Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Autonomous agents for interactive virtual environments
Richting: master in de informatica - Human Computer Interaction Jaar: 2008

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

TILKIN, Servaas

Datum: 5.11.2008

Autonomous agents for interactive virtual environments

Servaas Tilkin

promotor :
Prof. dr. Wim LAMOTTE

Eindverhandeling voorgedragen tot het bekomen van de graad
master in de informatica Human Computer Interaction

Samenvatting

Deze thesis bestaat uit een studie van de technieken die gebruikt worden binnen de artificiële intelligentie. Hierna volgt de bespreking van een implementatie van een intelligente agent in een virtuele omgeving.

De besproken technieken worden ingeleid door een overzicht van de principes waarop deze gebaseerd zijn, gevolgd door een blik op de huidige stand van zaken. Het concept *pathfinding* wordt meer in detail uitgediept.

Tevens wordt er gekeken naar virtuele werelden die gevuld zijn met *intelligente objecten*. Intelligente objecten zijn objecten die zelf informatie bevatten over hoe ermee geïnterageerd kan worden. Het doel van deze thesis is onderzoeken hoe deze intelligente objecten kunnen bijdragen tot een vereenvoudiging van het ontwerp voor agenten.

De implementatie bestaat uit een lightweight pathfinding bibliotheek, gekoppeld aan een agent die zich kan handhaven in een wereld met intelligente objecten.

Woord vooraf

Bij het realiseren van deze thesis werd ik door een aantal mensen direct en indirect gesteund. Deze zou ik via deze weg willen bedanken.

Allereerst de mensen die me begeleid hebben tijdens het maken van deze thesis: mijn promotor, begeleiders, en diegenen die geholpen hebben door nazicht en feedback.

Daarnaast wil ik ook mijn ouders bedanken, dankzij hen heb ik de kans gekregen deze opleiding aan te vatten en af te ronden.

Tenslotte wil ik ook graag mijn vriendin, vrienden, mede-studenten en kennissen bedanken voor de morele steun tijdens het realiseren van mijn thesis.

Bedankt !

Inhoudsopgave

Lijst van figuren

Hoofdstuk 1

Inleiding

1.1 Virtuele werelden

Een virtuele wereld die enkel door gebruikers bewoond wordt, zou erg leeg zijn, en over het algemeen niet interessant en bruikbaar overkomen. Indien in het beste geval elke gebruiker zich rigoureus aan zijn vooropgestelde taak in de virtuele wereld houdt, dan zullen deze gebruikers en de services die ze elkaar aanbieden nog steeds niet permanent beschikbaar zijn, door het feit dat mensen niet dag in dag uit elk uur van de dag online kunnen zijn. Bovendien is het aanbieden van een permanente service binnen een virtuele omgeving meestal een oninteressant en repetitief gegeven dat weinig mensen zal aanspreken.

Een handig alternatief om dit soort functies uit te voeren is ze te laten uitvoeren door een *autonome agent*. Het gaat hier om een computerprogramma dat door zijn acties menselijk gedrag simuleert, en in staat is te interageren met echte menselijke gebruikers en de virtuele wereld zelf. Op deze manier kunnen de agenten ingezet worden om de virtuele omgeving voor gebruikers interessanter te maken. Een eventueel probleem van 'onderbevolking' op bepaalde uren van de dag kan op die manier ook opgelost worden door de gebruikerspopulatie aan te vullen met autonome agenten.

Er is ook de mogelijkheid om autonome agenten in een virtuele wereld in te zetten om bepaalde mogelijkheden ervan duidelijk te maken aan gebruikers. Wanneer de gebruiker iets niet kan oplossen kan hij dit probleem voorleggen aan de agent, en dan kan deze het probleem trachten op te lossen. Wanneer de gebruiker vervolgens ziet welke handelingen de agent uitvoert, kan hij deze als voorbeeld nemen. Het kan ook omgekeerd, in dat geval kan de agent leren door de gebruikers te observeren.

We zien ook hoe langer hoe meer dat de virtuele werelden complexer en dynamischer worden. Dit kan leiden tot situaties waarbij de klassieke aanpak tekort zal schieten: de hoeveelheid gegevens wordt te groot, de agenten worden te complex en daardoor moeilijk onderhoudbaar. Daarom is er vraag naar nieuwe inzichten en initiatieven binnen dit domein.

1.2 Artificiële intelligentie

Het domein van artificiële intelligentie is een interessant onderzoeksdomein. Het dwingt ons onder andere om inzichten te verwerven in de manier waarop denkprocessen en leerprocessen bij de mens en andere organismen zich voordoen. Al honderden jaren heeft de mensheid geprobeerd om autonoom denkende dingen te produceren, hetzij via godsdienst en spiritualiteit, alchemie, mechanica, chemie of elektriciteit. Sinds de 20ste eeuw is deze evolutie in een stroomversnelling gekomen dankzij de ontwikkelingen in de IT sector. De afgelopen halve eeuw fluctueerde de interesse voor AI nogal sterk. Periodes van veel interesse en ontwikkeling wisselden af met periodes van weinig interesse en weinig vooruitgang. AI heeft ook interessante praktische toepassingen in verschillende domeinen: robotica, games, intelligente services, enzovoort. Tegenwoordig wordt AI vaak achter de schermen gebruikt als werkpaard van bepaalde applicaties, games, en praktische toepassingen [20].

Over het potentieel van artificiële intelligentie wordt ook heftig gedebatteerd en gespeculeerd. Machines die autonoom opereren en missies kunnen uitvoeren, kunnen van onschatbare waarde blijken, onder andere bij ruimtemissies die langer duren dan een mensenleven. Ook intelligente robots die mensen kunnen bijstaan in allerlei taken spreken tot de verbeelding. Vele van deze onderwerpen zijn ook een inspiratie geweest voor enkele boeiende fictionele werken [1]. Rond het creëren van intelligent leven hangen ook enkele interessante ethische kwesties: is het amoreel om een nieuw soort leven te creëren ? Hoe zit het met de 'mensenrechten' van deze nieuwe denkende wezens ?

1.3 Doel van deze thesis

Wanneer een agent een bepaalde taak tot een goed einde moet brengen, zal hier planning aan te pas komen. Deze planning kan grote hoeveelheden basisinformatie nodig hebben, zoals ook de meeste mensen rondlopen met een veelheid aan 'basiskennis' die hen dagdagelijks helpt allerlei taken tot een goed einde te brengen. Het doel van deze thesis is onderzoeken hoe deze hoeveelheid noodzakelijke initiële kennis kan geminimaliseerd worden. Hierdoor zou het ontwerp van artificiële agenten aanzienlijk vereenvoudigd kunnen worden. Meer specifiek wordt er onderzocht hoe intelligente objecten hieraan kunnen bijdragen, en hoe dit principe kan verfijnd worden om dit doel te bereiken.

De tekst wordt ingeleid door een literatuurstudie die de fundamenten van artificiële intelligentie onderzoekt. Vervolgens wordt er een overzicht gegeven van recente ontwikkelingen in het domein. Hierbij hoort ook een *proof of concept* implementatie van een agent in een virtuele omgeving. Deze agent moet zijn weg kunnen vinden binnen deze virtuele wereld, bijvoorbeeld door gebruik te maken van pathfinding. Hij moet ook kunnen interageren met deze omgeving. Wanneer hij voor eenvoudige of complexe taken wordt gesteld zou hij een

[1] 'I, Robot' van Isaac Asimov, de 'Matrix' trilogie, de Terminator films, etc. [15]

werkend plan moeten kunnen opstellen om deze taken uit te voeren, gebruik makend van de intelligente objecten en de informatie die zij voorzien.

1.4 Indeling van de tekst

In het eerste deel van deze tekst bevindt zich de literatuurstudie, waarin de fundamenten en basisprincipes van het domein aangebracht worden. Er wordt dieper ingegaan op het probleemgebied van pathfinding.

Deze fundamenten worden in een tweede deel gevolgd door een selectie en bespreking van relevante hedendaagse technologieën.

Het derde en laatste deel licht de eigen ontwikkelingen toe. Er wordt kort een pathfinding bibliotheek besproken, gevolgd door een meer uitgebreid overzicht van een implementatie van een agent.

Dit geheel wordt afgerond met een conclusie en bibliografie.

Deel I

Fundamenten van Artificiële Intelligentie(AI)

Hoofdstuk 2

Intelligent Agents

Hier wordt kort besproken wat intelligente agenten zijn, gevolgd door enkele mogelijke basisontwerpen [RN95].

2.1 Algemeen

Agenten kunnen beschouwd worden als entiteiten die hun omgeving *observeren* via sensoren, en dan in die omgeving overgaan tot *handelen*. Een menselijke agent zou zijn zintuigen kunnen gebruiken om te observeren en zijn handen, voeten en andere lichaamsdelen om een invloed uit te oefenen op de wereld. Een software agent zal observaties ervaren uit een virtueel domein, en ook zijn acties daarin uitvoeren.

De dingen die de agent tot op een bepaald punt in de tijd heeft geobserveerd noemen we de *perceptie-sequentie*. Wanneer een agent zijn gedrag kan bepalen door zijn eigen ervaringen, zeggen we dat de agent *autonoom gedrag* vertoont.

2.2 Lookup table

De eenvoudigste aanpak om een agent te implementeren is een tabel bijhouden met alle mogelijke perceptie-sequenties en de daarbij horende acties. Alhoewel deze agent zich correct zou gedragen, kampt deze aanpak met enkele belangrijke praktische problemen.

Een eerste probleem is de hoeveelheid gegevens. Aangezien we voor elke perceptie-sequentie de actie moeten bijhouden, kan dit zelfs voor beperkte probleemdomeinen al problematische proporties aannemen. Een schaak-agent zou bijvoorbeeld al een tabel nodig hebben die 35^{100} lijnen bevat. Het is bovendien quasi onhaalbaar om deze tabel te gaan opvullen, hetzij manueel, hetzij door de agent alle sequenties te laten ervaren.

Het tweede probleem is dat de agent geen autonomie vertoont, aangezien de acties volledig bepaald worden door vooraf bepaalde redeneringen. Wanneer er dus een verandering zou optreden in de omgeving, zou de agent hier geen rekening mee kunnen houden en niet langer correct functioneren.

2.3 Reflex agents

Een manier om te ontkomen aan het feit dat het onhaalbaar is een tabel op te stellen voor alle mogelijke perceptie-sequenties, is de tabel te verkleinen door bepaalde delen samen te vatten. We kunnen een aantal overeenkomsten zoeken tussen groepen *input/output* sequenties, en deze dan in een soort *als-dan-regel* gieten. De agent zal dan *at runtime* zijn perceptie analyseren om te kijken of deze met één van de gevonden regels overeenkomt. Als dit zo is, wordt de overeenkomstige actie in gang gezet.

Deze methode van perceptie die direct een actie tot gevolg heeft kan vergeleken worden met reflexmatig menselijk gedrag. Enkele gedragingen hiervan zijn aangeleerd, zoals de reflex om te remmen in de auto in een gevaarlijke situatie. Andere zijn aangeboren automatismen, zoals de ogen sluiten wanneer een object nadert.

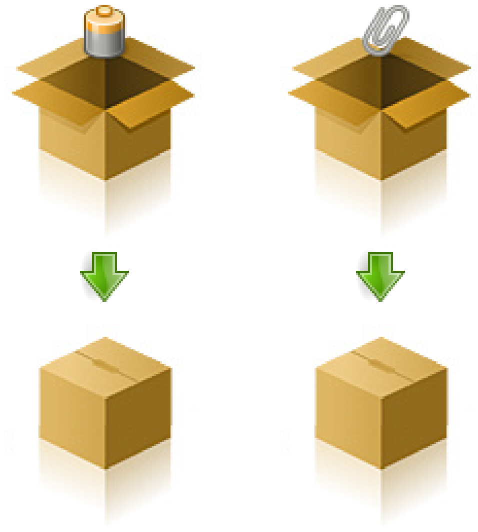
2.4 Agents die een status bijhouden

Soms is het voor een agent niet voldoende om enkel de huidige perceptie in rekening te brengen. Dit probleem ontstaat door het feit dat de zintuigen van de agent meestal niet in staat zijn de volledige wereld te observeren. In zulke gevallen moet de agent een soort interne representatie van de wereld bijhouden, om situaties die qua perceptie niet verschillen toch te kunnen onderscheiden. Deze functie kan vergeleken worden met de functie van het menselijke geheugen.

Een voorbeeld hiervan is te zien in figuur 2.1. Als we enkel de laatste perceptie bekijken (een gesloten doos[1]), dan is er voor de agent geen verschil. Als we echter in de interne representatie ook de vorige percepties bekijken[2], waar we zien welk object in de doos geplaatst wordt, is het onderscheid duidelijk.

Deze interne status up to date houden is niet altijd eenvoudig. Door het feit dat de agent zich in een dynamische omgeving bevindt waar meerdere gebruikers aan kunnen deelnemen, kan het zijn dat bepaalde aspecten van de wereld veranderd zijn zonder dat de agent dit rechtstreeks heeft kunnen observeren. Er is informatie nodig die de agent vertelt hoe de wereld evolueert onafhankelijk van de agent zelf, zoals het berekenen van het toekomstige pad van bewegende objecten. Daarnaast is er ook informatie nodig die de agent vertelt hoe zijn acties de wereld beïnvloeden.

[1] `http://www.istockphoto.com/` (afbeelding doos)
[2] `http://tango.freedesktop.org/` (overige icons)



(a) Een doos met een batterij

(b) Een doos met een paperclip

Figuur 2.1: Twee dozen die op basis van de laatste perceptie niet te onderscheiden zijn.

2.5 Goal-based agents

Een goed beeld hebben van de hele wereld op basis van de interne status is zelden voldoende om een correcte beslissing te nemen. Bijna altijd hangt een beslissing ook af van het einddoel dat de agent wil bereiken. Er is dus nood aan een soort beschrijving van een *gewenste situatie* die hij moet realiseren.

Het gewenste doel kan gecombineerd worden met de informatie die de agent heeft over het effect van zijn acties op de status van de wereld (zie 2.4). Zo kan hij acties uitkiezen die het doel realiseren. *Zoekalgoritmes* en *planning* zijn de velden binnen AI die zich bezighouden met het vinden van sequenties van acties die een doel realiseren (zie 4.3).

2.6 Utility-based agents

Doelen voorzien de agent van een onderscheid tussen gewenste en ongewenste situaties om in terecht te komen. Dit garandeert echter niet dat de agent een logische, betrouwbare of efficiënte weg hier naartoe volgt. Om een fijner onderscheid te kunnen maken tussen hoe nuttig bepaalde acties voor een agent zijn, kunnen we voor elke status een *nutsgehalte* berekenen.

Dit nutsgehalte kan op meerdere manieren gebruikt worden om de juiste beslissing te nemen in diverse situaties. Wanneer er twee conflicterende doelen zijn (bijvoorbeeld snelheid en veiligheid), kan hier de juiste afweging gemaakt worden. Wanneer er meerdere doelen zijn, waarvan geen enkele met zekerheid kan bereikt worden, kan een agent het nutsgehalte en de slaagkans tegenover elkaar afwegen en op basis daarvan een goede keuze maken.

Hoofdstuk 3

Technologiën die aan de oorsprong liggen van AI

In dit hoofdstuk worden enkele technologieën, theoriën en ontwikkelingen besproken die de fundamenten hebben gelegd voor de ontwikkeling van artificiële intelligentie. Achtereenvolgens komen eerste orde logica, grafen en bomen, neurale netwerken en robotica aan bod.

3.1 Eerste orde logica [RN95]

Eerste orde logica kan gebruikt worden om de kennis van een artificiële agent voor te stellen. Het stelt de wereld voor als een verzameling objecten die aan bepaalde eigenschappen voldoen. Tussen deze objecten bestaan relaties en functies. Een functie is een speciale soort relatie, waarbij om het even welk object een relatie heeft met exact één ander object.

Eerste orde logica heeft een grote rol gespeeld in wiskunde, filosofie, en AI omdat het eenvoudig is de wereld voor te stellen als bestaande uit objecten en relaties, en op die manier redeneringen op te bouwen. Het grote verschil met *propositielogica* is dat deze als enige veronderstelling maakt dat de wereld uit feiten bestaat. Dit maakt propositielogica te beperkt om zelfs eenvoudige werelden efficiënt voor te stellen. Eerste orde logica heeft bovenop feiten ook nog objecten en relaties, maar maakt verder geen veronderstellingen over categorieën, tijd, en gebeurtenissen. In tegenstelling tot eerste orde logica zijn de domeinen die hier wel rekening mee houden nog niet volledig doorgrond, waardoor ze minder in aanmerking komen om gebruikt te worden.

Enkele voorbeelden:

- *Objecten*: huis, boom, straat, mens, ...
- *Relaties*: groter dan, is verwant met, ...

- *Eigenschappen*: groen, groot, snel, ...
- *Functies*: vader van, één meer dan, ...

We onderscheiden bovendien ook nog *constante symbolen* die naar specifieke objecten verwijzen (A, B, Peter, ...).

Predicaten worden gebruikt om bepaalde relaties aan te duiden. Intern kunnen we deze voorstellen als lijsten met veelvouden van waarden. De lijst { (2,4); (3,9); (4,16); ... } zou bijvoorbeeld een deel kunnen zijn van een 'kwadraat' functie.

Een *term* is een logische expressie die naar een object refereert. Dit kan handiger zijn dan elk object rechtstreeks een naam te geven. Een term als *LeftBranch(Tree1)* kan dus gebruikt worden in plaats van de tak zelf een naam te geven.

We kunnen *logische operatoren* zoals $\wedge$ (en), $\vee$ (of), $\neg$ (niet), $\Rightarrow$ (als-dan) gebruiken om complexe zinnen te maken op basis van atomaire zinnen. Wanneer we willen aanduiden dat twee zinnen hetzelfde object beschrijven, gebruiken we de gelijkheidsoperator $=$ (is gelijk aan). Om iets meer te zeggen over groepen objecten en het bestaan van bepaalde soorten objecten kunnen we de universele operator $\forall$ (voor alle) en de existentiële operator $\exists$ (er bestaat) gebruiken.

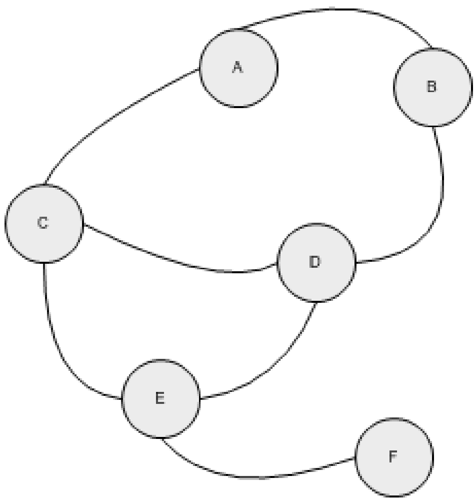
De basis afleidingsregels uit propositielogica gelden ook in eerste orde logica: *Modus Ponens*, *EN-eliminatie*, *EN-introductie*, *OF-introductie* en *oplossing* (zie bijlage A). Om met de universele en existentiële operatoren te kunnen werken komen daar nog *universele eliminatie*, *existentiële eliminatie* en *existentiële introductie* bij.

3.2 Grafen en bomen

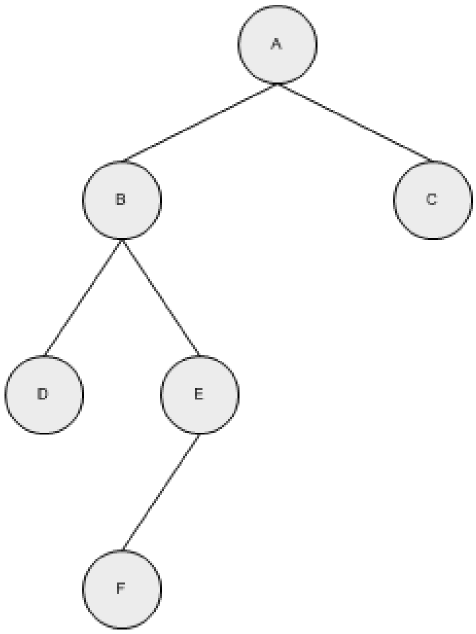
Een *graaf* bestaat uit knopen die met links verbonden zijn (zie figuur 3.1(a)). In de meeste gevallen is een link tussen een paar knopen bidirectioneel, zodat een verbinding van knoop A naar knoop B ook een verbinding in de omgekeerde richting impliceert. In het andere geval spreken we van een *gerichte graaf*, waarbij elke link een richting krijgt.

Een *boom* is een graaf waarin geen cycli voorkomen. Een alternatieve definitie is dat een boom een graaf is waarbij elke set knopen verbonden is met exact één pad. Meestal wordt in een boom één knoop als de wortelknoop gekozen; de knopen op de niveaus daaronder worden *kinderen* (en kleinkinderen) van de wortelknoop genoemd (zie figuur 3.1(b)).

Deze notaties kunnen binnen AI gebruikt worden om een probleem of een probleemruimte op een efficiënte manier voor te stellen. Het voordeel van deze voorstellingen is dat deze al veel bestudeerd zijn, en dat er efficiënte methodes en algoritmes beschreven zijn om hierin te navigeren. Als we dus ons probleemdomein kunnen voorstellen als een graaf, kunnen we deze algoritmen gebruiken om daarin een oplossing te zoeken (zie 4.3).

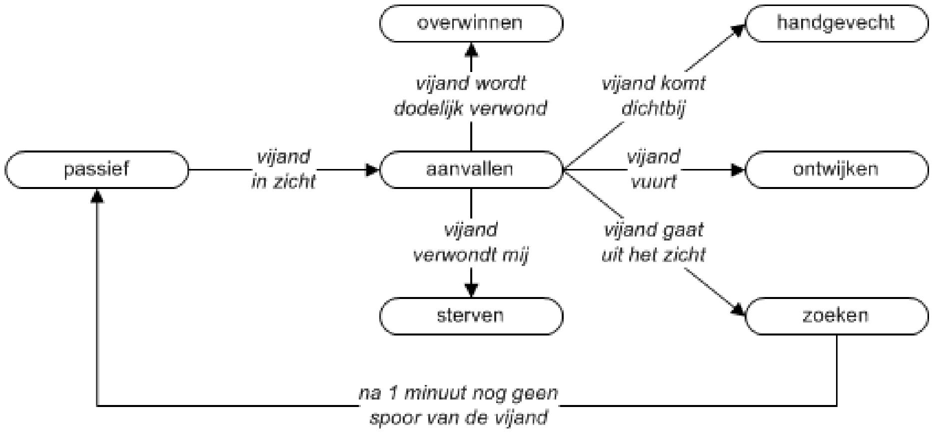


(a) Een graaf met 6 knopen en 7 bogen.



(b) Een boom met 6 knopen waarvan de wortelknoop A is.

Figuur 3.1: Grafen en Bomen



Figuur 3.2: Een finite state machine [PD05]

3.2.1 Finite state machines

Een speciaal soort graaf die vaak in AI gebruikt wordt is een *finite state machine*. Deze geeft op de knopen verschillende staten weer waarin *een agent* of object zich kan bevinden. Staten tussen dewelke er overgangen mogelijk zijn, worden met links verbonden. Deze links stellen dan acties of gebeurtenissen voor die ervoor zorgen dat de agent naar een andere status zal overgaan (zie figuur 3.2).

3.2.2 State space diagram

Wanneer we een probleemdomein willen voorstellen, kunnen we een opsomming maken van alle staten van de wereld die mogelijk zijn binnen dat probleemdomein. We vertrekken vanuit de initiële status, dan noteren we alle bereikbare staten uit dat punt, dan alle bereikbare staten uit elk van die staten, enzovoort. Als we elke staat verbinden met de staten die van daaruit bereikbaar zijn, krijgen we een graaf. Deze graaf is dan een voorstelling van de *state space* van het probleem (zie figuur 4.1).

3.3 Neurale netwerken [RN95]

Een *artificieel neuraal netwerk* is een netwerk dat bestaat uit eenvoudige rekenkundige eenheden die met elkaar verbonden zijn. Men zou kunnen stellen dat op een wiskundige manier de werking van het menselijk brein gemodelleerd wordt. De eenvoudige rekenkundige eenheden komen overeen met de neuronen in de hersenen, en het netwerk met de verbindingen tussen de neuronen.

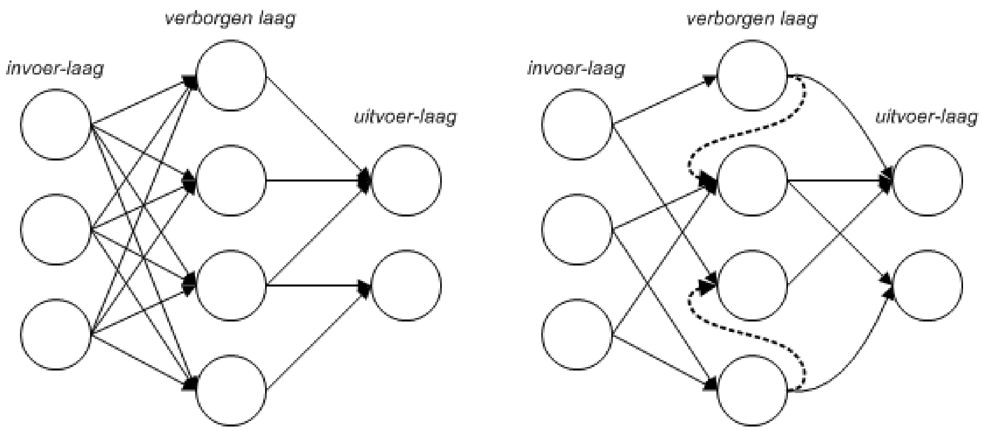
3.3.1 Principes

Een neuraal netwerk bestaat uit een verzameling knopen die met links verbonden zijn. Met deze links is telkens een gewicht geassocieerd. De leerprocessen in neurale netwerken vinden meestal plaats door deze gewichten aan te passen. Elke knoop heeft dus een aantal inkomende links die van andere knopen komen en een aantal uitgaande links die naar andere neuronen gaan. De knopen hebben ook een methode om op basis van de input een waarde te berekenen die met hun *activatiegrens* kan vergeleken worden om te kijken of zijzelf een signaal zullen uitsturen.

Het principe is erop gebaseerd dat elke knoop zijn berekeningen lokaal uitvoert, op basis van de input van zijn buren, zonder dat hiervoor gecentraliseerde organisatie nodig is. Er doet zich een vorm van *emergent intelligence* voor: intelligent gedrag ontstaat door de interactie van schijnbaar triviale eenheden.

3.3.2 Netwerkstructuren

Neurale netwerken worden meestal georganiseerd in lagen. Aan het ene uiteinde wordt de *input laag* onderscheiden die de invoer vanuit de wereld in het netwerk brengt. Aan het andere uiteinde heeft men de *output laag* die een resultaat zal presenteren op basis van de input. De eventuele tussenliggende lagen worden *verborgen lagen* genoemd (zie figuur 3.3).



(a) Een feed-forward neuraal netwerk met 3 lagen [21]

(b) Een recurrent neuraal netwerk met 3 lagen

Figuur 3.3: Een neuraal netwerk

Er wordt een onderscheid gemaakt tussen de netwerken op basis van de topologie. *Feed-forward* netwerken zijn acyclische netwerken waar de links slechts in één richting gaan: van de input richting output. Het voordeel van deze netwerken is dat de werking ervan goed begrepen wordt. Een nadeel van feed-forward netwerken is dat ze geen interne state kunnen voorstellen, aangezien de activatie van een vorige periode nooit meespeelt in de huidige berekening (door het ontbreken aan terugkerende links). Met deze netwerken kunnen adaptieve reflex-gebaseerde agenten gerealiseerd worden.

De andere soort worden *recurrent* netwerken genoemd. Bij deze soort kunnen de links ook terugkeren naar vorige lagen en nodes. In figuur 3.3(b) is een eenvoudig recurrent netwerk afgebeeld. Er zijn kleine stukken van de hersenen die vooral feed-forward zijn, maar over het algemeen kan men stellen dat de hersenen een recurrent netwerk zijn. Met dit soort netwerken kunnen veel complexere agenten en functies gemodelleerd worden. De berekeningen in een recurrent netwerk zijn echter niet zo ordelijk als die bij een feed-forward netwerk. Het kan voorkomen dat het netwerk onstabiel wordt, of begint te oscilleren. Dit houdt in dat de output van het netwerk zich niet stabiliseert, maar blijft fluctueren tussen twee of meer verschillende waarden.

3.3.3 Learning

Wanneer we een neuraal netwerk iets willen aanleren, gaan we het een aantal voorbeelden voorleggen. We geven bepaalde waarden voor de input en we kijken welke waarde er voor de output tevoorschijn komt. Als deze waarde foutief is, gaan we selectief de gewichten van de verbindingen aanpassen, zodat de uitkomst die het netwerk geeft meer in de buurt van de juiste uitkomst komt.

Voor eenvoudige netwerken zonder verborgen lagen (ook *perceptrons* genoemd) is dit een relatief eenvoudige taak. Wanneer we echter een multi-layered netwerk hebben, is het niet altijd even duidelijk bij welke knoop of verbinding de fout juist optreedt. Om hier het netwerk te laten bijleren maken we gebruik van *back-propagation*. Dit is een techniek waarbij er gekeken wordt welke knopen hebben bijgedragen tot een foutieve uitkomst. Bij die knopen wordt dan deels een aanpassing doorgevoerd. Dit doet men van aan de output laag, dan telkens één laag achterwaarts, tot men bij de verborgen laag het dichtst bij de input uitkomt.

3.3.4 Toepasbaarheid

Neurale netwerken zijn heel robuust en kunnen beter om met *noisy* gegevens dan de meeste andere AI methoden. Ze zijn uitermate geschikt om functies af te leiden uit observaties. Dit kan handig zijn wanneer de hoeveelheid gegevens te groot of te complex is om de functies handmatig af te leiden. Ze worden onder andere gebruikt om regressie-analyse te doen, patroonherkenning, enzovoort [21].

De netwerken zijn echter niet zo exact en expressief als eerste orde logica systemen (zie 3.1). Een ander nadeel is dat de werking van een neuraal netwerk niet volledig transparant is voor de gebruikers. Ook al geeft het netwerk bijna altijd een juiste output, het is zelden duidelijk *waarom* het dit doet. Dit maakt het ook moeilijk om een netwerk te intialiseren met vooraf opgedane kennis.

3.4 Robotica

De gebieden robotica en artificiële agenten zijn heel nauw met elkaar verwant. Net zoals bij een agent moet een robot ook zijn omgeving waarnemen, plannen opbouwen en deze dan uitvoeren. Het verschil is dat de robot zich in een echte wereld bevindt en de agent in een virtuele variant. Robots komen hierdoor veel vaker in contact met onvoorziene omstandigheden, omdat de echte wereld veel complexer en onvoorspelbaarder is dan een virtuele.

Het is eenvoudig in te zien dat bij het testen van de softwarecomponent van een robot, de lijn tussen robotica en intelligent agents erg vaag wordt: op dat moment is de software eigenlijk gewoon een agent. Het zal dan ook niet verbazen dat er tussen de beide gebieden theorieën en toepassingen uitgewisseld worden.

Hoofdstuk 4

AI Methodologiën

In dit hoofdstuk worden enkele standaardmethodes besproken die binnen de artificiële intelligentie worden gebruikt. De onderwerpen die aan bod komen zijn: logisch redeneren, zoekalgoritmes, regel-gebaseerde AI, werken met onzekerheden en leermethoden.

4.1 Logisch redeneren

De regels van eerste orde logica (zie sectie 3.1) kunnen door een intelligent systeem op een eenvoudige wijze benut worden om uit een hoeveelheid gegevens extra gegevens af te leiden door middel van logische deductie. Een dergelijk systeem kan dan vragen beantwoorden over informatie die uit zijn gegevensset kan afgeleid worden door hiervoor een logisch bewijs op te stellen.

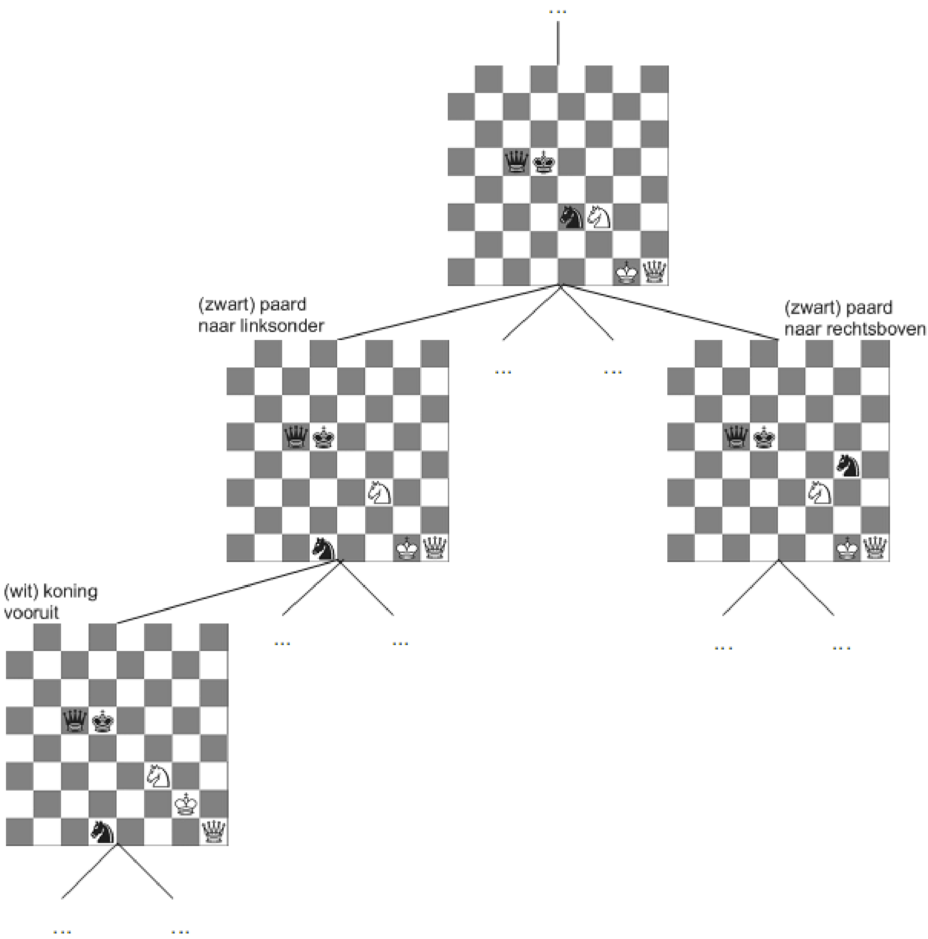
Er zijn twee manieren om met nieuwe gegevens om te gaan. Als we een nieuw feit toevoegen, kunnen we gaan kijken welke nieuwe conclusies we hieruit kunnen afleiden, en vervolgens opnieuw wat er uit deze laatste conclusies afgeleid kan worden. Dit heet *forward chaining*.

Bij de andere optie (*backward chaining*) starten we vanuit hetgeen we willen bewijzen, en proberen dan aan te tonen dat de premissen hiervan gelden [RN95].

4.2 Problemen oplossen

4.2.1 Oplossingen zoeken in de *state space*

Probleemoplossende agenten kunnen bepalen wat ze moeten doen door een bepaalde sequentie van activiteiten te vinden die leidt tot een gewenst doel. Zoals voorheen vermeld werd (zie sectie 3.2) kan een probleemdomein voorgesteld worden als een boom die een *state space* genoemd wordt. In deze boom kan dus een bepaald pad gezocht worden dat vertrekt vanuit de huidige situatie van de wereld en uitkomt in een situatie waarin aan



Figuur 4.1: Fragment van een statespace

bepaalde voorwaarden voldaan is. Dat kan bijvoorbeeld een situatie zijn waarin een doel bereikt is, of een spel gewonnen. Op deze manier vinden we dus een oplossing, en meteen ook de weg er naartoe in de vorm van een stappenplan dat door de verschillende staten gaat die tot de juiste oplossing leiden [RN95]. (Zoekmethodes worden verder behandeld in 4.3.)

Deze techniek wordt onder andere veel gebruikt bij het maken van artificiële intelligentie voor bordspellen zoals schaken en dammen, waar er een duidelijk afgelijnde status van de wereld kan geïdentificeerd worden. In figuur 4.1 zien we een stuk uit de boom die de statespace van een schaakspel voorstelt. We zien bijvoorbeeld op het tweede niveau van de boom dat zwart onder andere kan kiezen in welke richting hij zijn paard verplaatst. Dit zal een invloed hebben op het verder verloop van het spel. De AI speler van zwart kan dus in deze statespace zoeken naar een voor hem voordelig resultaat, bijvoorbeeld een situatie waarin wit schaakmat staat. Als hij die gevonden heeft kan hij de zetten kiezen die daar naartoe leiden, en kan hij op die manier een schaakspel spelen.

4.2.2 Oplossingen zoeken in de *planning space*

Bij de tweede aanpak wordt er gezocht in de *planning space* in plaats van de *state space*. Hier gaan we na voor bepaalde opeenvolgingen van acties of ze ons doel kunnen realiseren. We gaan deze dan gradueel en incrementeel uitbreiden en verfijnen terwijl we zoeken. Hier wordt meestal achterwaarts gewerkt: men vertrekt vanuit het doel en zoekt een reeks niet-conflicterende acties tot aan de beginstatus. Het voordeel bij deze aanpak is dat de acties meestal niet strikt geordend zijn; het maakt niet uit in welke volgorde ze uitgevoerd worden. Dit leent zich tot parallellisme, bijvoorbeeld in de vorm van samenwerkende agenten die tegelijkertijd een verschillende actie uitvoeren die hun dichter bij het doel brengt [CS05].

4.3 Zoekmethodes [RN95]

Zoekmethodes kunnen bij agenten voor verschillende doeleinden gebruikt worden. Een eerste mogelijkheid is deze in te zetten voor de navigatie van de agent binnen zijn omgeving. De omgeving zal dan eerst moeten voorgesteld worden als een boom of graaf, zodat daarop deze methodes toegepast kunnen worden. Het resultaat hiervan zal een route zijn die de agent moet volgen om van punt a naar punt b te komen.

Een tweede mogelijkheid is de agent een sequentie van acties te laten zoeken in een boom of graaf die de verschillende staten van de wereld voorstelt, bijvoorbeeld bij een bordspel (zie sectie 4.2.1).

Om een keuze te kunnen maken tussen de verschillende mogelijke stappen, wordt er een bepaalde 'kost' met elke link geassocieerd. Dit is een maat voor de inspanning die nodig is om deze link te overbruggen. Indien die voor alle stappen even groot is, spreken we van een graaf met een uniforme kostenverdeling.

4.3.1 Breadth-first

Een eerste intuïtieve manier om door een boom te zoeken is *breadth-first search*. Hier onderzoeken we de boom, laag na laag. We beginnen bij de wortel, dan alle kinderen van de wortel, dan alle kinderen van de vorige laag, enz. Indien de kost voor elke stap in de boom gelijk is, zal deze methode de kortste oplossing vinden, aangezien de boom niveau per niveau wordt afwerkt.

Het nadeel van breadth-first search is dat het algoritme erg veel geheugen inneemt, zeker bij bomen met een hoge vertakkingsgraad (b). Indien deze bijvoorbeeld 10 is, en de oplossing zich op diepte d bevindt, zal het algoritme $1 + 10 + 10^2 + 10^3 + ... + 10^d$ knopen moeten onderzoeken. Dit houdt in dat om een oplossing te zoeken die zich op het tiende niveau van de boom bevindt, er ongeveer één terabyte geheugen nodig is (gesteld dat een knoop kan opgeslagen worden in honderd bytes). Ook de tijdscomplexiteit is exponentieel ($O(b^d)$).

Het gevolg hiervan is dat enkel eenvoudige problemen met deze methode opgelost kunnen worden binnen afzienbare tijd.

Breadth-first search vindt de minst diepe oplossing in de boom, maar indien de kostenfunctie niet gelijk is voor elke stap, is dit niet de optimale oplossing. Een aanpak die het in die situatie beter doet is *uniform cost search*. In plaats van elke knoop van een specifiek diepteniveau te onderzoeken, kijken we enkel naar de knoop met de laagste kost en zoeken we van daaruit verder. Op deze manier is de eerste oplossing die gevonden wordt sowieso de goedkoopste, aangezien we de goedkoopste paden eerst onderzoeken.

4.3.2 Depth-first

De tegenpool van breadth-first search is *depth-first search*. Hierbij zoeken we altijd verder op de diepste knoop van de boom, tot we een doelknoop vinden, of tot we niet verder kunnen. In het laatste geval wordt er teruggekeerd en wordt er verder gegaan met de op één na diepste knoop. Het grote voordeel hiervan is dat het algoritme weinig plaats in het geheugen inneemt. De ruimtelijke complexiteit is hier $O(b * m)$, waarbij m de maximale diepte is van de boom. De tijdscomplexiteit is echter nog steeds exponentieel. In een probleemgebied waar er meerdere oplossingen zijn, kan depth-first search wel erg efficiënt zijn. Een nadeel treedt op wanneer de boom erg diep is, en depth-first al vroeg een fout pad begint te onderzoeken. Het zal dan heel lang duren voor hij terugkeert en een andere tak probeert. In oneindige bomen kan dit probleem er zelfs voor zorgen dat er nooit een oplossing gevonden wordt.

We kunnen de nadelen van depth-first search omzeilen door aan *depth-limited search* te doen: we stellen een maximum in op de zoekdiepte. Indien op deze maximale diepte geen doel gevonden is, keren we sowieso terug en proberen we een andere tak. De ruimtelijke complexiteit is $O(b * l)$, waar l de ingestelde dieptelimiet is. Omdat het niet altijd evident is een goede dieptelimiet te kiezen in een onbekend probleemdomein, kunnen we ook iteratief dieper en dieper zoeken, tot we een oplossing vinden. Deze aanpak heet *iterative deepening search*. Deze is optimaal en volledig, zoals breadth-first search, maar heeft een veel bescheidener geheugengebruik.

4.3.3 Bidirectional

Een andere manier om het excessieve geheugengebruik van breadth-first search te verminderen is *bidirectional search*. Hier beginnen we te zoeken vanuit zowel het startpunt als het doel. Het geheugengebruik wordt zo in de exponent gehalveerd: $O(b^d)$ wordt $O(b^{d/2})$, een serieuze verbetering.

Het probleemdomein moet zich hier wel toe lenen. We moeten op voorhand al weten waar een bepaalde oplossing zich bevindt, zodat we van daaruit kunnen achteruit werken. Bovendien moet de boom of graaf zo gebouwd zijn dat het makkelijk is om er achterstevoren door te navigeren (richting de ouders van een knoop, in plaats van richting de kinderen). Er moet ook op een efficiënte manier gecheckt kunnen worden of de recentst gevonden knopen zich in beide sets bevinden. Als dit het geval is hebben we een pad gevonden.

4.3.4 Best-first

Wanneer we extra informatie hebben over de nog af te leggen afstand tot het doel, kunnen we aan *best-first-search* doen: we zoeken verder aan de knoop die het meest voordelig lijkt. De eenvoudigste manier om dit te doen is de geschatte kost tot aan het doel minimaliseren, door altijd de meest optimistische schatting eerst te onderzoeken. Dit heet *greedy search*. De strategie geeft er dus de voorkeur aan om zo snel mogelijk een zo groot mogelijke stap te zetten in de richting van het doel. Deze zoekmethode vindt echter niet altijd de beste oplossing, en wanneer de boom een oneindige grootte heeft, kan het zijn dat het algoritme niet eindigt.

Omwille van plaatsgebrek is het niet altijd mogelijk de volledige gesorteerde lijst met alle reeds onderzochte paden op te slaan in het geheugen. Daarom kan ervoor gekozen worden slechts een bepaald aantal paden op te slaan, om zo het geheugengebruik binnen de perken te houden. Hier spreekt men van *beam search*. Het nadeel van deze aanpak is dat het optimale pad niet altijd gevonden zal worden.

4.3.5 A*

Als we aan deze schatting ook nog informatie over het reeds afgelegde pad mee in rekening brengen voor onze heuristische functie, dan komen we tot *A**. Voor elke knoop wordt de kostprijs van de afgelegde afstand opgeteld met de geschatte kost tot aan het doel. Hierdoor volgen we dus het voordeligste pad, maar dan vanaf het begin gerekend. Het voordeel hiervan is dat we met A* wel altijd de beste oplossing vinden, als er één bestaat.

Eigenschappen

A* heeft enkele eigenschappen die het een heel aantrekkelijk algoritme maken. Zo zal het algoritme altijd een oplossing vinden als die er is.

Wanneer er een *admissable* heuristiek gebruikt wordt, zal A* ook altijd het kortste pad vinden. Een *admissable* heuristiek is een heuristiek die nooit een overschatting maakt van de kost tot het einddoel.

A* is ook optimaal qua efficiëntie voor om het even welke gebruikte heuristiek. Dit wil zeggen dat er geen ander algoritme bestaat dat, gebruik makend van dezelfde heuristiek, minder paden zal onderzoeken dan A* [Sto00] [19].

Complexiteit

De tijdscomplexiteit van A* hangt af van de kwaliteit van de gebruikte heuristiek. Als de heuristiek een perfecte schatting zou zijn, zou het algoritme in één keer recht op het doel afgaan. Als de heuristiek overal nul is, wordt A* breadth-first search. In het slechtste geval is het aantal knopen dus exponentieel ten opzichte van de lengte van het kortste pad. Het is echter slechts polynomiaal als de fout op de heuristische functie niet groter is dan het logaritme van de perfecte schatting van de afstand [19].

IDA*

Omdat A*, zoals depth-first en greedy search, in oneindige bomen niet altijd eindigt, kan er ook hier gewerkt worden met een iteratieve aanpak. Bij A* leggen we een bovengrens op aan de totale kostprijs van het pad. Wanneer er voor een kostprijs onder deze grens geen oplossing gevonden wordt, gaan we deze grens verleggen en opnieuw zoeken, tot er een oplossing gevonden wordt. Deze aanpak noemt men *iterative deepening A**

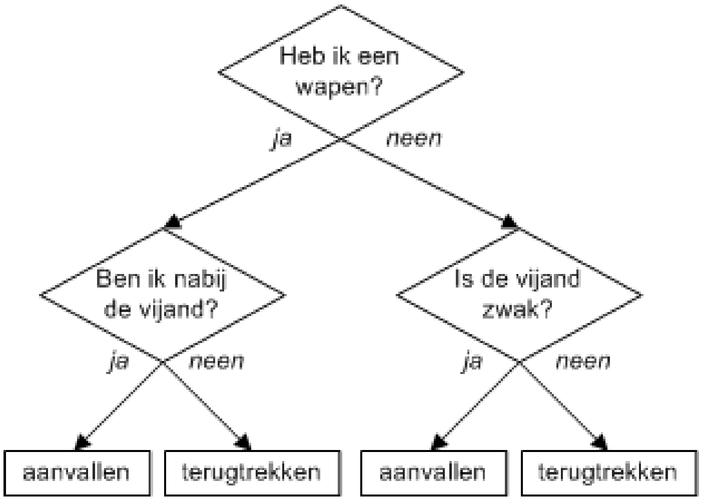
SMA*

Door het feit dat er telkens opnieuw gezocht wordt, kan het zijn dat er veel dubbelwerk gebeurt. Een aangepast algoritme onder de naam *simplified memory-bounded A** kan in dat geval een oplossing bieden. Het algoritme gebruikt al het geheugen dat het toegewezen krijgt. Indien er een knoop moet onderzocht worden, maar er is geen plaats meer, gaat SMA* de knoop met de slechtste heuristiek uit het geheugen verwijderen. Om te voorkomen dat er op deze manier te veel informatie verloren gaat, slaat SMA* in de ouderknoop informatie op over het beste deelpad van haar kinderen. Daardoor zal de knoop alleen opnieuw gegenereerd moeten worden als alle andere paden slechter lijken dan het pad dat verwijderd werd.

4.4 Regel-gebaseerde AI [PD05]

In veel games wordt de AI gerealiseerd door regel-gebaseerde systemen (zie 2.3). Het komt er bij deze systemen op neer dat een bepaalde situatie zal leiden tot een bepaalde actie. Er zullen regels zijn zoals *indien hongerig -> eet*.

Deze regels kunnen op verschillende manieren geimplementeerd worden. Een eerste aanpak is deze te definiëren in beslisbomen (*decision trees*). Hier hebben we op elke knoop van de boom een keuze, en naargelang de situatie zullen we in een andere knoop op het volgende niveau terechtkomen. Op het onderste niveau van de boom staan de acties zelf die dan uitgevoerd kunnen worden (zie figuur 4.2).



Figuur 4.2: Een beslisboom [PD05]

Een tweede mogelijke aanpak is gebruik maken van *finite state machines* (zie 3.2.1). Het voordeel is dat hier impliciet de verschillende mogelijke staten automatisch in elkaar overgaan. Deze techniek werd dan ook veel gebruikt in commerciële games zoals Quake, Half-Life, Doom, Age of Empires, ... [PD05].

Aan regel-gebaseerde systemen zijn ook enkele nadelen verbonden. Als er een uitzondering is op een bepaalde regel, dan moet deze ook beschreven worden (bijvoorbeeld: niet aanvallen als men bijna dood is). Als er op deze uitzondering weer uitzonderingen zijn, dan leidt dit al gauw tot een onoverzichtelijk netwerk van uitzonderingen dat lastig is om te onderhouden. Een ander nadeel is dat het gedrag van de AI volledig beperkt is door deze regels, en kan resulteren in niet-intelligent gedrag. Gedrag op basis van regels houdt zelden rekening met lange termijn gevolgen van acties, en kan slecht om met situaties die zelfs maar een klein beetje afwijken van de vooropgestelde situatie.

Een aanpak die meer geavanceerd is dan regel-gebaseerde systemen is het werken met model-gebaseerde AI. Hier worden de mogelijke effecten van de acties geëvalueerd, en wordt de actie gekozen die de beste verwachte uitkomst zal hebben. Hiervoor moet het effect van elke beslissing op de wereld gesimuleerd of geschat worden. In de literatuur[PD05] wordt er onder andere gewerkt met de Locust Engine, die een cyclus van 4 stappen gebruikt. De agent observeert de wereld en past op basis daarvan zijn interne voorstelling van de staat van de wereld aan. Hij neemt dan een beslissing op basis van de verwachte uitkomst van een bepaalde actie en voert dan deze actie uit. Bij deze aanpak is de agent niet gelimiteerd tot het regelgebaseerd gedrag, en kan hij beter inspelen op onvoorziene situaties.

4.5 Onzekerheden en waarschijnlijkheden [RN95]

Soms is het voor een agent niet mogelijk om alles met zekerheid te weten. Sommige regels gaan niet altijd op, maar slechts in een 'meerderheid' van gevallen. In zulke situaties is het handig dat we een maatstaf hebben om te werken met de waarschijnlijkheid van deze gevallen.

4.5.1 Waarschijnlijkheden

In *probability theory* houden we een waarschijnlijkheid bij voor onze logische zinnen. In plaats van te zeggen 'als een patiënt tandpijn heeft, heeft hij een gaatje', gaan we bijhouden dat dit slechts in 80% van de gevallen zo is. Het is mogelijk om deze waarschijnlijkheid te verhogen of te verlagen door meer bewijs te verzamelen, en zo een conclusie te formuleren.

We kunnen dan deze kansen gaan combineren door de regel van Bayes (formule 4.1) te gebruiken. Als we dan bijvoorbeeld $P(A|C)$ (de kans op A gegeven C) en $P(A|D)$ kennen, vervangen we in de formule van Bayes B door $C \wedge D$, en krijgen we zo een formule om $P(A|C \wedge D)$ te berekenen.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{4.1}$$

4.5.2 Fuzzy logic

Een bekende techniek die kan omgaan met vaagheden is fuzzy logic. Als we bijvoorbeeld een concept als 'Jan is groot' willen evalueren, wetende dat Jan 1m75 is, dan kan dat naargelang de situatie juist of onjuist zijn. In fuzzy logic zeggen we dat de *waarheidswaarde* van een functie zoals *isGrotePersoon(Jan)* schommelt tussen 0 en 1, en dus niet volledig juist of onjuist is.

Ook hier kunnen we waarheidswaarden gaan combineren volgens enkele regels. Als we willen berekenen wat de waarheidswaarde is van A *en* B, nemen we het minimum van de waarheidswaarden (zie formule 4.2). Om A *of* B te berekenen nemen we het maximum van de waarheidswaarden (zie formule 4.3).

$$T(A \wedge B) = min(T(A), T(B)) \tag{4.2}$$

$$T(A \vee B) = max(T(A), T(B)) \tag{4.3}$$

4.6 Plannen uitvoeren [RN95]

4.6.1 Execution monitoring

Eenvoudige agenten maken een plan en voeren het dan uit. Dit is een gevaarlijke strategie, want het zou kunnen dat de agent zijn plan gebaseerd heeft op onvolledige informatie van de wereld. Door te kijken wat er gebeurt terwijl hij het plan uitvoert, kan de agent evalueren of er iets misloopt. Op dat moment kan hij proberen opnieuw te plannen en kijken of zijn doel nog haalbaar is vanuit de nieuwe situatie.

4.6.2 Conditional planning

Een andere manier om met onvolledige informatie om te gaan is door een plan op te stellen voor elke mogelijke situatie. Door de wereld te observeren en de juiste voorwaarden te testen, kan de agent uitzoeken welk deel van het plan hij moet uitvoeren om zijn doel te bereiken. Het nadeel hieraan is dat een groot deel van de geplande acties niet zullen gebruikt worden, en dus ook niet berekend hadden moeten worden.

4.7 Dingen leren [RN95]

Het idee achter leren is dat de agent niet alleen zijn perceptie gebruikt om zijn handelingen te sturen, maar ook om zijn kennis te verbeteren en zijn gedrag te optimaliseren. De agent leert door zijn interactie met de wereld, en door het analyseren van zijn eigen beslissingsproces. Een eenvoudige manier van leren kan gewoon het memoriseren van voorbije ervaringen zijn.

4.7.1 Leren door observaties

In een geavanceerder stadium kan leren betekenen dat de agent dingen gaat afleiden door een theorie te formuleren over zijn observaties. Hier zou de agent een aantal voorbeelden voorgeschoteld krijgen onder de vorm van *input/output*-paren en daar een patroon uit proberen af te leiden. Als er dan een nieuw voorbeeld komt, kan de agent op basis van zijn afgeleid patroon proberen een inschatting te maken van wat het resultaat zal zijn. Het spreekt vanzelf dat als de agent meer (en liefst representatieve) voorbeelden te zien krijgt, zijn model van het probleem beter en accurater zal worden.

Het is de bedoeling dat de agent bij dit inductieve leerproces een zo eenvoudig mogelijk patroon vindt. Dit volgt het principe van *Ockham's razor*: de meest waarschijnlijke hypothese is de eenvoudigste die alle observaties verklaart.

4.7.2 Leren door bevestiging

Sommige domeinen zijn te uitgebreid om te kunnen aangeleerd worden door observaties. In deze domeinen kan het handig zijn de agent een evaluatiefunctie te leren die iets zegt over hoe groot de 'beloning' meestal is bij het uitvoeren van bepaalde acties. Deze techniek wordt ook gebruikt om dieren 'manieren' te leren, door feedback te geven op hun gedrag.

We zouden bijvoorbeeld elke keer dat een agent een schaakspel verliest kunnen stellen dat die reeks zetten een negatieve beloning veroorzaakt en elke keer dat hij wint een positieve. Initieel zal deze agent dus beginnen met willekeurige zetten uit te voeren die tot een bepaald einde zullen leiden. Telkens er een positief of negatief einde wordt bereikt, moet er een gewogen waarde toegekend worden aan de zetten die hiertoe geleid hebben.

Meestal is het leerzamer voor de agent om feedback te krijgen op elke individuele actie die hij uitvoert, en niet enkel op het geheel. Dit is echter niet altijd mogelijk.

Deel II

Huidige Ontwikkelingen binnen Artificiële Intelligentie

Hoofdstuk 5

Navigatie

In dit hoofdstuk vinden we enkele hedendaagse technologiën die door agenten gebruikt kunnen worden om te navigeren binnen virtuele omgevingen. Eerst worden enkele uitbreidingen op het A* algoritme besproken, waarna het concept *steering* wordt uitgelegd.

5.1 Uitbreidingen van A*

5.1.1 Optimaliseren voor meerdere aanroepen

Wanneer A* meermaals wordt aangeroepen kan het veel resources gaan gebruiken. Deze situatie zou bijvoorbeeld kunnen voorkomen in een *realtime strategy game* waarbij meerdere units min of meer hetzelfde pad moeten volgen. Er wordt een oplossing aangereikt die A* uitbreidt zodat het een optimaal pad kan berekenen, vertrekkende uit meerdere beginpunten, of eindigend in meerdere eindpunten. Hierbij wordt slechts één keer de graaf doorlopen. De snelheidswinst tegenover meerdere aanroepen van klassieke A* is vrij groot, want daarbij zou voor elk beginpunt en eindpunt de graaf één keer moeten doorlopen worden[MG05].

5.1.2 Optimaliseren voor een dynamische wereld

A* is het snelste *single source shortest path* algoritme dat er bestaat. Toch zijn er enkele nadelen aan verbonden. In een veranderende wereld zou men bij elke verandering helemaal opnieuw moeten beginnen zoeken, aangezien er mogelijk bepaalde links verdwenen zijn, heuristieken veranderd, enzovoort. In [Tom05] wordt hier een oplossing voor gesuggereerd. Men omzeilt het probleem door in de graaf enkel de knopen die grenzen aan de aangepaste knopen aan te passen. Het efficiëntie-voordeel tegenover klassieke A* wordt groter wanneer we met grotere grafen werken, aangezien daar de kost om te herberekenen ook groter wordt. Deze dynamische versie van A* staat bekend onder de naam D*.

5.1.3 Tactical pathfinding

In sommige virtuele omgevingen kan het nuttig zijn om tactische informatie mee te geven met de pahtfinding. Wanneer er zich vijandelijke of gevaarlijke situaties in de wereld bevinden, is het voor de agent nuttig om hiermee rekening te houden als hij door de wereld navigeert [vdS02]. Een intuïtieve aanpak hiervoor, is rekening houden met deze gevaren in de gebruikte heuristiek van A*. Paden die door een gevaarlijke zone komen, of in de buurt van vijanden, krijgen een slechtere heuristiek, waardoor het pad minder snel langs deze plaatsen zal komen.

Vaak is het nodig om nog met extra verschillen tussen deze gevarenzones rekening te houden, zoals bijvoorbeeld of een pad kort langs meerdere gevaren komt, of heel lang langs één gevaar. Het kan ook zijn dat een bepaald gebied meerdere gevaren overlapt, wat dan extra nadelig is voor de agent.

Met al deze dingen rekening houden heeft echter ook een nadeel: het in rekening brengen van al deze extra variabelen kost veel extra rekenkracht. In bepaalde situaties kan de toegevoegde complexiteit tot tien keer trager zijn dan een klassieke A* aanpak [vdS02]. Om hier een gulden middenweg voor te vinden, kan er gewerkt worden met ruwe benaderingen voor de gevarenzones die niet volledig accuraat zullen zijn, maar waarmee wel altijd een benadering van de veiligste route kan gevonden worden.

5.2 Steering [Rey99]

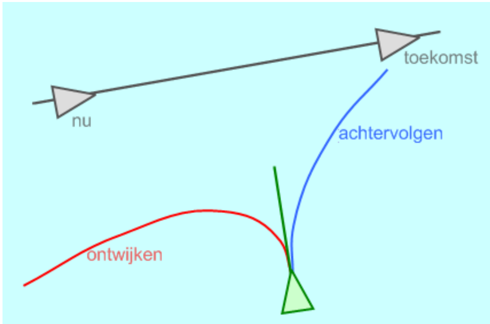
Naast pathfinding bestaat er nog een andere methode die door autonome karakters gebruikt kan worden om zich voort te bewegen in virtuele omgevingen: *steering behaviour*. Hier wordt geen gebruik gemaakt van een netwerk of grid van nodes. Het gaat hier meer om geïmproviseerd en geprogrammeerd gedrag dan om een echt plan om een doel te bereiken. Bij het sturen naar een bepaald doel gaat de agent zijn richting en snelheid aanpassen om zo dichter bij het doel te geraken. Combinaties van meerdere soorten behaviours kunnen gebruikt worden om complexere doelen te realiseren (bijvoobeeld 'ga naar een doel in het zuiden terwijl je obstakels ontwijkt').

5.2.1 Zoeken en vluchten

Zoekend gedrag is wanneer de agent een statisch doel wil bereiken. Hij zal zijn richting en snelheid aanpassen zodat zijn koers zal kruisen met het doelwit.

Wanneer het doelwit zelf een bewegend object is, spreken we van *achtervolgend gedrag*. Hierbij moet ook rekening gehouden worden met de toekomstige positie van het doelwit. Er zal een predictiefunctie gebruikt worden om hiervan een schatting te maken.

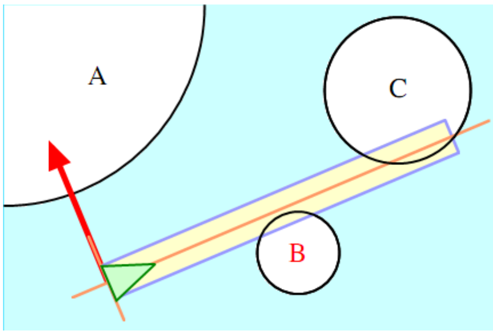
Het omgekeerde van zoekend gedrag is *vluchtend gedrag*. Daarbij zal de agent zich zo sturen dat hij van het doel weg beweegt. Ook hier is er de mogelijkheid om met een voorspelling te werken en te proberen de toekomstige positie van een bewegende vijand te ontwijken. We spreken dan van *ontwijkend gedrag* (zie figuur 5.1).



Figuur 5.1: Achtervolgen en ontwijken [Rey99]

5.2.2 Obstakels ontwijken

Wanneer een karakter op weg is naar een bepaald doel, en er staan obstakels in de weg (zie figuur 5.2), zal het zijn gedrag moeten aanpassen. Het grote verschil tussen obstakels ontwijken en vluchtgedrag is dat er bij vluchtgedrag van een object weggestuurd wordt, uiteindelijk in een bijna loodrechte richting. Bij ontwijken is het voldoende dat het karakter dicht langs het object passeert, en daarna terug de oorspronkelijke richting aanneemt.

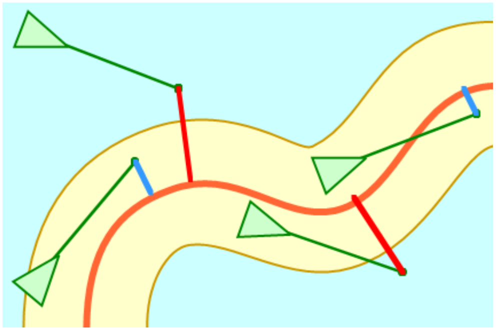


Figuur 5.2: Obstakels ontwijken [Rey99]

5.2.3 Rondwandelen

Het is mogelijk een karakter 'doelloos' te laten rondzwerven. Een naïeve aanpak zou kunnen zijn dat de richting elke seconde willekeurig aangepast wordt. Dit geeft echter een ongewenst resultaat: het karakter lijkt heen en weer te bibberen. Een betere aanpak is de huidige richting aanhouden en er elke seconde een kleine aanpassing aan maken, zodat een vloeiend resultaat ontstaat. Het onderwerp van rondzwerven is ook gerelateerd aan verkennen (exhaustief een bepaald oppervlak bekijken) en aan het verzamelen van grondstoffen (rondzwerven en verzamelen).

Een tweede manier om karakters te laten rondlopen, is door ze een pad te laten volgen (zie figuur 5.3). Bij het volgen van een pad wordt er een straal opgegeven waarbinnen de karakters het pad mogen volgen. Als zij dreigen af te wijken en buiten het pad terechtkomen zullen ze bijsturen om terug binnen het pad te komen.



Figuur 5.3: Een pad volgen met steering [Rey99]

Dit is niet hetzelfde als bij pathfinding, waar een karakter veel strikter een pad volgt. Men kan de vergelijking maken dat pathfinding eerder is zoals een trein die heel specifiek een treinspoor volgt; het volgen van een pad door steering lijkt meer op een voetganger op een voetpad.

5.2.4 Beweging in groep

Wanneer karakters in groep bewegen binnen een virtuele wereld zullen ze hun gedrag op elkaar moeten afstemmen. Ze moeten reageren op karakters in hun directe omgeving. Karakters die zich relatief ver van de agent bevinden worden genegeerd.

Spreiding

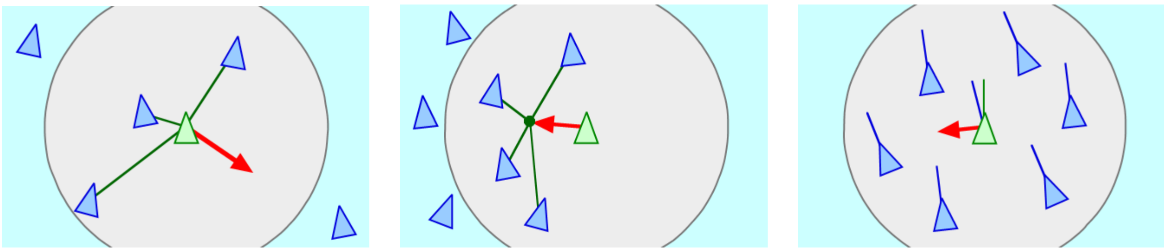
Het spreidend gedrag van karakters zorgt ervoor dat ze een bepaalde afstand kunnen handhaven tussen de anderen rondom hen. Voor elk karakter in de buurt wordt er gekeken waar deze zich bevinden en wordt er een afstotende kracht berekend. Al deze krachten worden opgeteld om de globale sturing te berekenen (zie figuur 5.4(a)).

Samenhang

Het tegengestelde van spreidend gedrag is werken naar samenhang. Dit stelt een individueel karakter in staat om een groep te vormen met andere karakters die in de buurt zijn. Dit gebeurt door een middelpunt te zoeken tussen alle nabije karakters en hier naartoe te bewegen (zie figuur 5.4(b)).

Uitlijnen

Een derde kracht die van belang is bij het komen tot een soort groepsgedrag is zich uitlijnen met de naburige karakters. Om dit te bereiken nemen we het gemiddelde van de richting en snelheid van alle naburige karakters en trachten dat te benaderen (zie figuur 5.4(c)).



(a) Spreiding (b) Samenhang (c) Uitlijning

Figuur 5.4: Groepsgedrag [Rey99]

Emergent Behaviour

Zoals gezegd kunnen deze gedragingen gecombineerd worden om zo tot complexer gedrag te komen. In het geval van beweging in groep kunnen dus deze drie (of eventueel nog andere) gedragingen via een gewogen som gecombineerd worden om zo een richting te bepalen voor elk karakter individueel. Door het feit dat alle karakters dit doen, zal het lijken alsof de groep op een gecoördineerde manier beweegt. Dit is echter niet zo, het is een indirect gevolg van het feit dat elk karakter zijn gedrag afstemt op dat van de karakters rond hem.

Hoofdstuk 6

Smart Objects

In dit hoofdstuk worden *smart objects* uitgelegd en enkele recente uitbreidingen en toepassingen ervan. Smart objects zijn objecten die informatie bevatten die aangeeft welke interactie ermee mogelijk is.

6.1 Algemeen [KT98]

Een smart object bestaat uit meerdere onderdelen. Als eerste bevat een smart object eigenschappen die onderdeel zijn van het design van het object, bijvoorbeeld de bewegende onderdelen, fysische eigenschappen zoals de vorm en het gewicht, enzovoort. Daarnaast bevat het interactie-eigenschappen zoals de positie van hendels, knoppen, handvatten, enzovoort. Er zijn ook verschillende behaviours toegankelijk, waarvan sommige wel of niet beschikbaar zullen zijn naargelang de status van het object. Tenslotte bevat het object ook extra informatie om de agenten te helpen: een soort hint naar welk gedrag een agent best vertoont als hij het object wilt benutten. De planning gebeurt door een actie te mappen op een object dat de actie toelaat, gevolgd door het opsporen van dit object en het uitvoeren van de actie.

Er zijn wel enkele mogelijke nadelen aan deze aanpak verbonden. Als het object de gedragingen van de agenten die ermee interageren zal bepalen, dan zullen alle agenten heel gelijkaardig of identiek gedrag vertonen. Dit valt sterk op in een virtuele omgeving en zal onnatuurlijk overkomen. Een ander nadeel is dat het gedrag van objecten en agenten gemengd wordt in de scripts, wat niet altijd de nodige afscherming zal voorzien die bij een object-georiënteerde aanpak gewenst is.

Een extra moeilijkheid duikt op wanneer smart objects door meerdere agenten gebruikt kunnen worden. Het is niet altijd evident om dergelijke situaties correct af te handelen. Een 'deur' object zal bijvoorbeeld moeten open blijven wanneer er langs twee kanten agenten door willen lopen en mag niet voortijdig sluiten.

6.2 3D interactie met smart objects

Interactie met objecten in een driedimensionale wereld is niet altijd eenvoudig. Om aan deze moeilijkheid tegemoet te komen kwam men op het idee smart objects uit te breiden met extra interactie-informatie voor de gebruiker [KT99]. De objecten bevatten dan nog extra informatie die 'verwacht gedrag' van de gebruiker zullen assisteren.

In het voorbeeld van een gebruiker in een driedimensionale wereld zouden dit hints kunnen zijn waar de gebruiker zijn handen moet positioneren om bepaalde behaviours van het object uit te lokken. Deze hints worden pas weergegeven als de gebruiker zijn hand in de richting van het object beweegt, om *clutter* in de wereld te vermijden. Hierdoor is het voor een gebruiker ook makkelijk om te zien hoeveel interactiemogelijkheden een object biedt. Dit zal overeenkomen met het aantal hints.

6.3 Planning met smart objects

In [ACT05] wordt het concept van Smart Objects uitgebreid door planning-gerelateerde data toe te voegen aan het object. Op deze manier moet de agent niet weten hoe hij moet interageren met elk soort object. Voor elke actie die het object aanbiedt, worden zowel de precondities voor deze actie opgeslagen als de effecten die de actie heeft.

Het voordeel hiervan is dat de agent zal kunnen interageren met tot dan toe onbekende objecten, dankzij de informatie die ze bevatten. Bovendien kan de agent zelf vrij eenvoudig ontworpen zijn: code voor navigatie en voor generische planning is voldoende.

Het is mogelijk dat de agent niet alle oplosbare problemen kan oplossen. Dit is het geval wanneer de agent nog niet alle objecten in zijn omgeving ontdekt heeft. Een andere mogelijkheid tot falen ligt erin dat het voor de agent niet altijd eenduidig zal zijn welke objecten hij moet selecteren om een bepaald doel te bereiken.

6.4 Efficiënte animatie-informatie

In [JWL05] wordt een belangrijke aanpassing gesuggereerd voor Smart Objects. Deze werd ontwikkeld voor Collaboratieve Virtuele Omgevingen (CVE). Een CVE is een, meestal genetwerkte, virtuele omgeving waarin meerdere deelnemers samen kunnen interageren met de wereld. De deelnemers kunnen zowel mensen als artificiele agenten zijn. In een dergelijke CVE moet de status van de wereld verdeeld en gesynchroniseerd worden tussen alle deelnemers via het netwerk, dit om de illusie te creëren dat ze zich op hetzelfde moment op dezelfde plaats bevinden. Het is dus essentieel voor de performantie dat de hoeveelheid gegevens, die over het netwerk verstuurd wordt, minimaal is.

Het probleem met de gewone smart objects is dat er animatie-informatie in het object zelf opgeslagen wordt. Dit wil dus ook zeggen dat voor elke verschillende soort avatar (mens, hond, kat, robot,...) er een aparte hoeveelheid animatie-informatie aan het object moet toegevoegd worden. Dus bij elk nieuw soort object dat toegevoegd wordt, moet er rekening gehouden worden met alle soorten avatars. Wanneer er een nieuw soort avatar toegevoegd wordt, moeten dus ook alle bestaande objecten aangepast worden! Deze animatie-gegevens leiden al gauw tot zeer grote data hoeveelheden die niet bruikbaar zijn om in een gedistribueerd systeem rond te sturen.

In plaats van interactie- en animatie-informatie voor alle soorten avatars op te slaan in het smart object, wordt de interactie-informatie hier veel algemener gehouden. In plaats van de volledige animatie per avatar op te slaan, worden er voor elk object interactiepunten bijgehouden. Elke avatar bepaalt zelf hoe hij deze punten activeert (bijvoobeeld een mens met zijn hand, een hond met zijn snuit, een robot door er tegen te rijden, ...). Hierdoor moet er in het object geen animatie-informatie opgeslagen worden. Deze wordt lokaal door alle deelnemers berekend. Om tot een realistische animatie te komen, wordt er gebruik gemaakt van Inverse Kinematics (zie verder).

6.4.1 Object ontwerp

De eigenschappen van het object kunnen in drie categorieën worden onderverdeeld: object-eigenschappen, interactie-eigenschappen en behaviours. Objectonderdelen kunnen at runtime toegevoegd en verwijderd worden. De object-eigenschappen zijn de fysieke eigenschappen zoals fysische en grafische informatie, een naam en beschrijving, en de statusvariabelen.

De statusvariabelen stellen het object in staat zijn gedrag aan te passen naargelang de staat waarin het zich bevindt. Zo kan bijvoorbeeld een televisie de functie 'zap' aanbieden als hij aangezet is, en deze functie verbergen indien hij uitgeschakeld is. De interactie-eigenschappen zijn de verzameling commando's die acties en gedragingen van het object in gang kunnen zetten, met eventuele parameters. De behaviours zijn scripted gemaakt, zodat ook deze gewijzigd kunnen worden zonder dat het noodzakelijk is opnieuw te compileren. In deze scripts kunnen de statusvariabelen aangesproken en gewijzigd worden.

De grote verdienste van dit ontwerp is dus dat de wereld kan veranderd worden zonder dat hij opnieuw zal moeten opgestart worden. Er kunnen objecten toegevoegd en verwijderd worden, objectonderdelen kunnen toegevoegd en verwijderd worden en de scripts kunnen at runtime aangepast worden. Hierdoor is de flexibiliteit in een dynamische wereld gegarandeerd.

6.4.2 Inverse kinematics

Het is ook noodzakelijk dat de animatie-informatie op een efficiënte manier opgeslagen en verspreid wordt. Tegenwoordig wordt voor animaties meestal *skeletal animation* gebruikt. Hierbij wordt er in een object een virtueel skelet gemodelleerd. Op basis van de mogelijke bewegingen van dit skelet wordt het model vervormd om zo realistische bewegingen te bekomen. De animaties worden vervolgens op de verbindingen van het skelet gedefiniëerd, in plaats van op het ganse model, hetgeen veel efficiënter is.

Wanneer gebruik gemaakt wordt van *keyframe animatie* om een animatie op te slaan, worden enkele opnames gemaakt van de posities van de verbindingen van dit skelet. Deze worden dan achter elkaar afgespeeld (en eventueel geïnterpoleerd) om zo tot een mooie animatie te komen. Het nadeel is echter dat deze animatie volledig statisch is, en dat de gegevens voor meerdere keyframes opgeslagen moeten worden.

Een aanpak die niet aan deze euvels lijdt is *inverse kinematics*. Hier wordt gebruikt gemaakt van het rechtstreeks specificeren van de gewenste positie en oriëntatie van het losse uiteinde van een aaneenschakeling van verbindingen (bijvoorbeeld de hand van een arm). Het animatiesysteem zal vervolgens berekenen hoe de verschillende verbindingen in deze ketting moeten gepositioneerd en georiënteerd worden opdat het uiteinde op die specifieke plaats kan terechtkomen.

Het grote voordeel hiervan is dat enkel de positie van het uiteinde moet opgeslagen of doorgestuurd worden. Bovendien is het op deze manier mogelijk om de animatie at run-time aan te passen aan een nieuwe situatie in de wereld. Een keyframe animatie van een grijpende hand, zal altijd op exact dezelfde wijze op exact dezelfde hoogte terechtkomen. Dit is natuurlijk niet altijd de bedoeling. Sommige objecten zullen groter of kleiner zijn, of zich iets hoger of lager in de wereld bevinden.

Hoofdstuk 7

Libraries en Talen

In deze sectie wordt er kort een selectie gemaakt van enkele libraries en scripting-talen. Deze kunnen gebruikt worden bij de implementatie van smart objects (zie 6.4.1) of in die van de agenten zelf, om ervoor te zorgen dat deze makkelijker aangepast kunnen worden.

7.1 Scripting-talen

Een scripting-taal (soms ook uitbreidingstaal genoemd) is een soort programmeertaal die een applicatie kan sturen. Vaak zijn deze scripts niet in dezelfde taal als de programma's waarin ze werken. Ze zijn meestal toegankelijk voor de eindgebruiker en voorzien zo een manier om het gedrag van een applicatie aan diens wensen aan te passen. Meestal worden scripts niet gecompileerd, maar at runtime geïnterpreteerd [25].

7.1.1 Scripting in virtuele omgevingen

Scripting-talen gebruiken binnen virtuele omgevingen kan enkele grote voordelen bieden. Zij kunnen er bijvoorbeeld voor zorgen dat bepaalde aspecten van de wereld aangepast kunnen worden en dat deze toch niet gehercompileerd of heropgestart moet worden. Zij kunnen dus ingezet worden om bepaalde aspecten van de planning van de agent te implementeren, of om de behaviour-aspecten van smart objects uit te werken [JWL05].

Het grote voordeel van werken met een scripting-taal tegenover bijvoorbeeld een taal om gegevens in op te slaan, zoals XML, is dat er in een scripting-taal procedureel gewerkt kan worden. Als we enkele objecten in een cirkel willen plaatsen op basis van een XML-bestand, zouden we de coördinaten van deze objecten vooraf moeten berekenen en elke keer manueel moeten updaten om bijvoorbeeld de grootte van deze cirkel te wijzigen. In een scripting taal kunnen we een procedure schrijven die de formule van een cirkel benut en op deze manier de objecten op de correcte locatie plaatst. Indien we in dat geval de grootte van de cirkel zouden willen aanpassen moeten we enkel de straal wijzigen die we aan de procedure meegeven.

7.1.2 Lua

Lua is een scripting taal die ontwikkeld werd aan de Universiteit van Rio de Janeiro. Deze taal is gratis verkrijgbaar, heel compact en erg krachtig. Ze wordt meestal gebruikt in combinatie met C++, maar kan ook gecombineerd worden met Java, C#, Perl en vele andere talen [12].

Ze heeft zich al bewezen in meer dan 20 industriële applicaties en meer dan 30 games, waaronder Adobe's Photoshop Lightroom, World of Warcraft en Far Cry [22]. In de meeste van deze applicaties staat Lua in voor de aanpasbaarheid van de interface. Dankzij het feit dat het een scripting taal betreft kunnen er dus aanpassingen doorgevoerd worden zonder dat er gehercompileerd moet worden.

7.1.3 Python

Python is een *high-level* scripting-taal, welke onder andere functioneel programmeren, object georiënteerd programmeren, en imperatief programmeren ondersteunt. Python is opensource, en wordt ontwikkeld sinds 1991 [23], [14].

Python wordt gebruikt in meer dan 50 applicaties en games, waaronder het statistisch pakket SPSS, de originele BitTorrent client, het en spel Battlefield II. Ook enkele grote bedrijven zoals Google en Yahoo gebruiken Python om bij te dragen aan hun portaalsites [24].

7.2 Libraries

In het domein van artificiële intelligentie zijn er verscheidene toolkits beschikbaar [1]. Sommige hiervan zijn commercieel, andere zijn beschikbaar onder een open source licensie. Vele hiervan zijn echter vrij specifiek geschreven voor een bepaalde situatie, een bepaalde engine of een bepaald spel. Vandaar ook het initiatief om zelf een pathfinding library in het leven te roepen (zie sectie 8).

7.2.1 OpenSteer

Opensteer is een bekende steering library, ontwikkeld door Craig Reynolds voor Sony Computer Entertainment America. De library is vrij beschikbaar onder de MIT open source license. Ze bevat enkele duidelijke voorbeelden over de verschillende soorten steering (zie 5.2), die aangepast kunnen worden teneinde snel bepaalde ontwerpen uit te testen.

Deel III

Implementatie

Hoofdstuk 8

A* Library

Hier wordt kort de implementatie van de pathfinding library besproken.

Er bestaan meerdere libraries die pathfinding oplossingen aanbieden, maar de meeste hiervan zijn niet eenvoudig aan te passen aan specifieke situaties. Daarom is het doel van deze implementatie een eenvoudige A* library maken die klein, snel, en aanpasbaar is zodat deze in verscheidene situaties gebruikt kan worden.

8.1 Knopen

De bibliotheek laat de gebruiker toe knopen aan te maken. Aan deze knopen kan een heuristische waarde toegekend worden. Deze dient een representatieve schatting te zijn voor de nog af te leggen afstand.

In deze bibliotheek worden de coördinaten niet sowieso in één bepaald formaat opgeslagen, omdat wanneer er voor één specifiek systeem van coördinaten gekozen wordt, dit niet noodzakelijk geschikt is voor de situatie waarin de gebruiker zich bevindt (zij het in 2D, 3D, ...). Er is dus geopteerd voor een generiek object, dat door de gebruiker naar wens kan ingevuld worden.

De gebruiker kan daarbij ook nog andere relevante informatie associëren met deze knopen, zodat hij deze informatie nadien nuttig kan gebruiken. Voorbeelden zijn onder andere de namen van de knopen, informatie voor de AI zoals hoe gevaarlijk de zone rond de knoop is, enzovoort ...

8.2 Links

Via de bibliotheek kunnen links toegevoegd worden tussen de knopen, zodat een graaf in het geheugen kan opgesteld worden. Op deze graaf kan dan het A* algoritme gebruikt worden om een pad te vinden. Hiervoor moet dus elke link een bepaald geassocieerd gewicht krijgen dat de kost aangeeft om dat pad te gebruiken. Er is zowel de mogelijkheid om te werken met tweewegsverbindingen (voor ongerichte grafen) als éénwegsverbindingen. In het geval van een tweewegsverbinding zal de verbinding in de omgekeerde richting automatisch toegevoegd worden.

8.3 Pathfinding

Wanneer de graaf geïnitialiseerd is, kan er met een eenvoudige aanroep het standaard A* algoritme geactiveerd worden. Als parameters hiervoor zijn het start- en eindpunt nodig. Het algoritme geeft vervolgens het optimale pad hiertussen terug (gesteld dat dit bestaat) in de vorm van een geordende lijst met knopen.

8.4 Gebruikte tools

De bibliotheek werd eerst ontworpen in *Visual Studio*[16] in de taal C# , omdat deze taal beschikt over verscheidene datastructuren die voorzien zijn van uitgebreide operaties. Dit is zeer geschikt voor het A* algoritme, omdat de snelheid van het algoritme sterk afhankelijk is van de efficiëntie van de datastructuren. A* gebruikt onder andere een *priority-queue* waarin de gedeeltelijke paden worden opgeslagen, een lijst waarin de knopen van de paden worden opgeslagen en een verzameling knopen waarmee de hele graaf voorgesteld wordt.

Om de bibliotheek iets meer *general-purpose* te maken, is er nadien gekozen om dezelfde functionaliteit ook aan te bieden in een C++ bibliotheek. Meestal resulteert C++ code bovendien in iets snellere applicaties dan diezelfde functionaliteit gemaakt in C# code [13].

Hoofdstuk 9

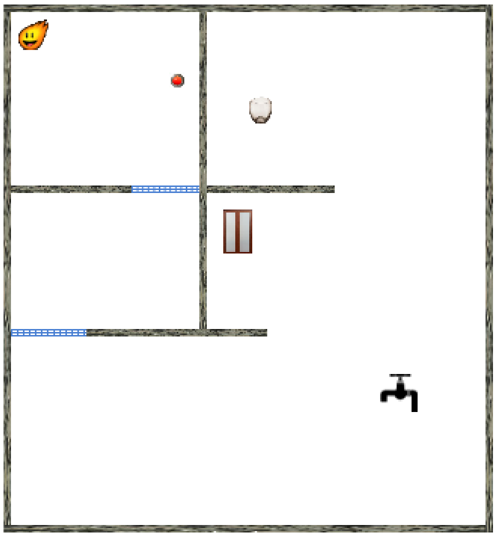
Testomgeving

Dit onderdeel behandelt de testomgeving waarin de agent ontwikkeld is. Eerst wordt uitgelegd hoe de wereld waarin de agent zich bevindt in elkaar zit. Hierop volgt een korte beschrijving van de intelligente objecten die zich in deze wereld bevinden. Tot slot wordt kort de testapplicatie zelf nader toegelicht.

9.1 Virtuele wereld

De wereld van de agent bestaat uit een bewandelbaar oppervlak, waarop zich meerdere objecten bevinden. Sommige van deze objecten zijn nuttig voor de agent, en met deze objecten is interactie mogelijk. De overige zijn obstructies of decor, zoals muren. Het is de bedoeling dat de agent door deze wereld loopt en binnen zijn visueel bereik de wereld observeert. Als er objecten aanwezig zijn, kan hij deze opmerken en in zijn *knowledge base* opslaan. Om de het testen te vereenvoudigen wordt er vanuit gegaan dat de exploratie voltooid is, en weet de agent reeds van het bestaan van alle objecten. De huidige implementatie is in 2D, maar aangezien de pathfinding onafhankelijk is van het coördinatensysteem van de wereld, maakt dit weinig uit: dezelfde agent zou zich in een 3D wereld ook kunnen voortbewegen (zie 8.1).

Een screenshot van de wereld is te bekijken in figuur 9.1. De voorbeeldwereld bestaat uit enkele kamers, verbonden met deuren. De deur in de kamer waar de agent zich bevindt, kan enkel geopend worden door op de ronde knop te drukken. De andere deur kan geopend worden op de gebruikelijke wijze, aan de deur zelf. De overige objecten die zich in deze wereld bevinden zijn: een glas dat een object kan bevatten (bovenaan in het midden), een kraan die water kan produceren (onderaan rechts), en een kast (centraal) die ook objecten kan bevatten. In deze eenvoudige maar veelzijdige wereld kunnen aspecten van de planning en de pathfinding op een transparante manier uitgetest worden.



Figuur 9.1: De 2D wereld met enkele deuren en objecten.

9.2 Intelligente objecten

De objecten in deze wereld bevatten een aantal standaardattributen: fysieke eigenschappen zoals de locatie, de grootte, het uitzicht, het gewicht, of het object beweegbaar of statisch is, enzovoort. In een 3D versie kunnen daarbij ook nog interactiepunten op het object aangegeven worden (grijppunten, knoppen, ...). Daarbuiten kan de agent aan deze objecten vragen hoe ermee geïnterageerd kan worden. Ze bevatten informatie over hun toepassingsgebied, de operaties die erop uitgevoerd kunnen worden, en de resultaten en voorwaarden voor deze operaties.

Het idee van objecten die zelf weten welke interactie ermee mogelijk is, is niet nieuw. Net zoals in de besproken versie van [JWL05] is het niet de bedoeling dat er in deze objecten avatar-specifieke animatie-informatie opgeslagen wordt. Ze bevatten zoals vermeld wel een fysieke omschrijving en een beschrijving van de interactiemogelijkheden (zie sectie 6.4).

De eigenschappen van de objecten kunnen at runtime gewijzigd worden in de interface (zie bijlage B.3), zodat onder andere muren verplaatst kunnen worden, deuren geopend en gesloten, objecten gereset, ... Op deze manier is het eenvoudig om verschillende dingen te testen zonder telkens opnieuw de wereld te moeten inladen.

9.3 Testapplicatie

Het geheel van de agent en de virtuele wereld zit vervat in een testapplicatie. Deze applicatie werd ontworpen in *Visual Studio*[16] in de taal C#. Deze taal maakt het mogelijk snel een eenvoudig een interface te ontwerpen die kan dienen als omgeving om bepaalde algoritmen in te testen. C# bevat bovendien *reflectie*, wat heel handig kan gebruikt worden bij het prototypen van interactie met intelligente objecten.

De interface bestaat uit een 2D weergave van de wereld in zijn huidige staat, een *property-grid* die het mogelijk maakt de eigenschappen van objecten in de wereld aan te passen, en enkele invoervelden waarmee de gebruiker de agent bevelen kan geven die deze dan moet trachten uit te voeren. Er is ook een uitvoerveld voorzien waarin de agent feedback kan melden over zijn planning en vooruitgang.

Enkele screenshots van de testapplicatie zijn opgenomen in bijlage B.

Hoofdstuk 10

Een Agent

In dit deel wordt de implementatie van de agent uit de doeken gedaan. Eerst wordt de methode uitgelegd die de agent gebruikt om zich binnen de virtuele wereld voort te bewegen. Hierna volgt een onderdeel dat behandelt hoe de agent problemen oplost, afgerond met een voorbeeld.

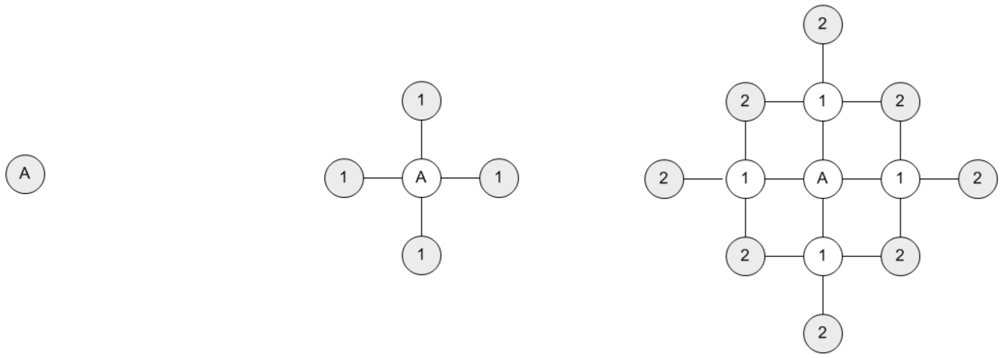
10.1 AI mesh

Als we A* willen gebruiken voor de pathfinding van de agent, komen we voor enkele problemen te staan. In een wereld waar dingen kunnen veranderen, objecten kunnen bijkomen en verdwijnen, is een statische graaf niet erg nuttig. Wanneer een muur van plaats verandert of een deur opent of sluit, is een vooraf berekende statische graaf niet meer bruikbaar. Wat voor de agent het meest flexibel is, is dat hij op het moment dat hij wil rondwandelen zelf een graaf opstelt die past binnen de huidige staat van de wereld (met bepaalde objecten al dan niet in de weg, sommige deuren open of gesloten). Zulk een graaf die door een agent gebruikt wordt om door de wereld te navigeren noemen we een *AI mesh*. Voor het opstellen van deze graaf zijn enkele intuïtieve benaderingen mogelijk.

10.1.1 Reguliere grid

De agent kan vanuit zijn startpositie in de vier windrichtingen een stap bekijken van een bepaalde grootte. Indien deze niet met een muur of ander object in contact komt voegt hij deze toe aan de mesh. Die punten linkt hij aan het beginpunt. Daarna doet hij dit opnieuw (recursief) voor de op deze manier gevonden punten, totdat hij het hele veld vol heeft gezet met punten voor alle bereikbare plaatsen.

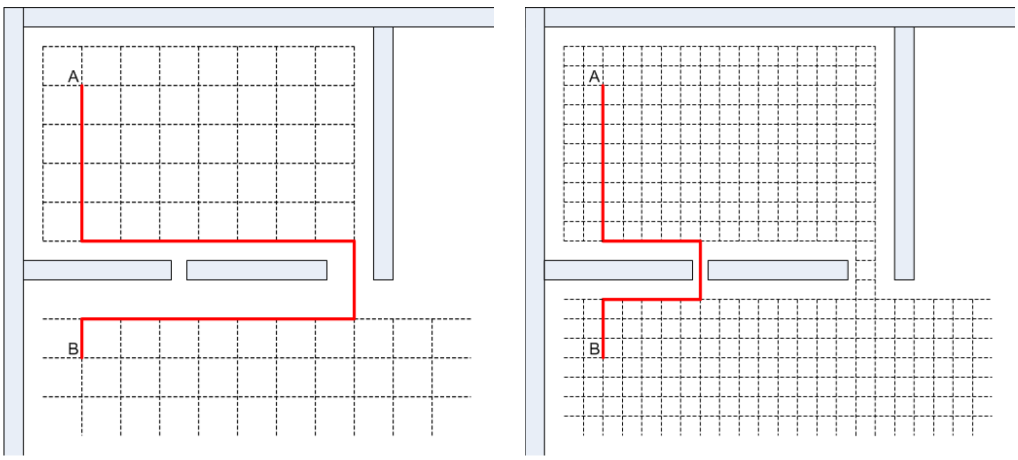
In figuur 10.1 zien we hoe dit in zijn werk gaat. Na een eerste recursieve aanroep zijn er 4 extra knopen toegevoegd aan de graaf (zie figuur 10.1(b)). Na de tweede recursieve aanroep komen hier nog 8 punten bij (zie figuur 10.1(c). Wanneer het algoritme een knoop wil toevoegen die reeds bestaat, zal in plaats van een extra knoop, er een link naar de bestaande knoop op die locatie gemaakt worden.



(a) Initieel (b) Na één recursie aanroep (c) Na een tweede recursieve aanroep

Figuur 10.1: Opbouwen AI mesh

Door de stapgrootte x van de grid te wijzigen kan de 'resolutie' van de grid verhoogd of verlaagd worden. Een grid met een hoge x heeft dus 'grote mazen' en weinig nodes. Berekeningen op een dergelijke *sparse* grid zijn heel snel, maar als de x te groot wordt kan het zijn dat bepaalde openingen tussen de muren 'voorbijgewandeld' worden (zie figuur 10.2(a)) en dan zal de agent deze niet kunnen gebruiken. Een grid met een heel lage x heeft extreem veel nodes en zal dus trager berekenen, maar kan wel door fijne openingen tussen muren glippen (zie figuur 10.2(b)). Om volledig zeker te zijn dat er geen openingen gemist worden zou een grid van één pixel moeten gebruikt worden. Dit is echter onaanvaardbaar qua performantie.

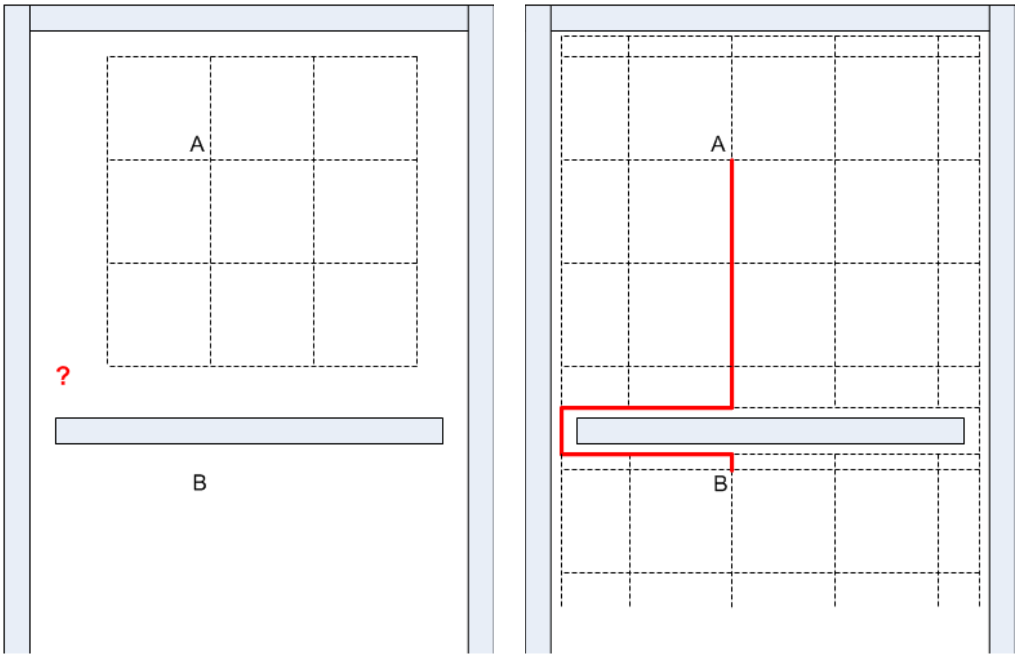


(a) Grove mesh, opening wordt gemist (b) Fijnere mesh, opening kan benut worden

Figuur 10.2: Mesh resolutie

10.1.2 Grid van variabele grootte

Een mogelijke oplossing voor de problemen die bij een reguliere grid ontstaan zijn het dynamisch aanpassen van de grootte van de grid. Hierdoor zouden de ’overschot’ stukjes grid kunnen opgevuld worden en zouden de deurgaten die zich in de hoeken bevinden bereikbaar zijn (zie figuur 10.3). Dit lost echter het probleem nog niet op voor deurgaten in het midden van een muur.



(a) Mesh zonder rest-stukjes

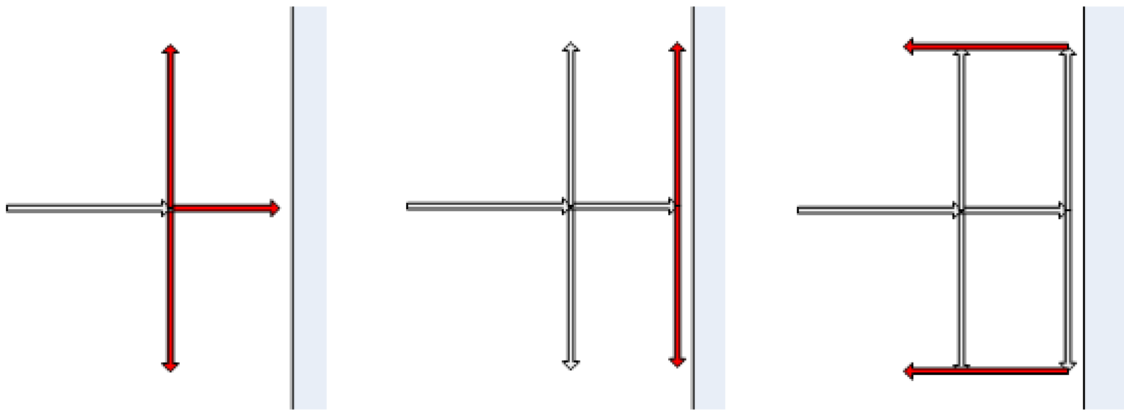
(b) Mesh met rest-stukjes

Figuur 10.3: Variabele mesh

Een nadeel van de aanpak met variabele grootte is dat er vanuit dit stukje irreguliere grid opnieuw een grid kan ontstaan die het gebied van de reguliere grid zal overlappen (zie figuur 10.4(c). Wanneer we werken met een reguliere grid komt dit probleem niet voor, door het feit dat links die terugkeren in de richting van het originele grid daar altijd perfect terug op uitkomen omdat steeds dezelfde afstand gehanteerd wordt.

In een irreguliere grid ontstaat hierdoor een ’dubbel’ grid dat niet altijd een juiste oplossing zal voorzien. Dit probleem kan omzeild worden op meerdere manieren. Men kan de irreguliere grids laten uitsterven na enkele stappen, maar dit kan er dan weer voor zorgen dat bepaalde delen van het veld onbedekt blijven. Dit zien we in het onderste deel van figuur 10.3(b).

Een betere optie is het werken met een gememoriseerde stapgrootte: als er in een bepaalde richting een irreguliere stap is gezet van grootte y, moet de volgende stap in de *omgekeerde* richting ook grootte y hebben. Op die manier zou de stap in de omgekeerde richting terug perfect op de reguliere grid uitkomen.



(a) Initieel

(b) Na één recursieve aanroep

(c) Na een tweede recursieve aanroep overlappen de pijlen een ander stuk van de grid

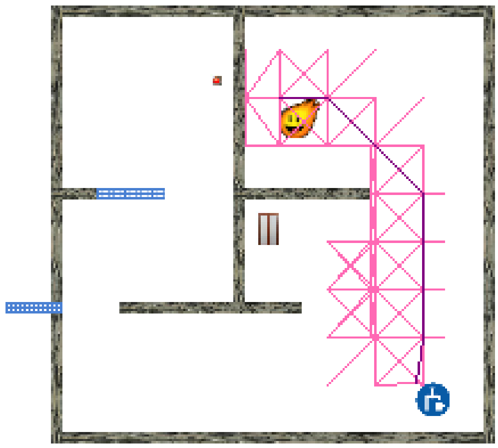
Figuur 10.4: Niet-reguliere AI mesh in de buurt van een muur

Bij het opbouwen van een mesh rond een onregelmatig obstakel is het mogelijk dat de uitlopers van de grid die voorbij het obstakel komen, niet mooi op elkaar aansluiten. Dit komt doordat de stappen die voorbij het object geraken, niet noodzakelijk vanop dezelfde afstand tegenover de reguliere grid vertrekken. Ze komen dan ook voort uit knopen die op verschillende afstanden ingekort zijn geweest, enkele stappen terug. Om dit op te lossen, moet ook de stap die *volgt* op een onregelmatige stap (en in dezelfde richting gaat) een aangepaste grootte krijgen. Deze stap moet dan bestaan uit het verschil tussen de standaard-stapgrootte en de speciale stapgrootte y. Op die manier ontstaat ook na het obstakel terug een regulier patroon.

10.1.3 Heuristisch opbouwen

Het is mogelijk om het opstellen van de AI mesh te optimaliseren, gebruik makend van de concepten die achter A* schuilen. Hierbij is het een vereiste dat we een eenvoudige of berekenbare heuristiek hanteren zoals bijvoorbeeld de afstand in vogelvlucht. We breiden eerst de knopen uit die ons dichter bij het doel lijken te brengen, rekening houdend met de reeds afgelegde afstand. Indien we ergens vast komen te zitten nemen we de tweede beste keuze, enzovoorts. Dit blijven we doen tot een voldoende groot deel van de AI mesh (een *spine*) is opgesteld om van de agent naar zijn doel te geraken. In het slechtste geval zal dit de volledige AI mesh opleveren, in het beste geval zal dit het optimale pad zijn met enkele korte vertakkingen.

Een illustratie van het concept is te vinden in figuur 10.5. Het is hier duidelijk overbodig om een mesh op te stellen voor het linkerdeel van de wereld, aangezien dit deel nooit benut zal worden tijdens het zoeken van een pad van de agent naar de kraan. De oppervlakte van de mesh wordt in dit geval door deze optimalisatie gehalveerd. Het zal dus aanzienlijk



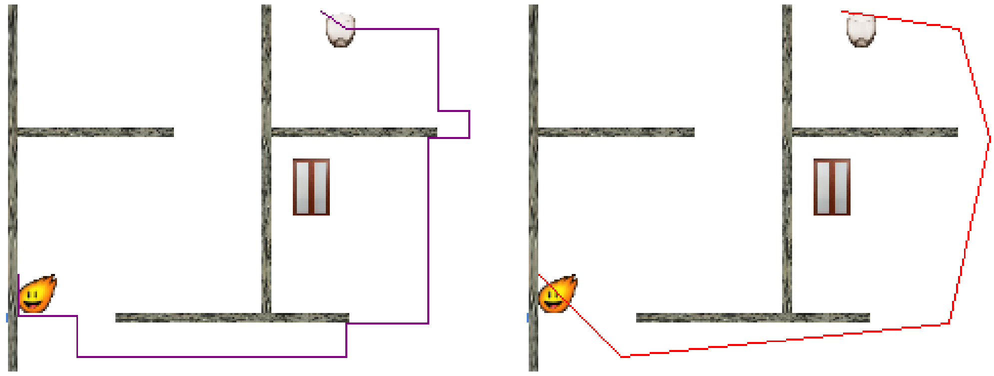
Figuur 10.5: Een subset van de mesh die het optimale pad bevat

minder tijd kosten om deze mesh op te stellen en daarop het pad te zoeken.

10.1.4 Hoekige paden

Door het gebruik van een grid op basis van rechte hoeken, ontstaat er een onnatuurlijk effect als de agent een pad op deze grid moet doorlopen. Om dit effect tegen te gaan kan er geopteerd worden de diagonalen van de grid ook toe te voegen. Dit lost het euvel echter slechts gedeeltelijk op.

Om tot een visueel aantrekkelijker resultaat te komen wordt een eenvoudige *smoothing* op het verkregen pad uitgevoerd. In het gebruikte algoritme wordt het pad éénmaal doorlopen. Er wordt voor de knopen die op de eerste knoop volgen telkens gekeken of er een rechtlijnig pad zonder obstructies tussen bestaat. Als dit zo is, worden de tussenliggende knopen overgeslagen. Wanneer er toch een obstructie gevonden wordt, wordt er verder gewerkt vanuit de laatste knoop die zonder obstructie met de beginknoop kon verbonden worden(zie figuur 10.6).



(a) Hoekig pad (b) Na smoothing

Figuur 10.6: Een pad, voor en na smoothing

10.2 Planning

10.2.1 Hiërarchie van de planner

Acties uitvoeren

De agent kan enkele zeer elementaire operaties:

- Hij kan een stap vooruit zetten in een bepaalde richting, vooropgesteld dat deze hem niet in contact zal brengen met een muur of een ander niet doorlatend object. De grootte van deze stap wordt beperkt door de maximale stap die de agent kan zetten. Indien er bevolen wordt een stap te zetten die groter is dan deze limiet, wordt deze stap automatisch opgesplitst.
- Hij kan objecten oppakken en neerzetten. De agent kan enkel objecten optillen die als 'transporteerbaar' zijn aangegeven. Met de agent is bovendien een bepaalde kracht geassocieerd, die vergeleken wordt met het gewicht van het object dat hij wenst op te tillen. Hij zal enkel het object kunnen optillen indien hij over voldoende kracht beschikt.
- Hij kan een 'complexe operatie' uitvoeren op een object. Dit is een operatie die dat object zelf aanbiedt. Soms heeft deze operatie nog extra voorwaarden, zoals een ander object in bezit hebben (bijvoorbeeld om een slot te openen, zal het bezit van de juiste sleutel noodzakelijk zijn).

Doelen

Een niveau hoger dan de acties hebben we de doelen. Deze zijn nog steeds vrij eenvoudig en bestaan uit 3 soorten. Doelen kunnen zijn: ergens naartoe willen, een object opnemen of een complexe operatie uitvoeren op een object. Zoals vermeld kan een complexe operatie nog extra voorwaarden met zich meebrengen, zoals 'ingrediënten' die aanwezig moeten zijn of de nabijheid van materialen of werktuigen. Deze extra voorwaarden zullen zich dan vertalen in doelen die gerealiseerd moeten worden alvorens aan deze complexe operatie te beginnen.

De doelen worden in de agent omgezet naar een serie opeenvolgende elementaire acties. Bijvoorbeeld het doel 'een object opnemen' kan bestaan uit tien elementaire 'stap' acties en één 'pik op' actie. Naar een bepaalde locatie navigeren zal hier door de pahtfinding engine van een doel omgezet worden naar een sequentie van elementaire stappen.

De doelen bevatten ook een evaluatiefunctie om te kunnen controleren of ze geslaagd zijn uitgevoerd. Bij de bewegingsdoelen houdt dit in dat er gecontroleerd wordt of het doel nu binnen (arm-) bereik is gekomen, en bij de oppik acties wordt er gekeken of het object in het *inventory* zit. Voor de complexe operaties is dit iets moeilijker. Als de operatie voltooit zonder fouten te veroorzaken, wordt er vanuit gegaan dat ze succesvol is uitgevoerd. Om ervoor te zorgen dat er na eventuele fouten opnieuw kan gepland worden, moeten deze fouten ook vrij strikt gespecificeerd worden.

Commando's en taken

Op het hoogste niveau worden commando's en taken gebruikt. Dit is de input die de agent van buitenaf ontvangt. De agent tracht bij het interpreteren van deze commando's af te leiden welke doelen en sub-doelen hij moet realiseren om de taak tot een goed einde te brengen.

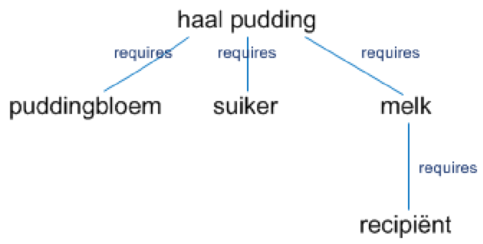
Een andere mogelijkheid om dit te doen is de agent rechtstreeks de doelen te geven die hij moet bereiken. Dit is eenvoudiger, aangezien er dan geen interpretatiestap noodzakelijk is. Bovendien kan van een doelstatus geëvalueerd worden of deze al bereikt is of niet.

10.2.2 Requirements planning

Er zijn verschillende mogelijkheden om de agent uit een commando een plan te laten opstellen. Een eerste optie is de agent te laten plannen door extensief gebruik te maken van *requirements*. Als we de agent bevelen pudding te halen, en pudding heeft bloem, suiker en melk nodig, dan weet hij dat hij deze dingen moet verzamelen voor hij pudding kan maken. Als melk bovendien een recipiënt nodig heeft, kan de agent de requirements op deze manier verder uitbreiden tot er een boom ontstaat (zie figuur 10.7).

De agent zal eerst de requirements op het laagste niveau vervullen, zodat deze als vervuld kunnen beschouwd worden. Als we de vervulde requirements verwijderen, wordt de boom die het probleem voorstelt minder diep. Als we dit blijven doen zal het hoofdplan uiteindelijk ook uitgevoerd kunnen worden met één triviale stap.

Als er ergens onderweg een probleem ontstaat, moet er op dat punt een alternatief gezocht worden, of moet het plan als onhaalbaar beschouwd worden.



Figuur 10.7: Requirements tree voor de actie 'haal pudding'

10.2.3 Planning door associatie

Een andere optie is de agent een plan te laten opstellen door te kijken naar de associaties tussen de objecten. We vragen de agent water te halen, maar hij merkt dat er in de wereld (nog) geen water objecten bestaan. Wanneer hij vervolgens de associaties van de objecten in zijn wereld nagaat, ontdekt hij dat met een 'kraan' 'water' geassocieerd wordt. Hij zal dan een operatie zoeken die hij kan uitvoeren op deze kraan, en die als resultaat water oplevert (bijvoorbeeld de kraan open draaien).

Als de operatie-informatie van alle objecten op een gestructureerde manier omschreven is, is het niet altijd noodzakelijk dat er nog een aparte lijst met associaties voorzien wordt, tenzij om eventuele verwarring met synoniemen op te lossen. Er kan meestal rechtstreeks in de operatie-informatie gezocht worden naar de juiste *return-types*. Een nadeel aan deze laatste aanpak is echter dat er verwarring kan ontstaan bij methodes die een algemeen object kunnen teruggeven. Dit is bijvoorbeeld het geval bij een container die een bepaald soort objecten kan bevatten. Hiervan is het return-type dan de superklase van deze objecten. Hier is natuurlijk niet impliciet duidelijk of dit om het juiste soort object zou gaan. Dit kan eventueel opgelost worden door de operatie-informatie niet rechtstreeks af te lezen van het object (bijvoorbeeld door middel van reflectie), maar het object dit ook zelf te laten aanbieden. De container zou dan deze informatie moeten aanpassen aan het object dat hij al of niet bevat. Het is dus essentieel dat de objecten zich correct gedragen en de juiste informatie aanbieden!

10.2.4 Planning via de resultaten

Een derde optie is een eenduidige beschrijving van het resultaat van de operaties opslaan in de intelligente objecten. Wanneer er dan een bepaald doel bereikt moet worden, kan er tussen deze resultaat-omschrijvingen gezocht worden naar het gewenste resultaat.

Een nadeel van deze laatste aanpak is de mogelijke ambiguïteit van de omschrijvingen. Om dit te vermijden zou er geopteerd moeten worden om een strikte taal te hanteren (bijvoorbeeld logica) om de resultaten te omschrijven.

10.2.5 Kiezen tussen meerdere kandidaten

Het is altijd mogelijk dat er meerdere kandidaat-operaties of objecten zijn om een bepaald doel te bereiken. Er zijn bijvoorbeeld meerdere deuren die geopend kunnen worden, of er zijn meerdere recipiënten waarin water kan vervoerd worden. Op dat moment zullen deze opties moeten gesorteerd worden op basis van relevantie. Mogelijke heuristieken zijn: bereikbaarheid, nabijheid, hoeveelheid extra requirements, ... Het berekenen van de nabijheid en de bereikbaarheid gebeurt door de pathfinding module.

10.2.6 Vertraagde uitvoering

Bij het opstellen van een plan is het soms noodzakelijk om een object dat nog niet bestaat voor te stellen door een *placeholder object*, bijvoorbeeld wanneer een object nog moet geassembleerd worden. Een placeholder object gebruiken heeft enkele nadelen. Er moet onder andere overal rekening gehouden worden met het feit dat objecten vervangen kunnen zijn door placeholders. Wanneer de creatie van het uiteindelijke object dan zou falen is dat niet altijd meteen duidelijk voor het bovenliggend plan.

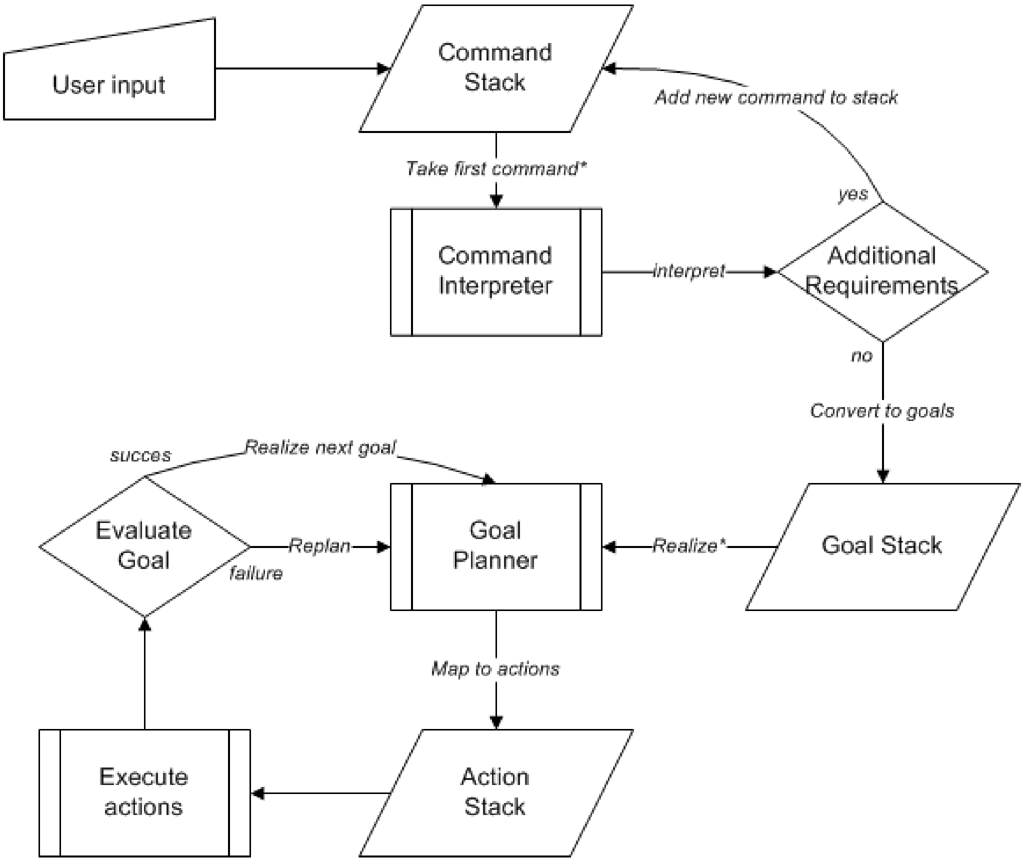
Bij het recursief requirements plannen (zie 10.2.2) hebben we dit probleem echter niet. In plaats van direct een plan op te stellen voor het gehele probleem, wordt er telkens geëvalueerd of een plan nog steeds haalbaar is. Als er nog requirements zijn die ingevuld moeten worden, wordt er eerst een plan opgesteld om deze in te vullen. Na elke uitwerking van een dergelijke requirement is er opnieuw een evaluatie die kijkt of het plan al haalbaar is geworden.

Deze re-evaluatie bestaat uit het opnieuw interpreteren van het commando uit de commandostack. Dit heeft ook het voordeel dat indien er iets wijzigt in de wereld, daar bij elke replanning automatisch rekening mee gehouden wordt. Hierdoor is het wel mogelijk dat de agent zal beginnen met dingen te doen die misschien niet direct verband lijken te houden met het hoofdplan, maar er enkel mee gelinkt zijn via de ketting van opeenvolgende requirements (zie figuur 10.8).

10.3 Een uitgewerkt voorbeeld

In deze sectie staat een planning voorbeeld uitgeschreven. Deze tekst wordt begeleid met de afbeeldingen in bijlage C.

Wanneer we de agent de opdracht 'fill glass with water' geven, wordt deze eerst op de commandostack gezet. We beschouwen de staat van de wereld zoals in figuur 9.1, met de deuren gesloten. Als de uitvoering aanvangt, nadat eventueel nog extra commando's toegevoegd zijn, wordt het eerste commando van de stack opgehaald. Dit commando wordt door de interpreter geanalyseerd. In dit specifieke geval zal eerst gekeken worden of het glas een beweegbaar object is, en indien dit zo is zal de agent trachten hiernaartoe te gaan



Figuur 10.8: Schematische voorstelling van de requirements processing

om het op te pakken. Dit gebeurt door de doelen 'ga naar glas' en 'pik glas op' op de goalstack te plaatsen.

Elk van deze twee doelen zal door de goal planner in elementaire operaties omgezet worden. In dit geval zal de goal planner merken dat het glas niet bereikbaar is, door het feit dat de deuren gesloten zijn. Hierdoor gaat de agent operaties uitvoeren op elementen die de pathfinding positief kunnen beïnvloeden. In deze wereld komt dat neer op deuren openen. Het commando 'open door' wordt op de commandostack geplaatst, helemaal vooraan. De goal stack wordt vervolgens leeggemaakt. De planning engine gaat verder door het nieuwe commando te interpreteren.

Er zal gekeken worden of er operaties zijn in de wereld die tot gevolg hebben dat deuren opengaan. In deze wereld zullen zich twee kandidaten aandienen: een druk op de knop in de eerste kamer, om de eerste deur te openen, en de deur in de tweede kamer, die aan de deur zelf te openen valt. Wanneer de eerste deur geopend is, wordt het commando van de stack gehaald en wordt er opnieuw verder gegaan met het commando 'fill glass with water' (zie bijlage C.1). Er zal gemerkt worden dat het glas nog steeds niet bereikbaar is, en er zal opnieuw geprobeerd worden om een deur te openen. Nadat deze laatste deur is geopend wordt het commando terug van de stack gehaald en wordt er terug overgegaan naar 'fill glass with water' (zie bijlage C.2).

Deze keer is het glas wel bereikbaar, en zal de pathfinding engine de agent tot bij het glas voeren. Het glas kan ook opgepikt worden, waardoor het in het bezit van de agent komt (zie bijlage C.3). Hierna volgt opnieuw een evaluatie van het eerste commando (zie 10.2.6), waarvan nu één requirement al vervuld is. De goal planner zet ditmaal de doelen 'ga naar kraan' op de stack, alsook de complexe operatie 'produceer', uit te voeren op het 'kraan'-object. Deze laatste operatie is door de planner gevonden via de resultaat-gedreven planning (zie 10.2.4). Wanneer deze actie uitgevoerd wordt, komt er een water object in de wereld, en dit is wat de agent wilt bereiken (zie bijlage C.4).

Nadat deze acties ook omgezet zijn in elementaire acties, en succesvol uitgevoerd, wordt het initiële commando een laatste maal geïnterpreteerd. Alle elementen zijn nu aanwezig om de actie in één stap uit te voeren. De goal planner zet het commando om in de complexe operatie 'vul', uit te voeren op het 'glas' object met als parameter 'water'.

Als deze operatie succesvol is uitgevoerd, wordt het commando 'fill glass with water' van de stack gehaald, en is de taak uitgevoerd.

Moest er onderweg gebleken zijn dat een van de objecten niet bereikbaar was, en er was geen alternatief (bijvoorbeeld een tweede glas), dan zou het plan onhaalbaar geacht worden, en niet verder uitgevoerd. Het commando wordt dan van de stack gehaald.

10.4 Discussie

De geconstrueerde agent heeft zoals verhoopt niet al te veel basiskennis nodig. Hij moet kunnen rondwandelen, objecten oppikken, neerzetten, en er een 'complexe operatie' op kunnen uitvoeren.

Om de complexe taken tot een goed einde te brengen wordt een planner gebruikt, in combinatie met een systeem dat de objecten in de wereld over hun operaties kan bevragen. Zolang de objecten en operaties op een correcte manier gedefiniëerd worden, zal de agent in staat zijn er taken mee uit te voeren.

Om te weten of een plan onmogelijk is om uit te voeren door een gebrek aan een bepaald item, zal de agent eerst exhaustief heel de wereld moeten onderzoeken om te kijken of het item zich ergens bevindt, of ergens kan gemaakt worden. Om diezelfde reden moet de agent ook zijn *knowledge base* trachten up to date houden elke keer er iets wijzigt in de wereld.

Een nadeel van het feit dat de pathfinding vooraf ingebakken zit in de agent, is dat de agent ook informatie nodig heeft over objecten die de pathfinding beïnvloeden. Zo zullen deuren een markering of opmerking (zie 10.2.4) nodig hebben die vertelt dat het openen ervan de pathfinding kan beïnvloeden. Het wordt al iets complexer bij objecten die een doorgang blokkeren. Indien we willen dat de agent hiermee overweg kan, moeten we een algoritme toevoegen dat berekent in welke richting we deze objecten willen verplaatsen. Dit is nodig om te voorkomen dat na verplaatsing een andere doorgang zou geblokkeerd worden.

Hoofdstuk 11

Conclusie en verder onderzoek

11.1 Conclusie

Virtuele werelden worden steeds complexer. Hierdoor wordt het voor intelligente agenten die ontworpen zijn met een klassieke aanpak steeds moeilijker om zich hierin te kunnen handhaven. Dit komt hoofdzakelijk door de steeds toenemende data-hoeveelheden en mogelijkheden van deze werelden. Er is dus nood aan krachtige agenten die zich kunnen aanpassen aan situaties die niet vooraf voorzien waren. Toch mogen deze agenten niet te complex worden.

Uit de besproken technologieën en de implementatie blijkt dat het mogelijk is om relatief eenvoudige agenten te ontwerpen die toch in staat zijn zich in diverse situaties te handhaven. Ook bij een veranderende wereld is het zelden nodig deze agenten opnieuw te compileren of in te laden. Een bespreking van de implementatie kan gevonden worden in sectie 10.4.

Deze voordelen zijn hoofdzakelijk mogelijk dankzij de verplaatsing van de belangrijke informatie naar de objecten voor welke ze relevant is, en de ondersteuning door scripting talen. Aangezien het in dat geval gaat om een gedecentraliseerd design, zal er veel minder snel een punt ontstaan dat overbelast is met verantwoordelijkheden.

11.2 Future work

Een interessante richting om verder te onderzoeken is hoe meerdere agenten van dit type zich zullen gedragen wanneer ze een coöperatieve taak moet oplossen. Hiervoor zal een soort communicatie tussen de agenten noodzakelijk zijn, en waarschijnlijk een uitwisseling van planning gegevens of doelstellingen [PMS+07] [Gar05] [VN04] [Mor04].

Bibliography

[ACT05] T. Abaci, J. Ciger, and D. Thalmann. Planning with Smart Objects. *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2005. 6.3

[BD02] Ben Board and Mike Ducker. Area navigation: Expanding the path-finding paradigm. In *Game Programming Gems 3*, pages 240–255. Charles River Media, Inc., 2002.

[BN04] Gerd Beuster and Roman Neruda. Configuring computational agents. In *Knowledge Grid and Grid Intelligence 2004*. Saint Mary's University, 2004.

[Bon85] Alain Bonnet. *Artificial Intelligence: Promise and Performance*. Prentice Hall, 1985.

[CS05] Jamie Cheng and Finnegan Southey. Implementing practical planning for game AI. In *Game Programming Gems 5*. Charles River Media, Inc., 2005. 4.2.2

[Daw08] Michael Dawe. Goal-oriented plan merging. In *Game Programming Gems 7*, pages 281–287. Charles River Media, Inc., 2008.

[DeL08] Robert Kirk DeLisle. Beyond A*: Ida* and fringe search. In *Game Programming Gems 7*, pages 289–294. Charles River Media, Inc., 2008.

[Gar05] Diego Garces. Achieving coordination with autonomous NPCs. In *Game Programming Gems 6*, pages 223–234. Charles River Media, Inc., 2005. 11.2

[Han03] John Hancock. An object-oriented utility-based decision architecture. In *Game Programming Gems 4*, pages 337–344. Charles River Media, Inc., 2003.

[JWL05] Pieter Jorissen, Maarten Wijnants, and Wim Lamotte. Dynamic interactions in physically realistic collaborative virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 2005. 6.4, 7.1.1, 9.2

[KT98] M. Kallmann and D. Thalmann. Modeling Objects for Interaction Tasks. In *Eurographics Workshop on Animation and Simulation, 1998*, 1998. (document), 6.1

[KT99] Marcelo Kallmann and Daniel Thalmann. Direct 3d interaction with smart objects. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, 1999. 6.2

[LC07] Jean-Luc Lugrin and Marc Cavazza. Making sense of virtual environments: action representation, grounding and common sense. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*. ACM, 2007.

[Lev95] L. Levison. Connecting planning and acting: Towards an architecture for object specific reasoning, 1995.

[MG05] Matthew Titelbaum Mario Grimani. Beyond A*. In *Game Programming Gems 5*. Charles River Media, Inc., 2005. 5.1.1

[Mor04] A.S. Morse. Coordination of groups of mobile autonomous agents. Technical report, Yale University, 2004. 11.2

[PA05] Hugo Pinto and Luis Otavio Alvares. A goal-oriented unreal bot: Building a game agent with goal-oriented behavior and simple personality using extended behavior networks. In *Game Programming Gems 6*, pages 259–271. Charles River Media, Inc., 2005.

[PD05] Armand Prieditis and Mukesh Dalal. Applying model-based decision-making methods to games: Applying the locust ai engine to Quake III. In *Game Programming Gems 6*, pages 211–222. Charles River Media, Inc., 2005. (document), 3.2, 4.4, 4.2, 4.4

[Piv03] Karen Pivazyan. NPC decision making: Dealing with randomness. In *Game Programming Gems 4*, pages 325–335. Charles River Media, Inc., 2003.

[PMS+07] Eric Platon, Marco Mamei, Nicolas Sabouret, Shinichi Honiden, and H. Van Parunak. Mechanisms for environments in multi-agent systems: Survey and opportunities. *Autonomous Agents and Multi-Agent Systems*, 2007. 11.2

[Rab00a] Steve Rabin. A* aestethic optimizations. In *Game Programming Gems 1*, pages 264–271. Charles River Media, Inc., 2000.

[Rab00b] Steve Rabin. A* speed optimizations. In *Game Programming Gems 1*, pages 272–287. Charles River Media, Inc., 2000.

[Rey99] Craig Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, 1999. (document), 5.2, 5.1, 5.2, 5.3, 5.4

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995. (document), 2, 3.1, 3.3, 4.1, 4.2.1, 4.3, 4.5, 4.6, 4.7, A

[Sno00] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. In *Game Programming Gems 1*, pages 288–304. Charles River Media, Inc., 2000.

[SSF08] Martin Scheffler, Jan P. Springer, and Bernd Froehlich. Object-capability security in virtual environments. In *IEEE Virtual Reality 2008*, page 51. Bauhaus-Universität Weimar, 2008.

[Sto00] Bryan Stout. The basics of A* for path planning. In *Game Programming Gems 1*, pages 254–262. Charles River Media, Inc., 2000. 4.3.5

[Tom05] Marco Tombesi. Advanced pathfinding with minimal replanning cost: Dynamic A star (D*). In *Game Programming Gems 5*. Charles River Media, Inc., 2005. 5.1.2

[UB08] Iskander Umarov and Anatoli Belialev. Managing AI algorithmic complexity: Generic programming approach. In *Game Programming Gems 7*, pages 229–247. Charles River Media, Inc., 2008.

[vdS02] William van der Sterren. Tactical path-finding with A*. In *Game Programming Gems 3*, pages 294–306. Charles River Media, Inc., 2002. 5.1.3

[VN04] Roman Vaculín and Roman Neruda. Autonomous behavior of computational agents. Technical report, Charles University (Prague), 2004. 11.2

[WC02] Stephen White and Christopher Christensen. A fast approach to navigation meshes. In *Game Programming Gems 3*, pages 307–320. Charles River Media, Inc., 2002.

[You02] Thomas Young. Choosing a relationship between path-finding and collision. In *Game Programming Gems 3*, pages 321–332. Charles River Media, Inc., 2002.

Website references

[1] AI toolkits. http://www.gameai.com/toolkits.html. 7.2

[2] Applications of artificial intelligence. http://en.wikipedia.org/wiki/Applications_of_artificial_intelligence.

[3] Artificial intelligence. http://en.wikipedia.org/wiki/Artificial_intelligence.

[4] Connectionism. http://en.wikipedia.org/wiki/Connectionism.

[5] Cybernetics. http://en.wikipedia.org/wiki/Cybernetics.

[6] Dynamic programming. http://en.wikipedia.org/wiki/Dynamic_programming.

[7] Emergence. http://en.wikipedia.org/wiki/Emergence.

[8] Hebbian learning. http://en.wikipedia.org/wiki/Hebbian_learning.

[9] History of AI. http://en.wikipedia.org/wiki/History_of_artificial_intelligence.

[10] Hopfield network. http://en.wikipedia.org/wiki/Hopfield_network.

[11] Intelligent agents. http://en.wikipedia.org/wiki/Intelligent_agents.

[12] Lua:about. http://www.lua.org/about.html. 7.1.2

[13] Runtime performance comparison. http://www.shudo.net/jit/perf/. 8.4

[14] Python:about. http://www.python.org/about/. 7.1.3

[15] AI in fiction. http://en.wikipedia.org/wiki/Artificial_intelligence_in_fiction. 1

[16] Visual studio 2005. http://msdn.microsoft.com/nl-nl/vs2005/default(en-us).aspx. 8.4, 9.3

[17] Artificial neural networks. http://en.wikipedia.org/wiki/Artificial_neural_networks.

[18] Evolutionary algorithm. http://en.wikipedia.org/wiki/Evolutionary_algorithm.

[19] A* search algorithm. http://en.wikipedia.org/wiki/A*. 4.3.5, 4.3.5

[20] Artificial intelligence portal. http://en.wikipedia.org/wiki/Portal:Artificial_intelligence. 1.2

[21] Artificial neural networks. http://en.wikipedia.org/wiki/Artificial_neural_network. 3.3(a), 3.3.4

[22] Lua (programming language). http://en.wikipedia.org/wiki/Lua_(programming_language). 7.1.2

[23] Python (programming language). http://en.wikipedia.org/wiki/Python_(programming_language). 7.1.3

[24] Python software. http://en.wikipedia.org/wiki/Python_software. 7.1.3

[25] Scripting language. http://en.wikipedia.org/wiki/Scripting. 7.1

[26] Self organizing maps. http://en.wikipedia.org/wiki/Self-organizing_maps.

Deel IV

Appendix

Bijlage A

Logische afleidingsregels

Hierin worden kort enkele logische afleidingsregels opgesomd [RN95].

A.1 Afleidingsregels

A.1.1 Modus Ponens

De modus ponens wordt ook de *implicatie-eliminatie* genoemd. Als een implicatie waar is, en de premisse van de implicatie is ook waar, dan is de conclusie ook waar.

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta} \tag{A.1}$$

A.1.2 En-Eliminatie

Als een conjunctie waar is, kan je daaruit afleiden dat om het even welk element ervan waar is.

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i} \tag{A.2}$$

A.1.3 En-Introductie

Als een lijst zinnen waar is, is ook hun conjunctie waar.

$$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n} \tag{A.3}$$

A.1.4 Of-Introductie

Als een zin waar is, is de disjunctie van die zin met om het even wat nog steeds waar.

$$\frac{\alpha_i}{\alpha_i \vee \alpha_1 \vee ... \vee \alpha_n} \tag{A.4}$$

A.1.5 Eliminatie van dubbele negatie

Als de dubbele negatie van een zin waar is, is de zin zelf ook waar.

$$\frac{\neg\neg\alpha}{\alpha} \tag{A.5}$$

A.1.6 Eenheidsoplossing

Wanneer een disjunctie waar is, en één van de delen ervan is onwaar, dan is het andere deel waar.

$$\frac{\alpha \vee \beta,\ \neg\beta}{\alpha} \tag{A.6}$$

A.1.7 Oplossing

Wanneer er twee disjuncties waar zijn, waarvan één met β en één met $\neg\beta$, weten we dat één van de overige disjuncten waar moet zijn, omdat β niet zowel waar als niet waar kan zijn.

$$\frac{\alpha \vee \beta,\ \neg\beta \vee \gamma}{\alpha \vee \gamma} \tag{A.7}$$

Hieruit volgt ook de transitiviteit van de implicatie ($\Rightarrow$).

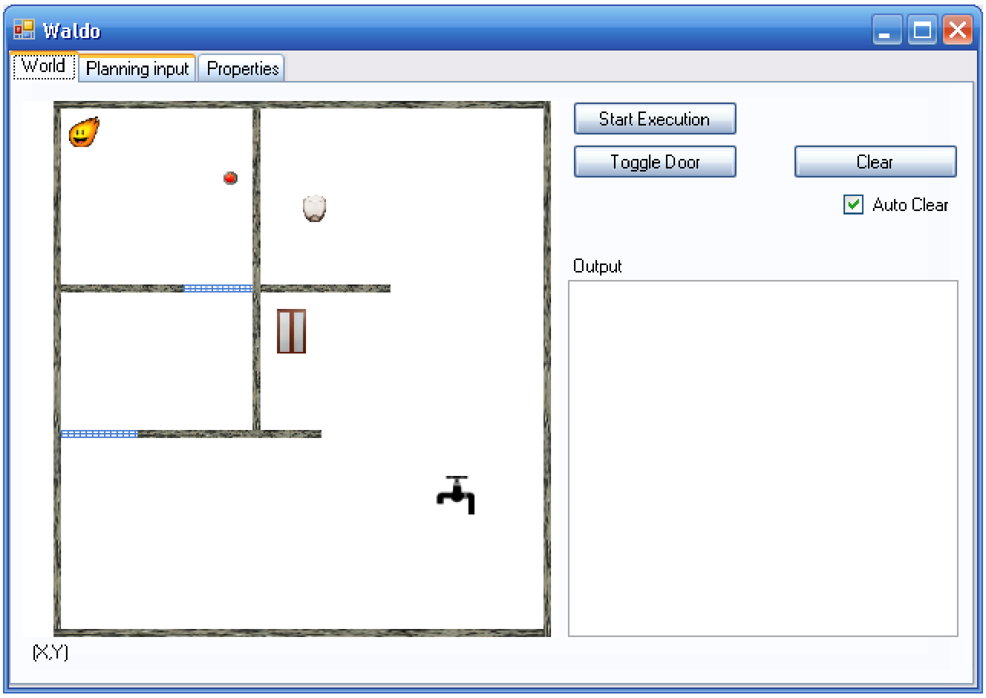
$$\frac{\neg\alpha \Rightarrow \beta,\ \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma} \tag{A.8}$$

Bijlage B

Screenshots van de testomgeving

In deze bijlage staan enkele screenshots van de testapplicatie (zie 9.3).

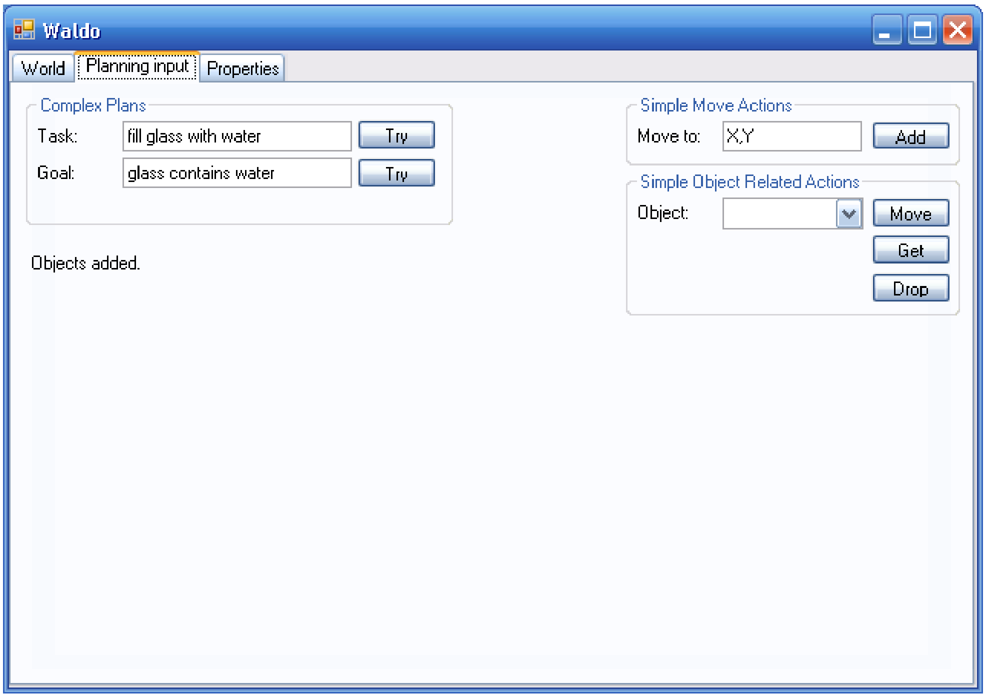
B.1 Output



Figuur B.1: De output tab

Op deze tab is het mogelijk de agent in actie te zien in de 2D wereld. De tekstuele output die van de agent zelf komt, wordt weergegeven in het output-veld. De functie 'auto clear' kan afgezet worden om een duidelijker beeld te krijgen van de voortgang tijdens het debuggen van de pathfinding. Het is tevens mogelijk objecten in dit scherm te selecteren door middel van picking. Deze worden dan automatisch ingeladen op het tabblad voor object-eigenschappen (zie B.3).

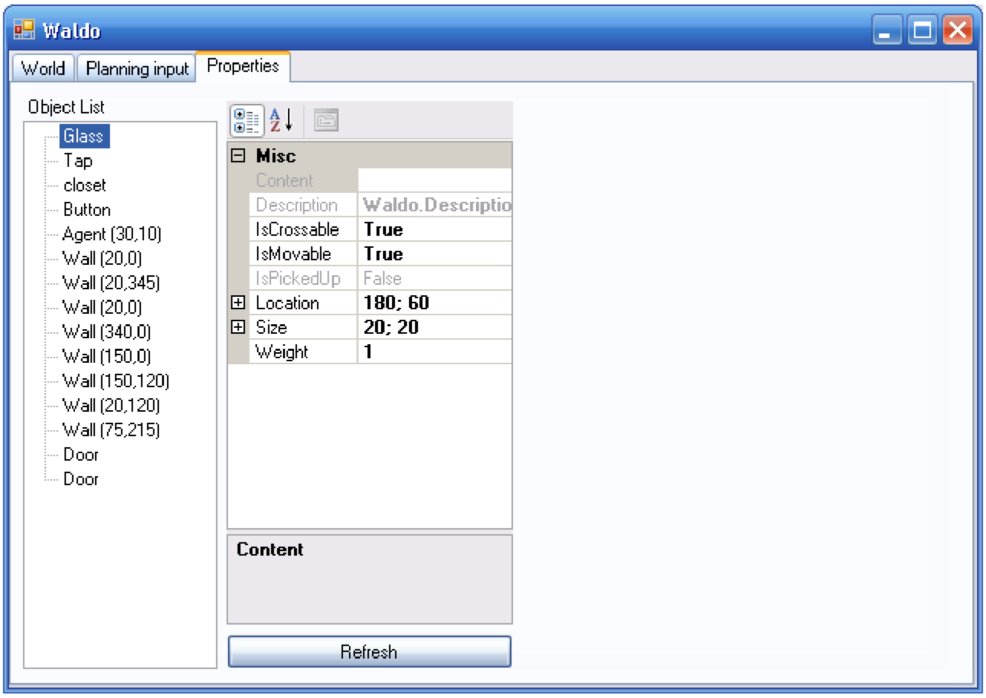
B.2 Input



Figuur B.2: Het tabblad voor planning invoer

Op deze tab kan men de opdrachten ingeven die de agent moet uitvoeren. Voor eenvoudige basisopdrachten kan de rechterkolom gebruik worden. Voor complexere opdrachten worden de invoervelden in de linkerkolom gebruikt. De invoer daar kan op twee manieren gegeven worden: via opdrachten (*tasks*) of via het doel dat de agent moet bereiken (*goals*) (zie 10.2.1).

B.3 Objecten



Figuur B.3: Het tabblad waar objecten aangepast kunnen worden

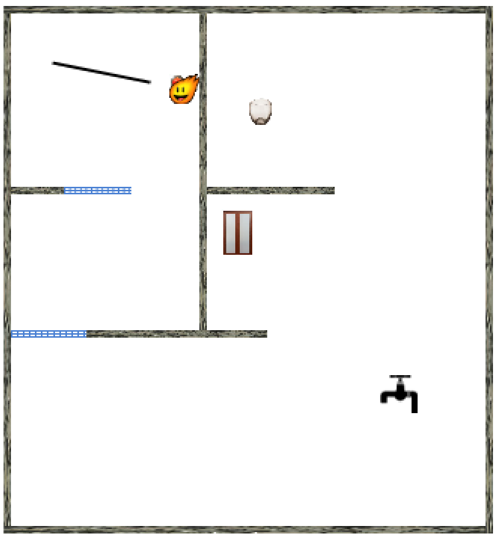
Op dit tabblad kunnen de objecteigenschappen gewijzigd worden. Wanneer links een object wordt geselecteerd, worden rechts de eigenschappen ingeladen. Hier kan ook handig gecontroleerd worden of een bepaalde opdracht van de agent het beoogde effect heeft gehad op het desbetreffende object.

Bijlage C

Screenshots bij het voorbeeld

In deze bijlage staan enkele screenshots ter illustratie van het voorbeeld in sectie 10.3. Het pad dat da agent gevolgd heeft om tot op zijn huidige locatie te komen is telkens aangeduid met een donkere lijn.

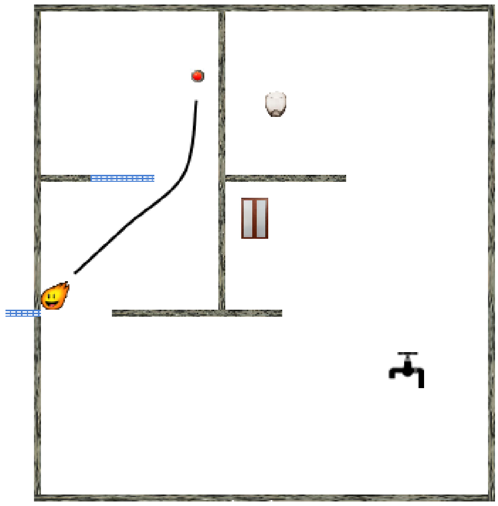
C.1 Eerste deur



Figuur C.1: Deur met knop

In deze figuur is de agent tot aan de knop gewandeld. Deze gebruikt hij om de eerste deur te openen.

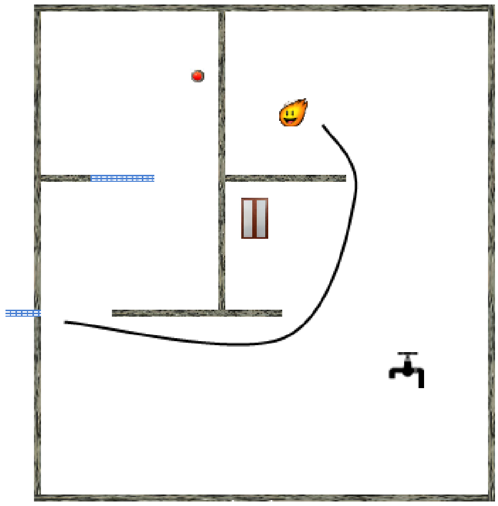
C.2 Tweede deur



Figuur C.2: Tweede deur

Hier is de agent tot aan de tweede deur gewandeld, en heeft deze geopend.

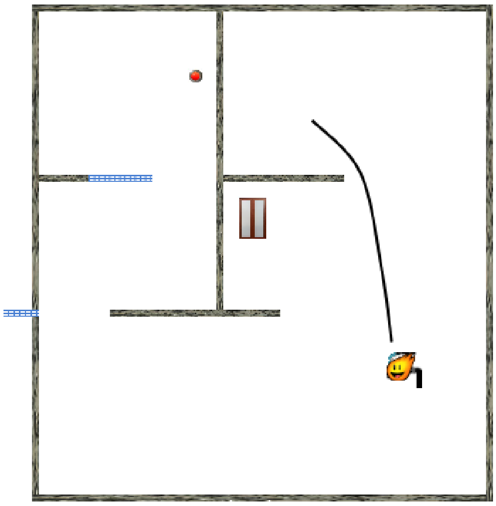
C.3 Glas



Figuur C.3: Het glas oppikken

De agent beweegt zich tot aan het glas, en pikt het op.

C.4 Kraan



Figuur C.4: Bij de kraan

De agent beweegt tot aan de kraan. Met deze kraan produceert hij water, en hiermee vult hij het glas.