Utveckling Av Artificiell Intelligens Till Datorspel

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH Ingenjörshögskolan vid Campus Helsingborg
Datateknik

Examensarbete:
Johan Källberg

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2011

Sammanfattning

Fokus i examensarbetet ligger på att undersöka grundläggande tekniker för att strukturera beslutsfattande och pathfinding för artificiell intelligens till datorspel. Vidare sammanfattas redan implementerade exempel på dessa strukturer och jämförs med varandra för att välja passande lösningar att integrera för att användas tillsammans med spelmotorn Unity vid utvecklingen av ett specifikt spel kallat Unknown.

För beslutsfattande valdes Behave, ett plugin med smidig hantering av beteendeträd som är lätt att överskåda och modifiera. Beteendeträd är en teknik som hanterar beslut i flera nivåer i en hierarkisk struktur som hanterar handlingar separat från beslut och är acyklisk i övergångar mellan beslut.

För pathfinding valdes A* pathfinding ett flexibelt plugin med stöd för många olika tekniker för att representera spelmiljön i en sökbar graf. Grafen genomsöks med algoritmen A* med olika valbara alternativ för hantering av heuristik och utjämning av den funna vägens kurvor. För att representera spelmiljön valdes ett rutnät som är den enklaste av de undersökta teknikerna. Anledningen till att den tekniken valdes var att den var så enkel att generera och hantera med A* pathfinding.

Med Behave och A* pathfinding skapades en tidig version av en agent som kan navigera runt hinder till olika punkter i spelmiljön, förfölja spelaren och utföra specifika handlingar vid kontakt med spelaren.

Nyckelord: Artificiell intelligens, Agent, Beslutsfattande,
Pathfinding, Unity

Abstract

Focus in this examination project is put on researching basic techniques for structuring decision-making and pathfinding for the artificial intelligence in computer games. Already implemented examples of these structures are summarized and compared to each other to choose fitting solutions to use together with the game engine Unity for the development of a specific game called Unkown.

For decision-making a plugin called Behave was chosen. Behave implements behavior trees in a way that makes them easy to create, modify and review. Behavior trees handle decisions in several levels in a hierarchical structure which handles actions separate from decisions and is acyclic in the transitions between decisions.

A* pathfinding was the plugin chosen for pathfinding purposes. It is a flexible plugin with support for several different methods for representing the game environment in a searchable graph. The graph is searched with the A* algorithm with different alternativs available for heuristics and smoothening of the path. The method chosen for representing the environment was a grid due to the simplicity and ease of generating it with A* pathfinding.

With Behave and A* pathfinding an early version of an agent was created with the ability to navigate past obstacles to different places in the environment, to follow the player and execute specific actions upon contact with the player.

Keywords: Artificial intelligence, Agent, Decision-making, Pathfinding, Unity

Förord

Detta examensarbete är skrivet under våren och sommaren 2011. Jag påbörjade arbetet med enbart ett fåtal månaders erfarenhet av spelutveckling och helt utan erfarenhet av artificiell intelligens men med ett stort intresse för områdena. Arbetet har varit väldigt utmanande då det är ett så pass nytt och stort område för mig men under arbetets gång har intresset bara växt och jag vill bara veta mer ju mer jag lär mig. Jag känner mig starkt motiverad att fortsätta utvecklingen av spelets artificiella intelligens och driva spelets utveckling framåt tills det är färdigt även efter att examensarbetet är slut.

Innehållsförteckning

1 Inledning

1.1 Bakgrund

Projektet är framtaget som en del i utvecklingen av ett spel som jag och Viktor Högqvist-Nilsson, en vän till mig, börjat utveckla förra året. Vi planerar att eventuellt starta företag inför senare lansering. Arbetsnamnet på spelet är Unknown. I projektet har jag huvudansvar för planering av arbete och spelets upplägg. Jag arbetar också som programmerare på spelet. Viktor arbetar med den estetiska biten i spelet som till exempel modellering, animering, arbete med texturer och andra visuella effekter. Vi samarbetar för att integrera dessa bitar till bästa möjliga produkt.

Spelet kallar vi Unknown och går ut på att spelaren skall överleva i en skräckinjagande miljö. För att ta sig vidare i spelet behöver spelaren lösa olika pussel för att komma åt nya platser i spelmiljön och för att föra spelets handling framåt. Handlingen kommer också att innefatta ett flertal mysterier och genom att utforska spelmiljön kan spelaren få mer information om vad som händer i spelvärlden. Inspiration till spelet har hämtats från många olika spel som vi spelat. Exempelvis är de populära skräckspelen *Silent hill*, *Fatal frame* och *Amnesia: The dark decent* inspirationskällor inom samma genre med spelelement liknande de vi planerar att skapa.

Än så länge har mitt fokus varit på att utveckla en rörelsekontroll för att smidigt kunna styra spelaren i en 3d miljö. Under mitt arbete med rörelsekontrollen har jag lärt mig vid att arbeta med spelmotorn Unity och lärt mig programmera i både C# och JavaScript. I processen med att utveckla ett spel finns många intressanta och avancerade områden där vi saknar kunskap och mycket tid används till att utbilda oss vidare. Den artificiella intelligensen hos de datorstyrda agenterna i spelet är en av de större bitarna i utvecklandet av ett spel och det är ett område jag har stort intresse för vilket är anledning till att jag valde att ha det som mitt examensarbete.

1.2 Syfte

Syftet med examensarbetet är att identifiera en lämplig struktur för beslutsfattande och en grund för målinriktad styrning anpassat för den artificiella intelligensen i Unknown. Av andra implementerade exempel på dessa problem undersöks för att välja ut passande lösningar på problemet. Valda lösningar ska integreras för användning till utveckling i Unknown eller alternativt implementeras.

1.2.1 Frågeställningar

Frågeställningar att besvara under projektet:

- Vad finns det för olika tekniker för artificiell intelligens till ett spel i en 3d miljö?
- Hur ser en bra struktur för artificiell intelligens ut när den är anpassad till Unity motorn?
- Vilka av dessa metoder är bäst lämpade för Unknown?
- Är det möjligt återanvända kod till flera olika agenter?

1.3 Målsättning

Jag har som mål att skaffa mig grundläggande färdigheter inom utveckling av artificiell intelligens till spel och att systemen som jag använder kommer vara smidiga att använda och väl anpassade för att bygga ut med flera funktioner. När det gäller pathfinding för agenterna har jag som målsättning att ett fungerande system för beräkning av riktningen agenten skall röra sig skall vara redo att användas i Unknown. Jag har förhoppning om att en första version av artificiell intelligens till en agent skall vara implementerad med lösningarna jag valt att använda.

1.4 Metod

1.4.1 Undersökning

Undersökningen innefattar en litteraturstudie baserad främst på tryckta källor men även ett antalwebbplatser. Litteraturstudien utfördes för att kunna ta ett informerat beslut om vilka strukturer som är lämpliga för beslutsfattande och pathfinding undersöks populära tekniker för de nämnda områdena i syfte att ge förståelse för hur teknikerna fungerar och vilka olika för och nackdelar de har.

Vidare utfördes efterforskningar för att hitta olika varianter på dessa lösningar och plugin. För att hitta passande plugin sökte jag främst igenom sidor knutna till gemenskapen som finns kring Unity i form av forum, wiki’s och Unitys egen ”Asset Store”. Troligtvis går det att hitta samtliga publika plugin kompatibla med Unity via dessa sidor. För att kompletera sökningen gjorde jag också en generell sökning efter passande plugin. [28]

1.4.2 Undersökning och testning av plugin

Utförandet av undersökningen har varierat på grund av ett flertal faktorer som skiljer sig från plugin till plugin. Exempel på skillnader är omfattning av funktionalitet, hur väl dokumentationen är utförd, om det finns exempel att granska, och hur komplicerat det är att skapa egna prototyper. Gemensamt för

alla undersökningar är att fokus ligger på att identifiera varje plugins grundläggande egenskaper och erbjudna funktionalitet.

På grund av skillnaderna i undersökningarna mellan olika plugin beskrivs varje undersökning enskilt tillsammans med beskrivningen av undersökt plugin i kapitel 3 med underkapitel.

1.4.3 Utveckling

Utvecklingen av en egen agent till Unknown gjordes med hjälp av de plugin som valts ut och programmeringsspråket C# med Unity's Aplication Programing Interface (API). Vid denna utveckling återanvändes kod som använts vid undersökning och testning av valda plugin.

1.4.4 Utvärdering

När jag jämför min ursprungliga plan för arbetet med det utförda arbetet finns det en stor skillnad. Skillnaden beror på en missuppfattning jag hade angående omfattningen av redan implementerade plugin som var tillgängliga för mig att använda. Nära all planerad tid för utveckling av egna implementationer för beslutsfattande och pathfinding lades istället på undersökning och testning av färdiga implementerade plugin. Tiden från utvecklingen som inte tillbringades på testning och undersökning var dock tillräcklig för att med hjälp av de implementerade lösningarna skapa en första version av en agent.

I övrigt fungerade mitt planerade upplägg ganska bra. Den avsatta tiden för litteratursökning kunde hållas och undersökningen var viktig för att ge mig grundkunskaper om artificiell intelligens. Den tidiga litteraturstudien hjälpte mig också att hitta de olika plugins jag undersökte och var väsentlig för att mitt val av plugin skulle vara informerat. Tiden som krävdes för att skriva rapporten var också ungefär vad jag uppskattat.

Med tidigare fokus på redan implementerade plugins hade jag möjligtvis haft bättre fokusering på de tekniker som fanns implementerade i min litteraturstudie samtidigt som jag haft möjlighet att ta mig längre i utvecklingen.

1.4.5 Avgränsningar

Arbetet är avgränsat från utgifter på mjukvara. Vidare begränsades mjukvaran till program kompatibla med Windows. De program och plugin som undersöktes för användning vid utveckling av Unknown krävdes vara kompatibla med Unity och C#.

1.4.6 Källkritik

Många av de tryckta källorna jag hade tillgång till var relativt gamla med tanke på att artificiell intelligens är ett område som utvecklas i hög takt. Tyvärr var nyare böcker svåra att få tag på i Sverige. Att de tryckta källorna använts för att informera mig om grunderna inom artificiell intelligens minskar konsekvensen av att källorna är så pass gamla. Ett exempel är A* algoritmen beskriven i kapitel 2.3.4 som är grundläggande inom pathfinding men som det gjorts anpassningar på som ökar effekten. Det finns flera anpassningar på A* men inga bättre alternativ som skiljer sig i grunderna.

Jag har också flera internetkällor som jag vid refererar till. När jag baserar text i examensarbetet på dessa källor så utvärderar jag trovärdigheten i källorna och använder mig i stor utsträckning av flera källor till samma information. Exempelvis är Alex J. Champandard, som jag låtit stå som ensam källa i arbetet ansvarig för en webbplats om artificiell intelligens inom spel med många artiklar från olika personer som jobbar inom området och med ett tillhörande forum.

Min användning av internetkällorna skiljer sig från det övriga arbetet där jag skriver om plugin som jag undersökt för att kunna välja ut de jag tänkt använda i utvecklingen av Unknown. Jag skriver om dessa plugin främst i kapitel 3. Här baserar jag texten på mina egna undersökningar, tester och utvärderingar av de plugin jag undersöker och referenserna pekar ut platsen som pluginet finns tillgänglig på. I kapitel 3.1.2 refererar jag exempelvis till ett forum inlägg av användare 1r0nM0nkey, texten är här alltså baserad på en utvärdering jag har utfört på programmet som 1r0nM0nkey skrivit.

1.5 Utvecklingsmiljö

1.5.1 Valet av spelmotor

Spelet utvecklas med Unity, en kommersiell spelmotor med en gratis version. Andra populära spelmotorer med gratis versioner och i vår prisklass innefattar exempelvis Unreal Development Kit (UDK) [25] och Ogre [22].

UDK och Ogre var starka kandidater för utvecklingen av Unknown. Alla tre spelmotorer har gratis versioner vilket var viktigt för att vi skulle kunna testa dem innan vi spenderar tusentals kronor. Samtliga har också en stor gemenskap av användare. Det är en stor positiv faktor då det finns mycket handledningsmaterial, många personer att ta hjälp av vid problem och dessutom många tillgångar som kan användas i egna spel och plugin som förbättrar olika delar av utvecklingsprocessen av egna spel. Funktionaliteten i spelmotorer är väldigt omfattande och tar tid att sätta sig in i och vi har därför inte gjort en full jämförelse mellan olika populära spelmotorer. Efter en första anblick av dessa tre kandidater ansåg vi Unity som mest tilltalande tack vare

hur lättöverskådligt det är och hur intuitivt det är att arbeta med det. Exempel på funktioner som ger detta intryck är att alla delar av ett spelobjekt, oberoende av typen av spelobjekt, finns samlade i en och samma vy varifrån enskilda delars egenskaper kan modifieras.

Av de tre kandidaterna valde vi att börja med att lära oss använda Unity. Det var till en början tänkt att lära oss mer om alla tre spelmotorer men på grund av att det tog så lång tid att lära sig arbetssättet och de olika funktionerna som finns tillgängliga i Unity valde vi att arbeta vidare med Unity när vi kände oss säkra på att vi kunde använda Unity för att utveckla Unknown.

1.5.2 Unity

Vid spelutveckling fungerar Unity som den grund som spelet byggs upp på. Många tillgångar som exempelvis 3d modeller, bilder, animationer och ljudfiler skapar man utanför Unity. Därefter kan dessa filer importeras till Unity där de kan hanteras på olika sätt beroende på vad det är för tillgång. Det finns vissa redskap för att skapa specifika tillgångar som exempelvis ljuskällor och terräng som utgör marken i spelmiljön men det går inte att utveckla spel enbart med Unity. När man hanterar olika objekt i Unity görs det i ett projekt som innefattar alla filer och tillgångar som används i spelet. I projektet finns det en eller flera scener som representerar en nivå i spelet. I en scen är alla objekt utplacerade för att bilda en spelmiljö. Här är alla objekt organiserade i hierarkier av objekt, till exempel kan alla ljuskällor i en scen grupperas under ett objekt. Det finns också möjligheten att spara hierarkier eller grupper olika objekt och tillgångar i projektet som så kallade ”prefabs” i Unity. Exempelvis kan ett prefab som hanterar en dörr bestå av en 3d modell för dörren och handtaget med ett tillhörande material som består av texturer och shaders för att ge dörren rätt yta. Dörren behöver också en ”collider” som gör att den fungerar som ett fast objekt, animationer som hanterar att handtaget trycks ner och att dörren öppnas och en ljudfil som spelas upp när dörren öppnas. Dessutom behövs ett script som hanterar funktionaliteten och tillåter att spelaren interagerar med objektet. [29]

Programmering med Unity

Programmeringen som görs vid spelutveckling med Unity är främst script. Det innebär att det är Unity som tolkar och exekverar koden.

Unity stödjer programmeringsspråken C#, JavaScript, och Boo. Jag har prövat på att använda mig av både JavaScript och C#. I Unity har det ingen påverkan på prestandan vilket språk som används och språken använder samma API från Unity. Unknown utvecklas med C#.

De script med tillhörande klasser som skapas för användning i Unity ärver alla från en basklass i Unity och använder Unity motorns namespace. På så vis får man tillgång till Unity's API. Mycket arbete utförs också genom att bygga funktionalitet i funktioner som ärvs från basklassen då dessa anropas vid specifika tillfällen som exempelvis varje frame, när spelaren klickar med musen eller när ett spelobjekt med en fysikkomponent kolliderar med ett objekt.

Vid arbete med scriptfilerna i Unity kan scriptfilerna enkelt dras och placeras på det spelobjekt scriptet ska påverka. Dessutom exponeras publika variabler vilka kan ändras direkt i Unity. Koden kompileras automatiskt när Unity används med ett "projekt" där scriptfilen ingår. Är scriptfilen bifogad till ett objekt i spelet kan spelet testköras direkt i Unity och effekten av koden visualiseras. För debugging finns möjligheten att skicka meddelanden till en konsol och rita ut vektorer i Unitys utvecklingsmiljö när spelet testkörs. [29]

Mer information om Unity finns deras hemsida unity3d.com[27].

1.5.3 C#

C# är ett programmeringsspråk designat för att vara ett lätt använt, modernt, objekt orienterat språk som inte är begränsat i sina användningsområden. Personligen anser jag att det gick väldigt lätt att lära sig programmera i C# med min tidigare erfarenhet av att programmera i språken Java och C++.

Jag har sammanfattat följande lista på några av de faktorer som är som kännetecknande för C#.

- Det har inga globala variabler eller funktioner i C# utan alla metoder och variabler måste placeras i en klass.
- Språket implementerar stark typning, vilket innebär begränsningar vid användning av kombinationer av olika typer av variabler.
- C# har efter version 4.0 en dynamisk datatyp som endast kontrolleras i runtime.
- Språket och dess implementationer har också gränsvärdeskontroll vilket kontrollerar att alla index och värden är inom utsatta gränser.
- Automatisk skräpsamling är en annan funktion som C# har stöd för, det är en funktion som automatiskt frigör minne som tas upp av objekt som inte kommer att användas längre.
- C# tillåter endast ärvning från en klass, det är däremot tillåtet att implementera ett godtyckligt antal interface.

[21]

1.5.4 System Development Kit

Det System Development Kit (SDK) jag använder mig av för programmering är Monodevelop. Monodevelop har bra stöd för C# och Unitys API med praktiska funktioner som "code completion" och "code formating". Dessutom kan Unity ställas in för att från Unity öppna scripts direkt i Monodevelop. Det är en praktisk funktion som används till exempel vid felmeddelanden från Unity om ett specifikt script, då används Unity för att från felmeddelandet direkt öppna scriptfilen i Monodevelop och markera rätt rad.

2 Litteraturstudie

2.1 Generella tekniker och tips om artificiell intelligens

2.1.1 Tips

Följande tips är hämtade från Steve Rabins *AI game programming wisdom*. [5] Jag har valt att ta med dem då jag uppfattar dem som bra att ha i åtanke under arbete med utvecklingen oberoende av vilken struktur man använder.

- **Håll det enkelt**

Bygg system av enkla delar som tillåter systemet att utföra komplexa uppgifter.

- **Lägg intelligens i världen och inte bara i agenten**

När en agent ska interagera med objekt i sin omvärld är det bra praxis att låta varje objekt känna till vad som kan utföras på det istället för att agenten har en lång lista med alla objekt och de handlingar som kan utföras på dem.

- **Ge varje uppgift en timeout och en reservrutin**

En av de brister hos olika agenter som lättast upptäcks av spelaren är när ett fel upprepas, exempelvis när en agent springer in i en vägg och inte kommer därifrån.

- **Låt inte agenter störa historieberättande händelser**

Det är lätt att glömma bort att extra kontroll över agenterna behövs när spelaren behöver ha sin uppmärksamhet på annat håll. Till exempel när historien i spelet utspelas ska en agent inte avbryta den händelsen.

- **Låt agenterna vara medvetna om spelvärdens globala tillstånd**

Ett enkelt sätt att få agenterna att verka smartare är att låta agenterna vara medvetna om händelser i spelet och låta dessa händelser påverka agentens beteende. Ett enkelt exempel är att en agent genom dialog med spelaren uttrycker sin åsikt om en handling spelaren utfört.

- **Skapa variation genom data och inte genom kod**

Att utforma koden på ett sätt som tillåter att beteenden kan anpassas genom att ändra variabler sparar mycket tid vid skapandet av variationer för olika beteenden. Det blir mindre kod att utveckla och därmed blir koden lättare att underhålla och organisera.

2.1.2 Fuzzy logic

Fuzzy logic är en teknik som används i flera olika områden inom artificiell intelligens. Tekniken använder en numerisk skala för att representera grader av gemenskap i ett antal olika grupper. Det medför att logiken som används kan beskrivas med generella termer som till exempel nära, långt borta, mycket, lite, ganska med flera. De generella termerna leder till att logiken blir mer subtilt och rikare i resonemanget än i traditionell boolsk logik. [1, 5]

2.1.3 Bayesiska nätverk

Bayesiska nätverk är riktade acykliska grafer modellerade efter relationerna mellan olika fenomen. Dessa grafer används i kombination med bayesisk logik och matematiska sannolikhetsresonemang för att skapa planer baserade på osäkerheter och inkomplett information. Det kan vara fördelaktigt att arbeta med inkomplett information då agenter ofta ska emulera en mänsklig spelare som inte har tillgång till all information om vad som händer runt om i spelvärden. [1, 5]

2.2 Struktur och Beslutsfattande

2.2.1 Tillståndsmaskin

Tillståndsmaskiner eller Finite State Machines (FSM) är regelbaserade system med ett antal tillstånd som är sammankopplade via övergångar mellan tillstånden. Tillståndsmaskinen kan beskrivas som en riktad graf och den befinner sig i ett begränsat antal tillstånd under exekvering. Ett tillstånd består av ett antal handlingar och representerar ett beteende hos agenten. [2, 4, 5]

Det finns många olika sätt att implementera tillståndsmaskiner men i allmänhet är de intuitiva och väldigt flexibla. Nya tillstånd kan läggas till efterhand som de behövs och tillståndsmaskinen kan användas som ett ramverk där olika tillstånd kan använda andra tekniker för artificiell intelligens.[2, 4]

På grund av de många fördelar som finns med tillståndsmaskiner har de under en lång tid varit det vanligaste sättet att strukturera beslutsfattande i artificiell intelligens. [2]

Ett problem med tillståndsmaskiner är att de kan vara svåra att organisera när de innefattar många tillstånd. Ett flertal olika metoder har utvecklats för att hantera detta problem. En annan svårighet med tillståndsmaskiner är att varje tillstånd är skapat för att passa in i ett visst sammanhang och kan därför vara svårt att återanvända, speciellt om övergångarna till andra tillstånd är en del av tillståndet.

Beslutsträd

Ett vanligt sätt att hantera övergångarna i en tillståndsmaskin är att använda ett beslutsträd. Efter att ett tillstånd exekverats anropas beslutsträdet för att välja vilket agentens nästa tillstånd skall bli. Trädets löv representerar agentens beslut om nästa tillstånd. [5]

Beslutsträd finns också som en struktur skild från tillståndsmaskiner vilket beskrivs mer detaljerat i kapitel 2.2.2.

Tillståndsövergångstabell

Ett enkelt strukturerat sätt att kontrollera sina tillstånd är att använda en tabell som man med jämna mellanrum tillfrågar om vilket tillstånd agenten skall befinna sig i. Tabellen består av agentens alla övergångar mellan olika tillstånd och specificerar nuvarande tillstånd, det villkor som gäller för den specifika övergången och det tillstånd som övergången leder till. Exempel på en tillståndsövergångstabell finns i tabell 2.1.

Nuvarande tillstånd	**Villkor**	**Nästa tillstånd**
Fly	Trygg	Patrullera
Attackera	StarkareFiende	Fly
Patrullera	Hotad OCH SvagareFiende	Attackera
Patrullera	Hotad OCH StarkareFiende	Fly

Tabell: 2.1 *(Exempel på struktur hos en tillståndsövergångstabell)*

Med den här strukturen kan varje tillstånd utformas som separata objekt eller funktioner som är externa från beslutsenheten i agenten. Beslutsenheten skapas med uppgifter som utvärderar spelmiljön, uppdaterar informationen om villkoren för övergångar och väljer vilket tillstånd agenten skall befinna sig i. [2]

Objektorienterad struktur

Ett sätt att strukturera tillstånden som kan användas om tillstånden definieras som objekt är att låta varje tillstånd ha en prioritet och en metod för att uppdatera och returnera denna prioritet. Då tillstånden behandlas som objekt kan de organiseras i en lista och en metod för att välja tillstånd kan baseras på att gå igenom listan och välja ut tillståndet med högst prioritet. [4]

En stor utmaning med denna struktur är att skapa metoderna för att bestämma prioriteten för varje enskilt tillstånd. Prioriteten bör baseras på dels agentens olika mål, som i sin tur är prioriterade, och dels på agentens möjlighet att

utföra dessa mål baserat på indata från den omgivande spelvärlden. Strukturen är i sig väldigt flexibel då det är enkelt att lägga till och ta bort olika tillstånd från listan som definierar agenten och därmed kan strukturen enkelt återanvändas för ett flertal olika typer av agenter. [4]

Hierarkisk Tillståndsmaskin
Idén med den hierarkiska strukturen är att flera tillstånd kan dela övergångar. Delandet av övergångar uppnås genom att använda flera nivåer av tillståndsmaskiner där tillstånd i övre nivåer av den hierarkiska tillståndsmaskinen utgör själva tillståndsmaskinen för den lägre nivån. Modellen kan drastiskt minska komplexiteten i hantering och överskådlighet av alla tillstånd och övergångar jämfört med att en tillståndsmaskin med en nivå. [7,11]

2.2.2 Beslutsträd
Beslutsträd fungerar genom ett antal beslut strukturerade i en riktad acyklisk graf. Det kallas beslutsträd trots att det inte är en trädstruktur då grenar och noder tillåts att återanvändas på olika delar i beslutsträdet. Ett beslutsträd traverseras tills lövnoderna nås. Lövnoderna i beslutsträdet representerar villkor eller handlinningar som testas respektive utförs omedelbart när lövnoderna nås. [5]

Ett vanligt sätt att strukturera beslutsträd är en hierarkisk struktur där alla noder representerar uppgifter och deluppgifter. Varje uppgift bryts rekursivt ner tills de når lövnoderna. Lövnoderna som är villkor eller handlingar är i sig deluppgifter som skall utföras. Övriga noder innefattar alla ett antal sätt att sorterade deluppgifter. Dessa noder kan vara exempelvis sekvenser, väljare, prioritetsväljare eller dekoratörer. Varje nod/deluppgift i ett beslutsträd kan antingen vara i körande tillstånd, lyckas eller misslyckas vilket är grunden för hur logiken i beslutsträd är uppbyggd. [7, 8, 9, 14]

En sekvens utför i tur och ordning alla deluppgifterna i sin lista så länge som varje deluppgift lyckas. Så fort en deluppgift misslyckas avbryter sekvensen sin exekvering och returnerar ett misslyckande. Endast om alla dess deluppgifter lyckas returnerar sekvensen information om att den lyckades. En sekvens försöker alltså utföra en mängd deluppgifter i en planerad ordning där var deluppgift kräver att de framförvarande deluppgifterna är utförda. [7, 8, 13, 14]

Väljare kompletterar detta genom att i tur och ordning utföra deluppgifterna så länge som varje deluppgift misslyckas. Så fort någon deluppgift lyckas avbryter väljaren exekvering och returnerar information om att den lyckades, endast om alla väljarens deluppgifter misslyckas returneras ett misslyckande

uppåt i trädet. Väljaren försöker alltså utföra en utav en mängd deluppgifter. Ordningen på väljarens uppgifter kan ses som att den första är högst prioriterad och därefter försöker väljaren exekvera bästa möjliga reservrutin. [7, 8, 12, 14]

Dekoratörer är en typ av nod som exekveras och endast har en barnnod. En dekoratör exekveras när den ingår i vägen från roten till en exekverande lövnod. Dekoratörer är ett sätt att tillåta att utökat beteende kan adderas till en gren utan att modifiera övrig kod.
[7, 8,10]

Strukturerna som byggs med detta system är väldigt dynamiska och tillåter att deluppgifter används som byggstenar och kan flyttas runt och återanvändas på olika ställen i trädet.
[7, 8, 9, 14]

Lärande och planering
Beslutsträd kan kombineras med olika lärande algoritmer och målinriktade planer som agenten utvecklar allteftersom den lär sig. En agent utför sina inplanerade uppgifter, därefter utvärderar agenten följderna av sina beslut jämfört med för agenten specificerade önskemål och åsikter. Informationen om hur bra ett beslut är används sen för att till exempel generera nya grenar till beslutsträdet med den nya informationen agenten lärt sig. Agenten kan också använda den nyinlärda informationen för att uppdatera sina mål och sin planering. [5]

2.2.3 Neurala nätverk
Neurala nätverk är en grupp tekniker för lärande som är utvecklat att efterlikna utformningen på kopplingarna i hjärnan och nervsystemet hos djur. Systemet fungerar genom att upprepade gånger justera interna numeriska parametrar i ett flertal komponenter i nätverket för att nå ett nära optimal svar på uppgiften agenten ska lära sig. Olika sådana uppgifter delas oftast upp i olika klasser som ingår i det neurala nätverket. Tekniken kräver att det finns data tillgänglig som det neurala nätverket kan bearbeta för att ställa in sina parametrar. [1, 5]

Inom spel är neurala nätverk en vanlig typ av “lärande” artificiell intelligens. Lärande är väldigt praktiskt när agenter ska kunna anpassa sig till externa faktorer som till exempel olika spelstilar hos spelaren. [5]

2.3 Pathfinding

Att kontrollera hur en agent kan röra sig i spelvärlden är ett trivialt problem jämfört med att styra vart agenten skall färdas. Hur en agent kan röra sig styrs lämpligen genom att definiera ett flertal variabler som till exempel hastighet och acceleration i olika riktningar, hur fort agenten kan vridas, om agenten kan hoppa, hur högt kan den hoppa, med mera. Dessa variabler används sedan för att påverka agenten enligt hur de specificerats. Variabler används för att enkelt kunna justeras för bästa möjliga resultat i spelet. Därmed blir koden också enkel att återanvända för agenter med olika rörelsemönster.[2]

Det är möjligt att styra agenten till sitt mål genom att läsa av omgivningen och beräkna vägar runt hinder. Då avläses det var i omgivningen det finns objekt som agenten kan kollidera med och den informationen behandlas för att hitta tänkbara vägar till sitt mål. Avsökning av geometrin kräver relativt mycket prestanda och kan ge ett osäkert resultat när det gäller att hitta snabbaste vägen till ett mål. Andra angreppsmetoder använder sig av olika typer av datastrukturer och är både effektivare och ger ett mer precist resultat. Att avläsa geometrin är därmed inte att rekommendera som bas för någon avancerad pathfinding. Däremot kan det vara praktiskt för att undvika dynamiska hinder i spelvärlden som en statiskt underliggande struktur inte kan hantera. [1]

För att undvika behovet att avsöka omgivningen kan en eller flera underliggande strukturer skapas som agenters navigering kan baseras på. Dessa strukturer kallas navigationsgrafer och är en abstraktion där alla positioner som en agent kan befinna sig på i en spelmiljö representeras av en nod och kopplingarna mellan dessa positioner representeras av en båge. Navigationsgrafen innehåller alltså alla möjliga vägar en agent kan färdas i spelet. Varje båge har en kostnad, vilket i enklaste fall representerar sträckan i spelet som bågen representeras. För att använda sig av denna information används en effektiv sökalgoritm. Prestandan för en sökning beror dock inte enbart på sökalgoritmen utan även mycket på den graf som sökningen utförs på. Vilken typ av datastruktur som bör användas varierar från spel till spel. [5]

2.3.1 Rutnät

En vanlig typ av navigationsgraf är ett rutnät av celler som täcker hela spelvärlden. Modellen är speciellt populär i strategispel och rollspel. Fördelar med rutnätsmodellen är att det är enkelt att spara information i var cell som senare kan användas för att beräkna passande vägar för agenterna. [2]

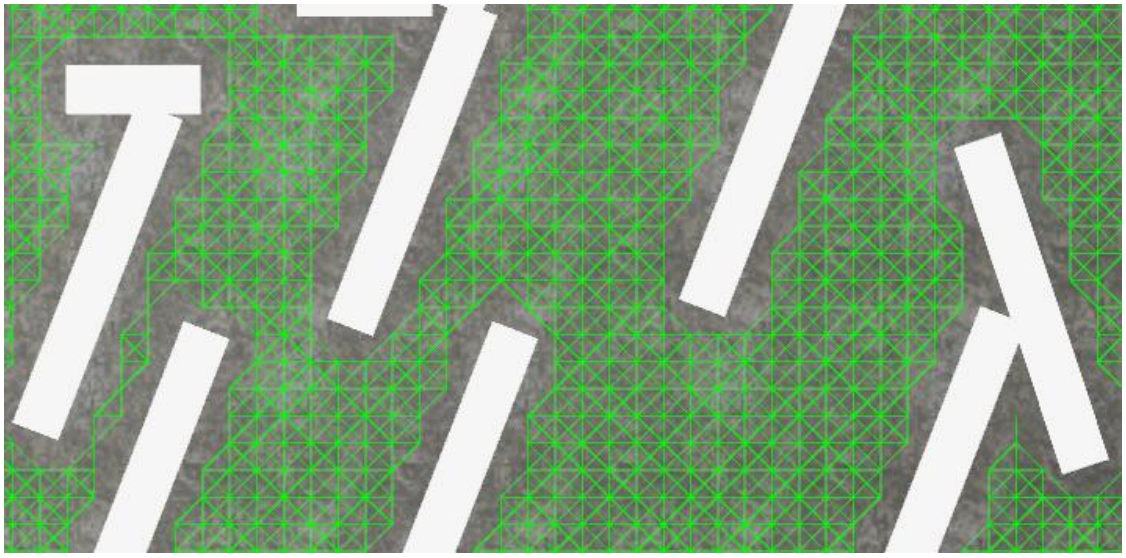

Bild 2.1 *(Exempel på ett rutnät där de gröna linjerna representerar sökbara vägar.)*

Den stora nackdelen med att använda sig av en rutnätsbaserad graf är att sökrymden snabbt blir väldigt stor och prestandan för sökningar blir därmed sämre. [2]

2.3.2 Point Of Visiblilty (POV)-graf

En POV-graf är en graf uppbyggd av strategiskt utplacerade noder i spelmiljön där varje nod kan kopplas till minst en annan nod via en rak linje som agenten kan färdas längsmed. Dessa noder kopplar samman alla områden i spelmiljön som agenterna kan befinna sig på. [1, 2, 23, 24, 30]

Där är mycket att ta hänsyn till vid placeringen av noderna, bland annat bör det undvikas att onödigt många bågar skapas då dessa kostar extra minne och prestanda för sökfunktionen. Det får heller inte vara för få noder så att blinda punkter skapas i geometrin. En blind punkt är en yta som inte kan nås av någon nod via en rak linje.
En brist med POV-grafer är att de behöver vara anpassade till hur agenten som använder grafen rör sig. Grafen kan inte inte ha för skarpa kurvor eller för smala öppningar för agenten, därmed är moddellen ej praktisk då man använder olika typer av agenter med varierande dimmensioner och rörelsemönster. Att skapa dessa grafer är ofta tidskrävande men det har utvecklats verktyg för att automatiskt generera dessa grafer utifrån spelets geometri. [1, 2, 24, 30]

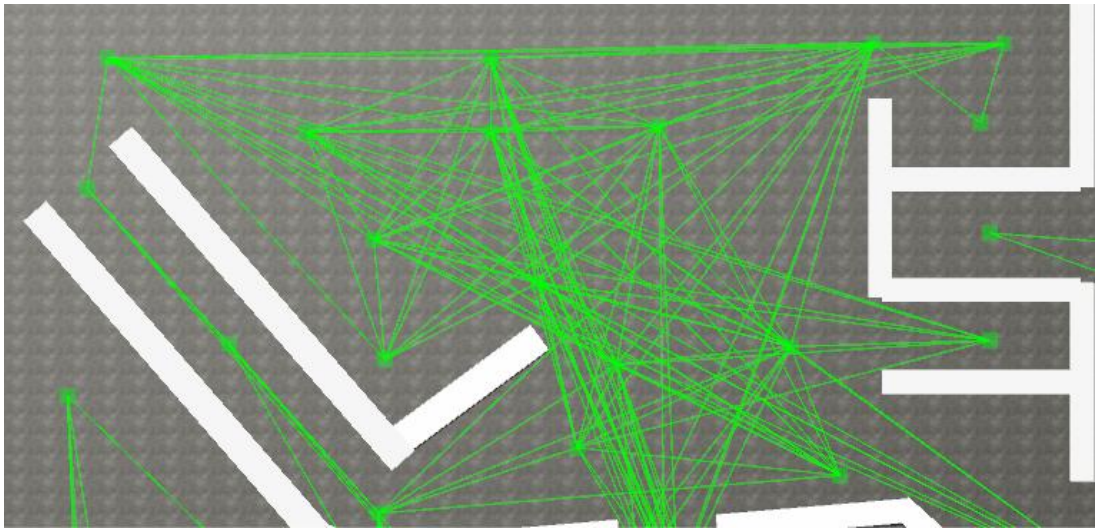

Bild 2.2 *(Exempel på en POV-grav där noder representeras av gröna kuber och bågar representeras av gröna linjer.)*

En sökning genom denna graf utförs i följande steg:

- Från nuvarande position identifiera närmast synliga nod (a)
- Från målets position identifiera närmast synliga nod (b)
- Identifiera den effektivaste vägen från (a) till (b)
- Flytta till (a)
- Följ funnen väg till (b)
- Flytta till målet

Att agenterna inte känner till något om omgivningen utanför bågarna mellan noderna leder också till ett flertal problem. Agentens väg från en godtycklig punkt till en nod i grafen och från en nod till ett slutmål behöver kontrolleras. Ett fenomen som uppstår är att agenterna ofta går i sicksack för att komma till sitt mål. För att lösa problemet används en extra algoritm för att släta ut agentens väg, det kostar extra prestanda men får agentens väg att se naturlig ut. [1, 2, 24, 30]

Ett annat problem är att det inte har något stöd för att rätta till sin vägbana, vilket leder till stora svårigheter att ha dynamiska hinder i spelmiljön. Ett exempel på detta kan vara en gång som är sammankopplad av en båge med en nod i var ände, om en bit av en agents väg längs den bågen blockeras av en låda kan POV-grafen inte hjälpa agenten navigera runt hindret. [1, 6]

2 3.3 Navmesher

Navmesher bygger på att all yta som en agent kan befinna sig på och röra sig på delas in i konvexa polygoner. Agenter kan alltså röra sig fritt inom var polygon. För att söka i en navmesh används noder antingen mitt i var polygon eller i de kanter som en polygon har gemensamt med en annan polygon och

låter dessa noder fungera som en POV-graf.
Navmeshen kan placeras ut för hand men där finns även verktyg som kan generera den utifrån spelets geometri. [2, 3, 6, 23, 24, 30]

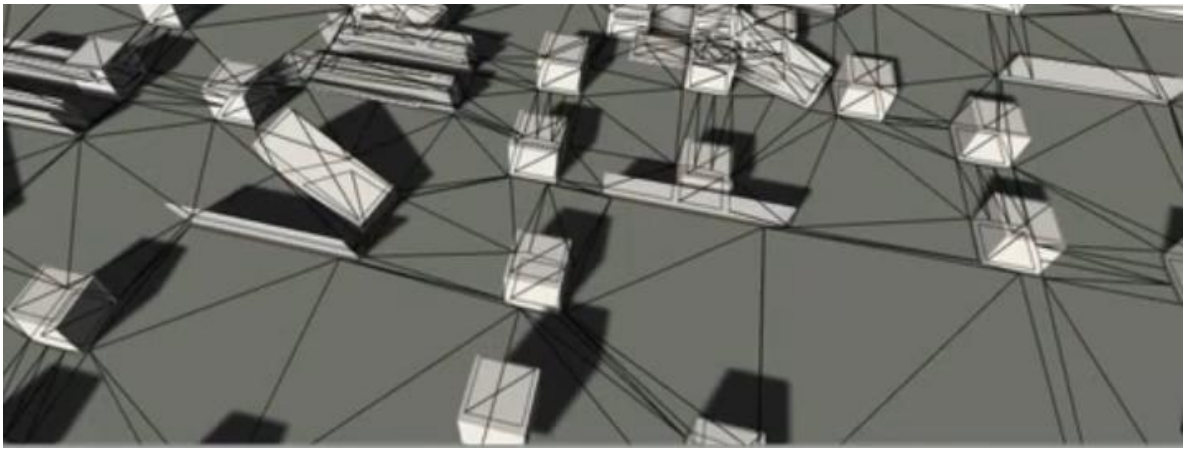

Bild 2.3 *(Exempel på en navmesh med triangulära polygoner utritade med svarta sträck.)*

En sökning i en navmesh utförs i följande steg:

- Identifiera startpolygon (a)
- Identifiera slutpolygon (b)
- Identifiera effektivaste väg från startposition till slutposition
- Följ funnen väg till slutposition

Navmesher används på ett sätt liknande POV-grafer.
En fördel gentemot POV-grafer är informationen som finns om ytor möjliga att röra sig på utöver de bågar som bildas av noderna. Det medför bland annat att samma graf kan användas för olika rörelsemönster då ett rörelsemönster kan anpassas till navmeshen istället för tvärtom. Upplägget med konvexa polygoner innebär att inga blinda punkter kan bildas i grafen. Dessutom kan agenten oftast förflyttas i en rak linje från en godtycklig punkt till en nod i grafen. Agenterna har en viss tendens att gå i sicksack mellan noderna vilket löses genom att lägga till en algoritm för att räta ut vägen på samma sätt som vid POV-grafer. Där finns inget stöd för dynamiska hinder i navmesher men informationen om den statiska geografin i spelet kan vara till nytta om vid integrering av ett system för dynamiska hinder. [2, 3, 6, 18, 23, 24, 30]

2.3.4 A*

A* är en algoritm som söker igenom en graf och hittar den effektivaste vägen till en målnod. Algoritmen bygger på att upprepade gånger undersöka den mest lovande icke undersökta nod som är granne med en undersökt nod. Inom artificiell intelligens och pathfinding har algoritmen använts sedan 1968. [1, 3]

Det finns många varianter och anpassningar som bygger på A*. Exempelvis kan dessa varianter arbeta dynamiskt eller inkludera optimeringar som hade behövt göras efter att A* funnit en sökväg. Andra anpassningar fungerar enbart på en viss typ av navigationsgraf. Detta examensarbete omfattar dock ej en undersökning av de olika varianterna eller utbyggnaderna på algoritmen. [5]

Algoritmen använder sig av två listor, Open för icke undersökta och Closed för undersökta noder. I början innehåller Open en startnod som representerar agentens nuvarande position och Closed är tom. I varje iteration plockar algoritmen ut den mest lovande noden ur Open och undersöker den. Om noden inte är slutnoden placeras den i Closed och alla dess grannoder gås igenom, de som inte redan finns i Closed beräknas den uppskattade kostnaden till målnoden för och sorterar in i Open, om en nod redan skulle finnas i open jämförs dess tidigare värde med det nya och uppdaterar om det nya var mindre. Dessa iterationer pågår tills slutnoden är funnen eller Open är tom. Beräknandet av den uppskattade kostnaden till målnoden görs enligt $F(x) = G(x) + H(x)$. Där x är noden som undersöks, F(x) är den uppskattade kostnaden till målnoden från startnoden via x, G(x) är kostnaden för vägen från ursprungsnoden till x och H(x) är en heuristisk formel för uppskattning av kostnaden från x till målnoden.

Där finns flera olika heuristiska formler som kan användas, dessa kan sen viktas för att definiera hur mycket sökningen skall bero på heuristiken jämfört med den beräknade kostnaden. Om kostnaden till målet överskattas kan det leda till att det inte kan garanteras att algoritmen kan hitta den optimala vägen till målet. Å andra sidan ökar effektiviteten hos en sökning om det uppskattade värdet tillåts vara större. Det är därför viktigt att avgöra vilken heuristik att använda för att optimera algoritmen för hastighet. Ett vanligt sätt att beräkna heuristiken, H(x), vid pathfinding är att använda den euklidiska sträckan. Den euklidiska sträckan är den faktiska sträckan på spelets karta multiplicerad med kostnaden för den lägst bekostade terrängen. Den euklidiska sträckan passar som heuristik då den är kortare eller ekvivalent med den faktiska sträckan agenten måste färdas då sträckan inte kan vara kortare än fågelvägen. En enkel och ofta använd optimering är att istället multiplicera med den typiska kostnaden av terrängen agenten befinner sig i. [1, 3]

Dijkstras algoritm med heuristik

A* har stora likheter med Dijkstras algoritm, som är den effektivaste för att finna alla de kortaste vägarna till en nod. Skillnaden mellan dem, som ger A* en fördel vid pathfinding ligger i heuristiken och att Dijkstras algoritm alltid undersöker alla noder. Att använda Dijkstras algoritm för att finna kortaste vägen mellan två punkter är ekvivalent med att använda A* utan heuristik. [2]

Pseudokod

```
Open: priorityqueue of searchnode
Closed list of searchnode
AStarSearch(location StartLoc, location GoalLoc, agenttyp Agent){
     Clear Opand and Closed
     //initialize startcode
     StartNode.Loc = StartLoc
     StartNode.CostFromStart = 0
     StartNode.CostToGoal = PathCostEstimate(StartLoc,GoalLoc,Agent)
     StartNode.Parent = null
     push StartNode on Open
     //process the list until success or failure
     while Open is not empty{
      pop Node from Open //Node has lowest cost
      if(Node is a goal node){
            construct path backward from Node to StartLoc
            return success
      }else{
            for each successor NewNode of Node{
                  NewCost = Node.CostFromStart + TraverseCost(Node,
NewNode,Agent)
                  if(NewNode is in Open or Closed) and
(NewNode.CostFromStart <= NewCost){
                        continue
                  }else{
                        NewNode.Parent = Node
                        NewNode.CostFromStart = NewCost
                        NewNode.CostToGoal =
PathCostEstimate(NewNode.Loc,GoalLoc, Agent)
                        NewNode.TotalCost = NewNode.CostFromStart +
NewNode.CostToGoal
                        if(NewNode is in Closed)
                              remove NewNode from Closed
                        if(NewNode is in Open)
                              adjust NewNode's position in Open
                        else
                              push NewNode onto Open
                  }
            }//done with Node
      }
      push Node onto Closed
     }
     return failure
}
```

[3 sida 255]

Path smoothing

Ett problem som ofta uppstår i vägar planerade efter datastrukturer är att vägen till målet innehåller onödigt många svängar. En av de enklare angreppsmetoderna för att lösa problemet är att testa om det är möjligt att färdas i en rak sträcka från en godtycklig nod n till nod (n+2) och om så är fallet tas nod n+1 bort. [2]

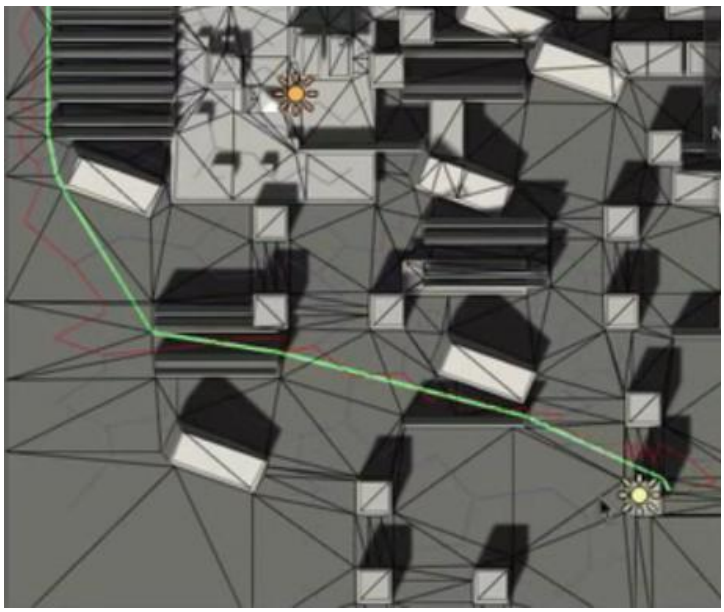

Bild 2.4 *(Exempel på pathsmoothing där sökt väg representeras av en röd linje och utjämnad väg representeras av en grön linje.)*

De skarpa svängar som ofta bildas i vägarna skapade av A* kan få agenternas rörelsemönster att se onaturligt ut. Att dämpa rotationen hos agenterna kan maskera de skarpa svängarna men får istället agenten att svänga ut i en kort omväg i dessa skarpa svängar. Ett alternativ till detta är att använda splines som beräknas matematiska formler som används på den funna vägen för att generera extra noder som får kurvan att bli jämnare ut. Exempel på en sådan formel är Catmull-Rom formeln. [2]

Hierarkier av datastrukturer

En effektiv metod för att minska kostnaden av pathfinding är att använda sig av två eller eventuellt flera hierarkier av datastrukturer. Ett vanligt exempel på användningsområde för detta är att på en högre nivå låta en graf representera olika dörröppningar mellan rum och korridorer i en byggnad och på en lägre nivå låta en annan graf representera alla möjliga vägar agenten kan färdas i dessa rum och korridorer. Den hierarkiska strukturen används enklast genom att först finna bästa vägen i grafen på hög nivå och därefter successivt finna den bästa vägen mellan var dörröppning i grafen på lägre nivå. [3]

Ett problem med denna metod är att agenterna kommer att färdas genom mitten av dörröppningarna som representerar delmål i grafen. Om dörröppningarna är breda jämfört med agenten kan detta se konstigt ut. För en mer direkt väg till målet söks den bästa vägen till dörröppningen som följer nästa dörröppning. Var gång agenten passerar en dörr utförs en ny sökning och resten av den beräknade vägen används ej. Detta innebär en stor extra kostnad i prestanda men finner garanterat den optimala vägen till målet. [3]

3 Implementerade lösningar

Det finns ett flertal redan implementerade plugin som hanterar beslutsfattande och pathfinding. För att spara tid på egen utveckling och för att hitta effektiva verktyg att arbeta med undersöktes ett flertal implementerade lösningar inför vidare användning i Unknown. Undersökningen fokuserades på plugin skapade för användning med Unity och är begränsad till plugin som är tillgängliga att använda utan kostnad.

Utförandet av undersökningen har varierat på grund av ett flertal faktorer som skiljer sig från plugin till plugin. Exempel på skillnader är omfattning av funktionalitet, hur väl dokumentationen är utförd, om det finns exempel att granska, och hur komplicerat det är att skapa egna prototyper. Gemensamt för alla undersökningar är att fokus ligger på att identifiera varje plugins grundläggande egenskaper och erbjudna funktionalitet.

3.1 Plugins för strukturering

3.1.1 Finite State Machine

Den här tillståndsmaskinen erbjuder fri tillgång till all källkod och är anpassad för användning i Unity. Den är programmerad i C# vilket medför att den är kompatibel med Unknown. [26]

Funktionaliteten innefattar ett enkelt ramverk för hantering av övergångar, tillstånd och själva tillståndsmaskinen. Det ingår också ett exempel på användning som kan modifieras för att användas i egna projekt. [26]

Undersökning
Finite State Machine granskades genom att läsa igenom källkod och exekvera tillgängligt exempel i Unity. Samt verifierades resultat från exemplet genom utskrifter i Unity vid övergångar till nya tillstånd.

3.1.2 unity FSM

unity FSM är en tillståndsmaskin som är anpassad för Unity och erbjuder fri tillgång till källkod. Den är programmerad i C# vilket medför kompatibilitet med Unknown. Utöver ramverket som hanterar tillståndsmaskinen har unity FSM även struktur för att skapa handlingar som kan återanvändas i flera olika tillstånd eller övergångar och hantering av vilket sammanhang som tillståndsmaskinen används i. Hanteringen av sammanhang innebär att en tillståndsmaskin kan användas för att hantera många olika agenter vilket sparar mycket prestanda jämfört med att varje agent har sin egen kopia av tillståndsmaskinen. Det ingår exempel på användning av tillståndsmaskinen för

att lättare förstå hur den används. Dessutom har unity FSM ett enklare gränssnitt för att testa logiken som byggts in i sin tillståndsmaskin. [20]

Undersökning
unity FSM granskades genom att läsa igenom källkod och exekvera tillgängligt exempel i Unity. Samt verifierades resultat från exemplet genom användning av det tillgängliga gränssnittet för testning.

3.1.3 Behave 1.3
Behave är ett plugin designat för Unity med fullt stöd för C# och därmed kompatibel med Unknown. Behave är gratis att använda med specifika användarvillkor. Källkoden till Behave är inte tillgänglig. Funktionaliteten hos Behave är omfattande och innefattar bland annat ett grafiskt interface integrerat till Unity för enkelt skapande av beslutsträd med 6 olika typer av noder och hanteringen av övergångar mellan dessa noder. Det finns även stöd för återanvändning av grenar på flera platser olika beslutsträd. Utifrån trädet som skapats genereras en abstract klass av Behave som implementeras i ett script som hanterar stegande genom trädet och de handlingar som utförs i olika noder. Behave har också en debugfunktion för att i runtime följa vilken lövnod i ett specifikt träd som är aktiv under körning. [15]

Undersökning
Behave granskades genom att läsa igenom dokumentation och se på en instruktionsvideo från skaparen av Behave. Ett flertal egna beteendeträd skapades och testades i Unity med hjälp av debug funktionen i Behave samt utförda handlingar av agenten med hjälp av Behave.

3.2 Plugins för pathfinding

3.2.1 Recast och Detour
Denna lösning går utanför de avgränsningar som beskrivs i kapitel 1.4.5. Anledningen till att ett undantag gjordes beror dels på att NMgen i kapitel 3.2.2 är baserad på denna lösning och dessutom är en inkomplett port. [18] Dels på att jag ansåg att en det var lämpligt med en undersökning som kontrollerar vilka problem som uppstår då programvaran ej är anpassad för Unity.

Recast och Detour är tillsammans en lösning för pathfinding med navmesher. Recast implementerar en algoritm för att generera navmeshen utifrån ett flertal parametrar och avläsning av spelvärldens geometri. Detour innefattar bland annat sökalgorithmen och algoritmer för att i runtime uppdatera delar av en navmesh för temporära hinder i spelgeometrin. Recast och Detour är programmerade i C++ och ej anpassade för Unity vilket medför ett krav på

pro-versionen av Unity. Unity's stöd för C++ kräver dessutom att funktioner deklareras med C-länkning vilket kräver anpassning av koden. Vidare krävs det att ett omslag till de funktioner som skall anropas från C# kodas i C++ och kompileras till en DLL fil och en referens till denna fil C# koden som använder metoderna. Recast och Detour har blivit väldigt populära som effektiva och funktionella lösningar för pathfinding och används i många projekt inom spelutveckling.[7, 19]

Undersökning
Undersökning påbörjades med att läsa igenom relevanta delar av utvecklarens blog. Därefter granskades delar av källkoden och det medföljande exempel programmet testades. I testet flyttades en egen spelmiljö in till exempelprogrammet där en navmesh genererades varpå sökningar utfördes i den genererade navmeshen.

3.2.2 NMgen
NMgen är ett Unityanpassat plugin programmerat i Java med krav på Unity pro. NMgen genererar navmesher och är baserat på Recast men är ingen fullständig port. NMgen innefattar ingen sökfunktion och behöver kompletteras med en sådan för att fungera som en komplett pathfindinglösning. [18]

Undersökning
NMgen undersöktes genom att granska dokumentation och delar av källkoden. Inga tester utfördes på NMgen på grund av kravet på Unity pro.

3.2.3 Path 2.0.0f2
Path är en Unityanpassad pathfindinglösning baserad på POV-grafer. Path är gratis med specifika användarvillkor. Det är kompatibelt med C# och har inga kompabilitetskonflikter med Unknown. Path har många funktioner för att göra det enkelt att skapa POV-grafen. Det är smidigt att placera noder med samma verktyg som används i Unity för att flytta objekt. Noder kan täcka olika stora ytor. För att skapa bågar mellan noder finns det en funktion för att automatiskt generera dessa alternativt kan bågar hanteras individuellt. Varje båge har en viss bredd och kan ha olika vikter för att representera olika terränger agenten rör sig i. Path hanterar hinder i specifika lager från Unity. Det finns stöd för att temporärt blockera bågar i Path och låta agenten reagera på när en beräknad väg för agenten blockeras. Path har också en valbar funktion för att ta genvägar förbi onödiga noder i den beräknade vägen. Man får själv implementera kontrollen som styr agenten längs grafen och vill man generera jämnare kurvor finns det utrymme för att implementera detta utanför Path. [15]

Undersökning
Path granskades genom at undersöka dokumentation och se på en instruktionsvideo av användaren. Därefter skapades ett flertal POV-grafer med Path i Unity och sökning utförs och verifieras med hjälp av debug funktioner som ritar ut den beräknade vägen.

3.2.4 UnitySteer 2.2
UnitySteer är en rörelsekontroll som fokuserar på att undvika kollisioner med både statiska och rörliga objekt. UnitySteer är fritt tillgängligt under en open source licens och är anpassat till Unity och skrivet i C#. UnitySteer kan användas för att styra mot ett mål men det är inte anpassat för att ta den effektivaste vägen. Det är möjligt att kombinera UnitySteer med andra pathfindinglösningar där man låter agenten färdas mot noder längs en beräknad väg med hjälp av UnitySteer som även kontrollerar att agenten inte kolliderar med dynamiska objekt. UnitySteer innefattar många komponenter och många olika Script vilket gör det svårt att överblicka, flera exempelprojekt för Unity gör det dock enklare att komma igång att använda det. [16]

Undersökning
UnitySteer undersöktes genom att granska och testa medföljande exempel. Det medföljde sexton olika exempel miljöer som kördes i Unity av dessa granskades den kontrollerande koden på tio och det resulterade beteende på samtliga sexton.

3.2.5 A* Pathfinding 2.95
A* Pathfinding är en gratis pathfindinglösning som stödjer flera olika datastrukturer som bas. Det är Unityanpassat och programmerat i C#. A* pathfinding hanterar rutnät, navmesher och POV-grafer och även några olika tekniker för att generera dessa. Hinder i geometrin hanteras i specificerade lager i Unity. Det ingår hantering för enskilda bågar och möjlighet att blockera delar av den beräknade representationen av geometrin. Det ingår även valbara alternativ för utjämning av kurvor och genvägar förbi onödiga noder. Debugalternativ tillåter visualisering av all information om grafen och sökningar som utförs i grafen. För rutnät finns det möjligheten att i runtime uppdatera delar av rutnätet, vilket är praktiskt för att undvika dynamiska objekt. Det finns ett flertal exempel till Unity för att testa de olika teknikerna som kan användas med A* pathfinding och där ingår även rörelsekontroll för agenter som är fri att anpassas till egna projekt. [17]

Undersökning
Undersökningen av A* pathfinding påbörjades genom att granska de tio exempel som följde med. Även den dokumentation som finns på hemsidan för pluginet granskades. Källkoden för styrning av agenten längs sökta vägar undersöktes samt generades ett flertal rutnät med hjälp av A* pathfinding.

4 Val för utveckling

Positiva egenskaper vid utveckling
För att bestämma vilken/vilka lösningar som ska användas för utvecklingen av Unknown specificerade jag en lista med positiva egenskaper vid utveckling som jag letar efter i de plugin och tekniker jag utvärderar. Mitt val av plugin är baserat på kriterierna i följande lista.

1. En struktur uppbyggd av enkla delar (För lättanvänd och lättöverskådlig kod.)
2. Få begränsningar i användandet
3. Låg tidskostnad för användning
4. Låg tidskostnad för integrering och anpassning till Unknown
5. Möjlighet att använda gratis-versionen av Unity
6. Möjlighet för svårförutsägbart beteende i agenterna.
7. Prestanda
8. Återanvändbar utveckling (Möjligt att enkelt anpassa delar av utvecklingen till flera olika agenter.)

4.1 Tekniker för beslutsfattande

För neurala nätverk ansåg jag att fördelarna med lärande ej är speciellt åtråvärt för Unknown. Tillsammans med min uppfattning av komplexiteten i att utveckla tekniken och identifiera den data som behövs för att kontrollera en agent och lära den rätt saker leder till mitt beslut att inte fokusera på den tekniken.

Gällande tillståndsmaskiner anser jag att hantering av komplexare beslutsfattning bör använda en heuristisk variant som hanterar handlingar separat från tillstånd. En sådan modell tillåter återanvändning av både handlingar och länkar mellan tillstånd. Beteendeträd kan ses som en variant på denna typ av tillståndsmaskin med begränsningen att beteendeträdet är acyklisk i sina övergångar. Denna begränsning är dock inget hinder för någon funktionalitet och jag anser att det är ett praktiskt sätt att strukturera beslutsfattande.

4.2 Val av beslutsfattande

Jag har valt att använda mig av Behave 1.3 för strukturering av beslutsfattande för Unknown. De många praktiska funktionerna implementerade i Behave gör det enkelt att bygga upp logiken, ger god översikt för strukturen och erbjuder möjlighet för återanvändning och enkel revidering av logiken. Dessa funktioner ger en stor fördel i hur effektivt det är att arbeta med Behave jämfört med Finite State Machine och unity FSM. En punkt i listan som ingen av alternativen erbjuder utan vidare utveckling är punkt 5.

4.3 Tekniker för pathfinding

För representation av spelmiljön har jag uppfattningen att både rutnät, POV-grafer och navmesher täcker in Unknowns behov av funktionalitet. Det är dock önskvärt med optimeringar för färre och jämnare svängar i agentens rörelsemönster. Rutnätstekniken är mer kostsam i prestanda än de andra två teknikerna. Jag har uppfattat navmesher som aningen bättre än POV-grafer på grund av att agenten känner till stora ytor som den kan färdas på snarare än enbart smalare linjer mellan noderna i en POV-graf. Dessutom är det lättare att verifiera vilka ytor en agent har tillgång till och där blir inga blinda punkter vilket kan skapas då en POV-graf inte är korrekt utformad.

4.4 Val av pathfinding

Jag har valt att använda mig av A* pathfinding för att hantera pathfinding i Unknown. Till att börja med kommer jag att använda rutnätsmodellen som är tillgänglig i A* pathfinding på grund av hur enkla de är att generera och använda samt funktionaliteten med dynamisk uppdatering av rutnätet. Skulle den visa sig för prestandakrävande i miljöer andra än den jag testat modellen i planerar jag att använda mig av navmesher istället vilket A* pathfinding tillåter med endast en mindre korrigering i koden. Den större tidskostnaden med navmesher i A* pathfinding är tiden som krävs för att skapa eller integrera verktyg som genererar navmeshen. Då jag redan har verktyg som gör detta för rutnät så innebär det ingen större tidsförlust att börja med rutnät.

Recast och Detour uppfattade jag som en lösning med högre prestanda än A* pathfinding på grund av navmesh metoden som användes. Funktionaliteten är väl utformad och speciellt funktionen att uppdatera delar av meshen i realtime är praktisk. Tyvärr anser jag att tidskostnaden för en anpassning till Unity uppskattas vara för stor.

Path uppfattade jag som en väl utformad lösning med POV-grafer med välanpassade funktioner. Att jag valde A* pathfinding över path berodde mycket på hur smidigt det var att använda och anpassa kontrollen som styr

agenten i A* pathfinding jämfört med att implementera en egen i Path. Även den sparade tiden i generering av datastrukturer i A* pathfinding jämfört med placerandet av noder i Path samt en bug i Path där genvägar förbi noder beräknades genom hinder istället för runt dem i min test miljö vid ett fåtal tillfällen för Path ledde till att jag valde A* pathfinding istället för Path.

UnitySteer kommer jag inte använda då jag uppfattar funktioner i A* pathfinding som tillräckliga för att hantera de få dynamiska objekt och agenter vi planerar att ha i varje område med pathfinding och att det riskerar att bli en stor tidskostnad att sätta sig in i alla de system som UnitySteer erbjuder och anpassa dessa till Unknown och A* pathfinding för en relativt liten vinst som dessutom kommer med en extra prestandakostnad.

5 Resultat av utvecklingen

Resultatet från utvecklingen av artificiell intelligens till Unknown är ett beslutsträd utformat i Behave och skapat för en specifik typ av agent i spelet. Beslutsträdet har ett tillhörande script som itererar genom beslutsträdet och utför olika exempeluppgifter. Bland annat följer agenten efter spelaren.

Teknikerna som används tillåter enkel återanvändning av kod mellan flera agenter av samma typ. För olika typer av agenter kan olika grenar i beteendeträdet återanvändas och kod för hantering av de lövnoder som hanterar handlingarna kan återanvändas. Ett exempel är den nod som ser till att agenten följer efter spelaren. Koden som hanterar rörelsekontrollen behöver ofta inte ändras från en agent typ till en annan då den är utformad på ett sätt som tillåter enkla ändringar i variabler att ge ett annat rörelsemönster.

I listan över positiva egenskaper för utveckling har vissa prioriteringar fått göras. Exempelvis valet att arbeta med en rutnätsmodell ger mer fokus på effektiv utveckling, (punkt 2), än hög prestanda, (punkt 6). I dess nuvarande stadie är också punkt 5 angående oförutsägbart beteende heller ej uppfylld och det är möjligt att jag får utöka beslutsfattningen med ett system för detta. Troligtvis ett FuzzyLogic system.

Exempel från utvecklingen
För att visa hur utvecklingen hanteras har jag valt att demonstrera en mindre del av beteendeträdet jag skapat med Behave 1.3, förklara hur det hanteras och ge exempel på kod som exekveras från beslutsträdet och hanterar pathfinding. Jag tänkte också visa upp av en bit av funktionaliteten i A*pathfinding med visualisering av sökning på ett rutnät.

Jag vill nämna att även denna lilla del av utvecklingen som jag valt att demonstrera är en tidig implementation med utrymme för förändringar.

5.1 Utveckling med Behave

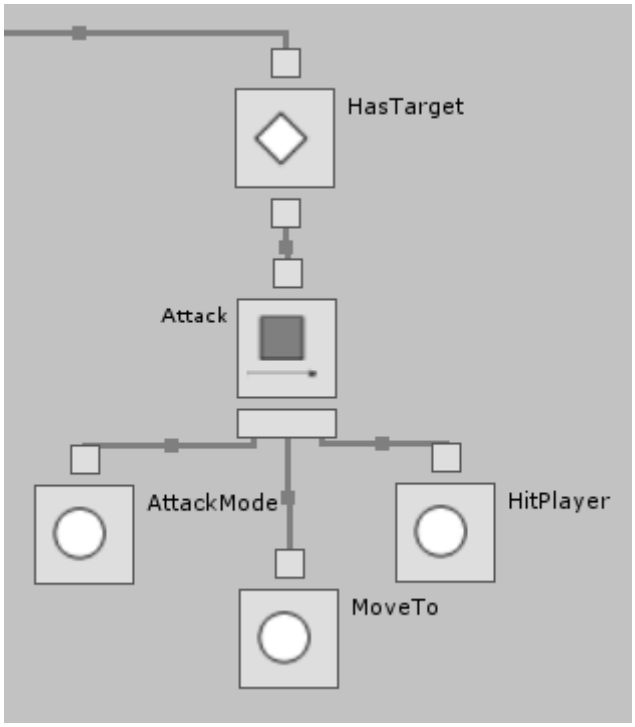

Bild 5.1 *(En gren i ett beteendeträd.)*

I bild 5.1 visas en gren i beslutsträdet som hanterar ett aggressivt beteende hos agenten.
Grenen består av:

- 1 Dekoratör (HasTarget)
- 1 Sekvens (Attack)
- 3 Handlingar (AttackMode, MoveTo, HitPlayer)

I HasTarget finns kod som ser till att det aggressiva beteendet exekveras så länge som agenten har en fiende som är vid liv och agenten vet var fienden har för position. Fler villkor skulle kunna läggas till HasTarget men det är inte nödvändigt för den här agenten. Skulle villkoren inte vara tillfredsställande för aggressivt beteende avlutas pågående handlingar från HasTarget.

Attack kan som en sekvens inte påverkas efter att den har skapats utan utför handlingarna från vänster till höger så länge som villkoren är uppfyllda i HasTarget. Skulle en handling misslyckas börjar sekvensen om från vänster.

AttackMode används för att sätta flaggor som beskriver ett tillstånd för agenten. Andra noder i den exekverande grenen eller andra script kan användas för att kontrollera agentens tillstånd.

I MoveTo startas rutiner som styr agenten mot spelaren och låter dessa rutiner pågå tills agenten är nära spelaren.

HitPlayer skickar ett meddelande till spelaren om att den blivit träffad.

Alla handlingar och dekoratorer finns i en klass som hanterar en specifik agent. I den klassen startas en rutin som tickar genom trädets lövnoder med en vald frekvens. Det största arbetet i grenen kan man säga sker i MoveTo. Följande är C# metoden som hanterar MoveTo:

```
public BehaveResult TickMoveToAction(Tree Sender)
{
if( !aiFollow.continousTargetSearch || aiFollow.target != 
player)
{
     aiFollow.StartFollowing(player);
    }
if(GetTargetDistance(player)<=close && 
InLineOfSight(player))
{
     aiFollow.StopFollowing();
     return BehaveResult.Success;
    }
    else
     return BehaveResult.Running;
    }
}
```

Som kan utläsas hanterar beslutsträdet pathfinding indirekt via aiFollow-objektet som anropas för att starta och stoppa agenten från att följa spelaren via metoderna StartFollowing(Transform target) och StopFollowing(). Klassen aiFollow är skapad från ett exempelscript från A*pathfinding som jag har modifierat för att kunna starta och stoppa från agentens beslutsträd. Det är inte i aiFollow som själva sökalgoritmen finns, däremot finns rutinerna som upprepade gånger anropar sökningar på vägar till ett rörligt mål och styr agenten efter de funna vägarna. Koden som hanterar pathfinding och styrning av agenten är omfattande om man ska ta den i sin helhet med varje detalj och mer än jag anser rimligt att förklara i det här exemplet. TickMoveToAction-metoden är ett exempel på hur strukturen jag använder sätter den övergripande kontrollen av agentens handlingar i beslutsträdet.

5.2 Utveckling med A* pathfinding

I det här kapitlet demonstrerar jag arbete med ett rutnät i A* pathfinding.
I bild 5.2 visualiseras scenariot och problemet som jag ska lösa i exemplet. En agent har fått syn på spelaren genom ett fönster, beslutat att attackera och funnit en väg till spelaren som den färdas längs med. Sträckan agenten för tillfället färdas längs med är den tunna blåa linjen och den beräknade vägen till spelaren är den orange linjen. Problemet som uppstått är att den funna vägen går genom en vägg vilket naturligtvis inte är meningen. Problem området visas tydligare i bild 5.3.

I bild 5.4 har jag ställt in A* pathfinding på att rita ut rutnätet som visualiseras med blå linjer där agenten kan färdas och röda klossar där algoritmen som skapar rutnätet har känt av hinder i spelmiljön och inte placerat ut några noder till rutnätet. Som standard är visualiseringen av rutnätet avstängd då den kräver mycket prestanda när det är ett så stort rutnät som i mitt fall.

I bild 5.4 kan man se att det blivit fel och placerats ut noder inuti väggen. Agenten kände därmed inte till att väggen fanns där och uppfattade det som en kortare väg till spelaren.

I bild 5.5 åtgärdas problemet genom att använda verktyg som finns i A* pathfinding för att hantera noder där jag i det här fallet väljer ”Node Disabler” som blockerar den nod som skapar en väg genom väggen.

I bild 5.6 visualiseras den blockerade noden som en röd boll på grund av inställningar i A*pathfinding. Bild 5.6 visar också resultatet då agenten den här gången har den funnit en korrekt väg till spelaren genom dörröppningen.

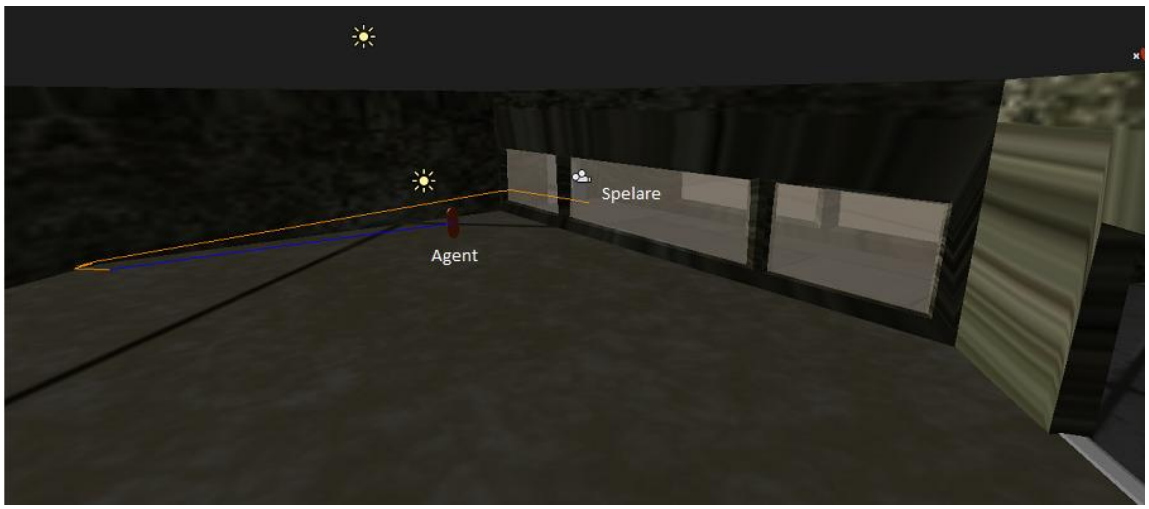

Bild 5.2 *(En agent med beräknad sökväg till spelaren visualiserad med en blå linje till närmsta vändpunkt och resterande sökväg med en orange linje. Den beräknade sökvägen går genom en väg.)*

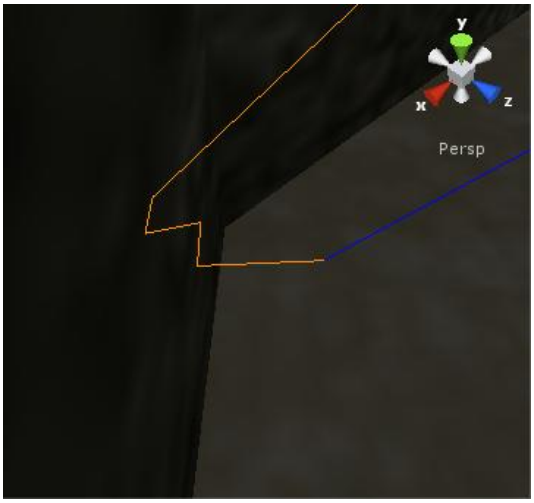

Bild 5.3 *(Inzoommning på ett problem med den beräknade sökvägen)*

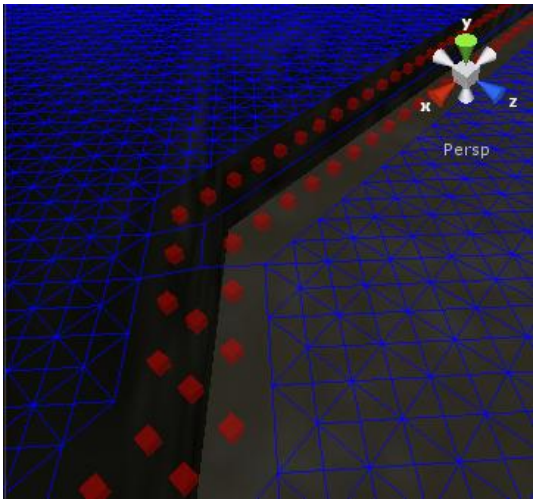

Bild 5.4 *(Rutnätet visualiserat med blåa linjer för sökbar yta)*

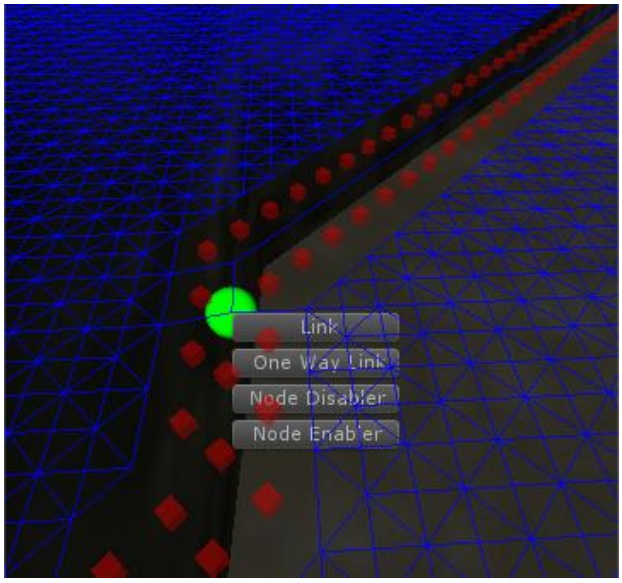

Bild 5.5 *(Grafiskt interface för att modifiera rutnätet)*

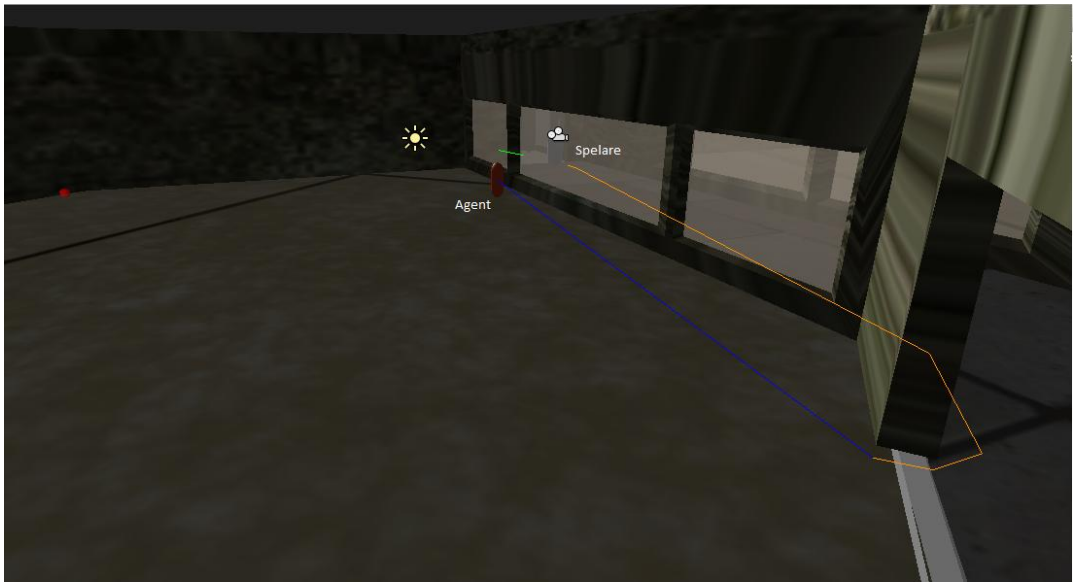

Bild 5.6 *(En agent med beräknad sökväg till spelaren visualiserad med en blå linje till närmsta vändpunkt och resterande sökväg med en orange linje. Den beräknade sökvägen är nu korrekt.)*

6 Planer för vidare utveckling

Arbetet som utförts i projektet har i sig varit en förberedande fas till utvecklingen av den artificiella intelligensen för Unknown. Tidigt i arbetet tog jag tillsammans med min medarbetare Viktor Högqvist-Nilsson fram en lista på handlingar som vi planerat att agenten ska kunna utföra. Jag hade då tanken att se till så att de tekniker som kommer att användas vid utvecklingen inte är begränsande på ett sätt som hindrar någon av handlingarna att implementeras. Efter att undersökningarna i kapitel 2 och 3 utförts var det dock uppenbart att ingen struktur för beslutsfattande begränsar vilka handlingar som kan utföras

och att en pathfindinglösning hanterar samma handlingar oberoende på vilken teknik som används. Därmed kan listan ses som irrelevant vid motivering till val av teknik och plugin men den kan fortfarande vara intressant för att visa vilken vidare utveckling som är planerad.

Planerade handlingar till vidareutveckling
En lista över handlingar som agenterna i Unknown planeras kunna utföra.

- Attackera spelare
- Detektering av spelare genom "synfält" och "hörsel"
- Gömma sig för spelaren
- Imobilisera spelaren
- Jaga spelaren
- Kasta saker
- Leta efter spelaren
- Patrullera mellan ett valbart antal punkter
- Skrika åt spelaren med ljud och gester
- Slå spelaren till marken
- Slå sönder fönster
- Ta sig igenom fönster
- Vakta ett objekt
- Vara overksam/strosa omkring
- Vänta i bakhåll för spelaren
- Öppna/ slå in dörrar

7 Slutsatser

Teknikerna som jag funnit i mina undersökningar är beskrivna i kapitel 2. För beslutsfattande beskrivs grundegenskaperna hos tillståndsmaskiner, beslutsträd och neurala nätverk. För tillståndsmaskiner och beslutsträd beskrivs även olika metoder för strukturering av dessa tekniker. För pathfinding innefattar teknikerna rutnät, POV-grafer och navmesher, vilka alla är olika datastrukturer som representerar spelets geometri. Sökning genom dessa datastrukturer utförs med sökalgoritmen A* som också förklaras i kapitel 2.

För utveckling i Unity har jag inte uppfattat att specifika anpassningar behöver göras för utveckling av artificiell intelligens jämfört med andra utvecklingsmiljöer. Undersökningen av implementerade lösningar i kapitel 3 innehöll ett flertal bra funktioner och idéer men de hade varit lika bra idéer oberoende av vilken utvecklingsmiljö som använts. Att tekniker för utvecklingen är generell gäller inte bara för utvecklingsmiljön. Det gäller i de flesta fall även för spelen som utvecklas. I kapitel 6 förklarar jag mer detaljerat

hur planerade egenskaper för agenterna inte påverkade varken valet av pathfinding eller beslutsfattningsteknik.

Beslutet av vilka tekniker att använda baserades alltså på generella egenskaper och hur väl olika tekniker och implementerade lösningar uppfyller dessa. Implementerade lösningar finns summerade i kapitel 3 och egenskaper jag ansåg positiva vid utvecklingen finns summerade i en lista i kapitel 4. I kapitel 4 motiverar jag också mitt val att använda mig av Behave och A* pathfinding och diskuterar för och nackdelar med olika tekniker.

Angående min frågeställning om möjlighet att återanvända kod mellan flera olika agenter är det en egenskap som finns i varierande omfattning i samtliga lösningar för både beslutsfattande och pathfinding. I beslutsfattande handlar det mycket om heuristik och möjligheten att bygga upp logiken i återanvändbara delar. I pathfinding och andra handlingar är det viktigt att kunna variera beteende genom data istället för genom kod för att förenkla återanvändning.

8 Terminologi

agent: En enhet med artificiell intelligens.
API: Application Programming Interface
artificiell intelligens: Intelligens simulerad av en dator.
code completion: Funktion som föreslår möjliga alternative för att avsluta ett ord vid programering.
code formatting: Funktion som formaterar koden efter standard automatiskt.
debug: Att hitta och rätta fel i vid programmering.
frame: En enskild bildruta som visas till exempel på en datorskärm. Mäts i FPS. (Frames Per Second)
konvex: Ett område där alla par av punkter kan bindas samman med en rät linje.
Monodevelop: En SDK för programmering i C#.
namespace: En abstrakt behållare av unika identifierare. Ett sätt att organisera kod.
pathfinding: Beräkning av den kortaste vägen från en punkt till en annan.
plugin: Ett program som inte körs fristående utan används som en del av ett annat program.
polygoner: En sluten yta uppbyggd av räta linjer.
runtime: Tiden då ett program körs.
script: Små program eller delar av program skrivna för att tolkas av en specifik utvecklingsmiljö, i det här fallet Unity.
Unity: En spelmotor. Se [27].
Unknown: Arbetsnamn för spelet som är under utveckling
utvecklingsmiljö: Ett eller flera program som tillsammans används för att utveckla nya program.

9 Referenser

9.1 Litteratur

1. Bourg, David M. & Seeman, Glenn. (2004). *AI For Game Developers*. O'Reilly.
2. Buckland, Mat, (2005). *Programming game AI by example*. Worldware Publishing, Inc.
3. DeLoura, Mark et al. (2002). *Game Programming gems*. Charles River Media.
4. Kirmse, Andrew et al. (2004). *Game Programming gems 4*. Charles River Media.
5. Rabin, Steve et al. (2002). *AI game programming wisdom*. Charles River Media

9.2 Internet

6. http://www.AI-blog.net/archives/000152.html
(Juni 2011) Utvecklare blog av Alt, Greg et al.
7. http://AIgamedev.com/
(Juni 2011) Artikel samling och forum.
8. http://aigamedev.com/insider/presentations/behavior-trees/
(Juni 2011) Video presentation med Champandard, Alex J.
9. http://AIgamedev.com/open/articles/bt-overview
(Juni 2011) Artikel av Champandard, Alex J.
10. http://aigamedev.com/open/articles/decorator/
(Juni 2011) Artikel av Champandard, Alex J.
11. http://AIgamedev.com/open/articles/hfsm-gist/
(Juni 2011) Artikel av Champandard, Alex J.
12. http://aigamedev.com/open/articles/selector/
(Juni 2011) Artikel av Champandard, Alex J.
13. http://aigamedev.com/open/articles/sequence/
(Juni 2011) Artikel av Champandard, Alex J.
14. http://altdevblogaday.org/2011/02/24/introduction-to-behavior-trees/
(Juni 2011) Blog inlägg av Knafla, Bjoern.
15. http://angryant.com/
(Juni 2011) Utvecklare blog av Johansen, Emil.
16. http://www.arges-systems.com/articles/274/unitysteer-2-2-released/
(Juni 2011) Utvecklare blog av Méndez, Ricardo J.
17. http://www.arongranberg.com/unity/a-pathfinding/
(Juni 2011) Utvecklare blog av Granberg Aron.
18. http://www.critterai.org/nmgen
(Juni 2011) Utvecklare blog av Pratt, Stephen.

19. http://digestingduck.blogspot.com/
(Juni 2011) Utvecklare blog av Mononen, Mikko.
20. http://forum.unity3d.com/threads/26030-unity-FSM
(Juni 2011) Forum inlägg från användare 1r0nM0nkey.
21. http://www.microsoft.com/download/en/details.aspx?id=7029
(Juli 2011) Nedladdningsplats för ”Csharp Language Specification”.
22. http://www.ogre3d.org/
(Juli 2011) Hemsida för Ogre
23. http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html (Juni 2011) Utvecklare blog av Amit
24. http://undergraduate.csse.uwa.edu.au/units/CITS4242/17-paths.pdf
(Juni 2011) Pdf till föreläsning vid The University Of Western Australia.
25. http://udk.com/
(Juli 2011) Hemsida för Unreal Development Kit
26. http://www.unifycommunity.com/wiki/index.php?title=Finite_State_Machine
(Juni 2011) Wiki post av Bianchini, Roberto Cezar.
27. unity3d.com
(Juni 2011) Hemsida för Unity
28. http://unity3d.com/support/community
(Juli 2011) Länksamling av Unity till sidor om gemenskapen kring Unity
29. http://unity3d.com/support/documentation/
(Juli 2011) Länksamling av Unity till sidor innehållande dokumentation av Unity
30. http://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/G-Path.pdf
(Juni 2011) Pdf till föreläsning vid Worcester Polytechnic Institute