de 4 types*. Geraadpleegd op 18 oktober 2020, van https://www.meemetdenieuwemaak.be/2017/10/05/artificiele-intelligentie-de-4-types/

Edwards, J.S (2018, 6 juni). A Comparison of Path Planning Algorithms for Robotic Vacuum Cleaners. Geraadpleegd op 5 januari 2021, van https://kth.diva-portal.org/smash/get/diva2:1214422/FULLTEXT01.pdf

John, T. C. H., Prakash, E. C., & Chaudhari, N. S. (2008). Strategic Team AI Path Plans: Probabilistic Pathfinding. *International Journal of Computer Games Technology*, *2008*, 1–6. https://doi.org/10.1155/2008/834616

Kim, Y. M. I. (2019, 13 juni). Dijkstra Algorithm: Key to Finding the Shortest Path, Google Map to Waze. Geraadpleegd op 9 November 2020, van https://medium.com/@yk392/dijkstra-algorithm-key-to-finding-the-shortest-path-google-map-to-waze-56ff3d9f92f0

Knapton, S. (2017, 6 december). *Entire human chess knowledge learned and surpassed by DeepMind's AlphaZero in four hours*. Geraadpleegd op 18 oktober 2020, van https://www.telegraph.co.uk/science/2017/12/06/entire-human-chess-knowledge-learned-surpassed-deepminds-alphazero/

KU Leuven, & Puttemans, S. (2017). *Exploiting scene constraints to improve object detection algorithms for industrial applications*. KU Leuven – Faculty of Engineering Technology. Geraadpleegd van https://limo.libis.be/primo-explore/fulldisplay?docid=32LIBIS_ALMA_DS71202344110001471&=&=&=&=&=

&=&=&=&=&=&context=L&vid=KULeuven&lang=en_US&search_scope=ALL_CONTENT&adaptor=Local%20Search%20Engine&tab=all_content_tab&query=any,contains,Steven%20puttemans&offset=0

Sebastian Lague. (2014, December 16). *A* Pathfinding (E01: algorithm explanation)* [Video file]. Geraadpleegd op 5 januari 2021 van https://www.youtube.com/watch?v=-L-WgKMFuhE&t=1s

Sebastian Lague. (2015, 11 januari). A* Pathfinding (E06: weights) [Videobestand]. Geraadpleegd op 5 januari 2021 van https://www.youtube.com/watch?v=T0Qv4-KkAUo&t=2s

Wikipedia-bijdragers. (2020, 11 september). *Kunstmatige intelligentie*. Geraadpleegd op 18 oktober 2020, van https://nl.wikipedia.org/wiki/Kunstmatige_intelligentie

Wikipedia-bijdragers. (2020, 22 december). Heap (data structure). Geraadpleegd op 1 januari 2021, van https://en.wikipedia.org/wiki/Heap_(data_structure)

Wikipedia-bijdragers. (2021, 2 januari). Node (computer science). Geraadpleegd op 5 januari 2021, van *Wikipedia*. https://en.wikipedia.org/wiki/Node_(computer_science)

Pathfinding scripts geraadpleegd van:

Sebastian Lague (2017, 8 december) Pathfinding-2D. Geraadpleegd op 5 januari 2021, van https://github.com/SebLague/Pathfinding-2D

Pseudocode geraadpleegd van:

Sebastian Lague (2017, 8 december) Pathfinding. Geraadpleegd op 5 januari 2021, van https://github.com/SebLague/Pathfinding

Afbeelding

Wikipedia-bijdragers. (2020, 22 december). Heap (data structure). Geraadpleegd op 12 januari 2021, van https://en.wikipedia.org/wiki/Heap_(data_structure)

Bijlage 1

Grid.cs:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Grid : MonoBehaviour {

    public bool displayGridGizmos;

    public LayerMask unwalkableMask;
    public Vector2 gridWorldSize;
    public float nodeRadius;

    Node[,] grid;
    float nodeDiameter;
    int gridSizeX, gridSizeY;

    void Awake() {
        nodeDiameter = nodeRadius*2;
        gridSizeX = Mathf.RoundToInt(gridWorldSize.x/nodeDiameter);
        gridSizeY = Mathf.RoundToInt(gridWorldSize.y/nodeDiameter);

        CreateGrid();
    }

    public int MaxSize {
        get {
            return gridSizeX * gridSizeY;
        }
    }

    void CreateGrid() {
        grid = new Node[gridSizeX,gridSizeY];
        Vector2 worldBottomLeft = (Vector2)transform.position - Vector2.right * gridWorldSize.x/2 -
 Vector2.up * gridWorldSize.y/2;

        for (int x = 0; x < gridSizeX; x ++) {
            for (int y = 0; y < gridSizeY; y ++) {
                Vector2 worldPoint = worldBottomLeft + Vector2.right * (x * nodeDiameter + nodeRadi
us) + Vector2.up * (y * nodeDiameter + nodeRadius);
                bool walkable = (Physics2D.OverlapCircle(worldPoint,nodeRadius,unwalkableMask) == n
ull); // if no collider2D is returned by overlap circle, then this node is walkable
```

```
            grid[x,y] = new Node(walkable,worldPoint, x,y);
        }
    }
}


public List<Node> GetNeighbours(Node node, int depth = 1) {
    List<Node> neighbours = new List<Node>();

    for (int x = -depth; x <= depth; x++) {
        for (int y = -depth; y <= depth; y++) {
            if (x == 0 && y == 0)
                continue;

            int checkX = node.gridX + x;
            int checkY = node.gridY + y;

            if (checkX >= 0 && checkX < gridSizeX && checkY >= 0 && checkY < gridSizeY) {
                neighbours.Add(grid[checkX,checkY]);
            }
        }
    }

    return neighbours;
}


public Node NodeFromWorldPoint(Vector2 worldPosition) {
    float percentX = (worldPosition.x + gridWorldSize.x/2) / gridWorldSize.x;
    float percentY = (worldPosition.y + gridWorldSize.y/2) / gridWorldSize.y;
    percentX = Mathf.Clamp01(percentX);
    percentY = Mathf.Clamp01(percentY);

    int x = Mathf.RoundToInt((gridSizeX-1) * percentX);
    int y = Mathf.RoundToInt((gridSizeY-1) * percentY);
    return grid[x,y];
}

public Node ClosestWalkableNode(Node node) {
    int maxRadius = Mathf.Max (gridSizeX, gridSizeY) / 2;
    for (int i = 1; i < maxRadius; i++) {
        Node n = FindWalkableInRadius (node.gridX, node.gridY, i);
        if (n != null) {
            return n;

        }
```

```
    }
    return null;
}
Node FindWalkableInRadius(int centreX, int centreY, int radius) {

    for (int i = -radius; i <= radius; i ++) {
        int verticalSearchX = i + centreX;
        int horizontalSearchY = i + centreY;

        // top
        if (InBounds(verticalSearchX, centreY + radius)) {
            if (grid[verticalSearchX, centreY + radius].walkable) {
                return grid [verticalSearchX, centreY + radius];
            }
        }

        // bottom
        if (InBounds(verticalSearchX, centreY - radius)) {
            if (grid[verticalSearchX, centreY - radius].walkable) {
                return grid [verticalSearchX, centreY - radius];
            }
        }
        // right
        if (InBounds(centreY + radius, horizontalSearchY)) {
            if (grid[centreX + radius, horizontalSearchY].walkable) {
                return grid [centreX + radius, horizontalSearchY];
            }
        }

        // left
        if (InBounds(centreY - radius, horizontalSearchY)) {
            if (grid[centreX - radius, horizontalSearchY].walkable) {
                return grid [centreX - radius, horizontalSearchY];
            }
        }

    }

    return null;

}

bool InBounds(int x, int y) {
    return x>=0 && x<gridSizeX && y>= 0 && y<gridSizeY;
}
```

```
    void OnDrawGizmos() {
        Gizmos.DrawWireCube(transform.position,new Vector2(gridWorldSize.x,gridWorldSize.y));
        if (grid != null && displayGridGizmos) {
            foreach (Node n in grid) {
                Gizmos.color = Color.red;
                if (n.walkable)
                    Gizmos.color = Color.white;

                Gizmos.DrawCube(n.worldPosition, Vector3.one * (nodeDiameter-.1f));
            }
        }
    }

}
```

Bijlage 2

Node.cs:

```
using UnityEngine;
using System.Collections;

public class Node : IHeapItem<Node> {

    public bool walkable;
    public Vector2 worldPosition;
    public int gridX;
    public int gridY;

    public int gCost;
    public int hCost;
    public Node parent;
    int heapIndex;

    public Node(bool _walkable, Vector2 _worldPos, int _gridX, int _gridY) {
        walkable = _walkable;
        worldPosition = _worldPos;
        gridX = _gridX;
        gridY = _gridY;
    }

    public int fCost {
        get {
            return gCost + hCost;
        }
```

```
    }

    public int HeapIndex {
        get {
            return heapIndex;
        }
        set {
            heapIndex = value;
        }
    }

    public int CompareTo(Node nodeToCompare) {
        int compare = fCost.CompareTo(nodeToCompare.fCost);
        if (compare == 0) {
            compare = hCost.CompareTo(nodeToCompare.hCost);
        }
        return -compare;
    }
}
```

Bijlage 3

Pathfinding.cs:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System;

public class Pathfinding : MonoBehaviour {

    Grid grid;
    static Pathfinding instance;

    void Awake() {
        grid = GetComponent<Grid>();
        instance = this;
    }

    public static Vector2[] RequestPath(Vector2 from, Vector2 to) {
        return instance.FindPath (from, to);
    }
```

```
Vector2[] FindPath(Vector2 from, Vector2 to) {

    Stopwatch sw = new Stopwatch();
    sw.Start();

    Vector2[] waypoints = new Vector2[0];
    bool pathSuccess = false;

    Node startNode = grid.NodeFromWorldPoint(from);
    Node targetNode = grid.NodeFromWorldPoint(to);
    startNode.parent = startNode;

    if (!startNode.walkable) {
        startNode = grid.ClosestWalkableNode (startNode);
    }
    if (!targetNode.walkable) {
        targetNode = grid.ClosestWalkableNode (targetNode);
    }

    if (startNode.walkable && targetNode.walkable) {

        Heap<Node> openSet = new Heap<Node>(grid.MaxSize);
        HashSet<Node> closedSet = new HashSet<Node>();
        openSet.Add(startNode);

        while (openSet.Count > 0) {
            Node currentNode = openSet.RemoveFirst();
            closedSet.Add(currentNode);

            if (currentNode == targetNode) {
                sw.Stop();
                print ("Path found: " + sw.ElapsedMilliseconds + " ms");
                pathSuccess = true;
                break;
            }

            foreach (Node neighbour in grid.GetNeighbours(currentNode)) {
                if (!neighbour.walkable || closedSet.Contains(neighbour)) {
                    continue;
                }

                int newMovementCostToNeighbour = currentNode.gCost + GetDistance(currentNode,
neighbour)+TurningCost(currentNode,neighbour);
                if (newMovementCostToNeighbour < neighbour.gCost || !openSet.Contains(neighbou
r)) {
```

```
                neighbour.gCost = newMovementCostToNeighbour;
                neighbour.hCost = GetDistance(neighbour, targetNode);
                neighbour.parent = currentNode;

                if (!openSet.Contains(neighbour))
                    openSet.Add(neighbour);
                else
                    openSet.UpdateItem(neighbour);
            }
        }
    }

    if (pathSuccess) {
        waypoints = RetracePath(startNode,targetNode);
    }

    return waypoints;

}


int TurningCost(Node from, Node to) {
    /*
    Vector2 dirOld = new Vector2(from.gridX - from.parent.gridX, from.gridY - from.parent.gridY
);
    Vector2 dirNew = new Vector2(to.gridX - from.gridX, to.gridY - from.gridY);
    if (dirNew == dirOld)
        return 0;
    else if (dirOld.x != 0 && dirOld.y != 0 && dirNew.x != 0 && dirNew.y != 0) {
        return 5;
    }
    else {
        return 10;
    }
    */

    return 0;
}

Vector2[] RetracePath(Node startNode, Node endNode) {
    List<Node> path = new List<Node>();
    Node currentNode = endNode;

    while (currentNode != startNode) {
        path.Add(currentNode);
```

```
            currentNode = currentNode.parent;
        }
        Vector2[] waypoints = SimplifyPath(path);
        Array.Reverse(waypoints);
        return waypoints;

    }

    Vector2[] SimplifyPath(List<Node> path) {
        List<Vector2> waypoints = new List<Vector2>();
        Vector2 directionOld = Vector2.zero;

        for (int i = 1; i < path.Count; i ++) {
            Vector2 directionNew = new Vector2(path[i-1].gridX - path[i].gridX,path[i-
1].gridY - path[i].gridY);
            if (directionNew != directionOld) {
                waypoints.Add(path[i].worldPosition);
            }
            directionOld = directionNew;
        }
        return waypoints.ToArray();
    }

    int GetDistance(Node nodeA, Node nodeB) {
        int dstX = Mathf.Abs(nodeA.gridX - nodeB.gridX);
        int dstY = Mathf.Abs(nodeA.gridY - nodeB.gridY);

        if (dstX > dstY)
            return 14*dstY + 10* (dstX-dstY);
        return 14*dstX + 10 * (dstY-dstX);
    }


}
```

Bijlage 4

Unit.cs:

```
using UnityEngine;
using System.Collections;

public class Unit : MonoBehaviour {

    public Transform target;
```

```
    public float speed = 20;

    Vector2[] path;
    int targetIndex;

    void Start() {
        StartCoroutine (RefreshPath ());
    }

    IEnumerator RefreshPath() {
        Vector2 targetPositionOld = (Vector2)target.position + Vector2.up; // ensure != to target.pos
ition initially

        while (true) {
            if (targetPositionOld != (Vector2)target.position) {
                targetPositionOld = (Vector2)target.position;

                path = Pathfinding.RequestPath (transform.position, target.position);
                StopCoroutine ("FollowPath");
                StartCoroutine ("FollowPath");
            }

            yield return new WaitForSeconds (.25f);
        }
    }

    IEnumerator FollowPath() {
        if (path.Length > 0) {
            targetIndex = 0;
            Vector2 currentWaypoint = path [0];

            while (true) {
                if ((Vector2)transform.position == currentWaypoint) {
                    targetIndex++;
                    if (targetIndex >= path.Length) {
                        yield break;
                    }
                    currentWaypoint = path [targetIndex];
                }

                transform.position = Vector2.MoveTowards (transform.position, currentWaypoint, spee
d * Time.deltaTime);
                yield return null;

            }
        }
```

```
    }

    public void OnDrawGizmos() {
        if (path != null) {
            for (int i = targetIndex; i < path.Length; i ++) {
                Gizmos.color = Color.black;
                //Gizmos.DrawCube((Vector3)path[i], Vector3.one *.5f);

                if (i == targetIndex) {
                    Gizmos.DrawLine(transform.position, path[i]);
                }
                else {
                    Gizmos.DrawLine(path[i-1],path[i]);
                }
            }
        }
    }
}
```

Bijlage 5

Heap.cs:

```
using UnityEngine;
using System.Collections;
using System;

public class Heap<T> where T : IHeapItem<T> {

    T[] items;
    int currentItemCount;

    public Heap(int maxHeapSize) {
        items = new T[maxHeapSize];
    }

    public void Add(T item) {
        item.HeapIndex = currentItemCount;
        items[currentItemCount] = item;
        SortUp(item);
        currentItemCount++;
    }

    public T RemoveFirst() {
        T firstItem = items[0];
```

```
    currentItemCount--;
    items[0] = items[currentItemCount];
    items[0].HeapIndex = 0;
    SortDown(items[0]);
    return firstItem;
}

public void UpdateItem(T item) {
    SortUp(item);
}

public int Count {
    get {
        return currentItemCount;
    }
}

public bool Contains(T item) {
    return Equals(items[item.HeapIndex], item);
}

void SortDown(T item) {
    while (true) {
        int childIndexLeft = item.HeapIndex * 2 + 1;
        int childIndexRight = item.HeapIndex * 2 + 2;
        int swapIndex = 0;

        if (childIndexLeft < currentItemCount) {
            swapIndex = childIndexLeft;

            if (childIndexRight < currentItemCount) {
                if (items[childIndexLeft].CompareTo(items[childIndexRight]) < 0) {
                    swapIndex = childIndexRight;
                }
            }

            if (item.CompareTo(items[swapIndex]) < 0) {
                Swap (item,items[swapIndex]);
            }
            else {
                return;
            }

        }
        else {
            return;
```

```
        }

      }
    }

    void SortUp(T item) {
      int parentIndex = (item.HeapIndex-1)/2;

      while (true) {
        T parentItem = items[parentIndex];
        if (item.CompareTo(parentItem) > 0) {
          Swap (item,parentItem);
        }
        else {
          break;
        }

        parentIndex = (item.HeapIndex-1)/2;
      }
    }

    void Swap(T itemA, T itemB) {
      items[itemA.HeapIndex] = itemB;
      items[itemB.HeapIndex] = itemA;
      int itemAIndex = itemA.HeapIndex;
      itemA.HeapIndex = itemB.HeapIndex;
      itemB.HeapIndex = itemAIndex;
    }



}

public interface IHeapItem<T> : IComparable<T> {
    int HeapIndex {
      get;
      set;
    }
}
```

License:

```
MIT License
```