Design och implementation av artificiell intelligens som finit tillståndsmaskin i spel

Kandidatavhandling i datavetenskap

Joel Sjöberg

Åbo Akademi

Fakulteten för naturvetenskaper och teknik

Handledare: Annamari Soini

4.4.2018

Referat

Implementering av artificiell intelligens i speldesign varierar från genre till genre. Att designa ett spel för en viss genre kräver olika problemlösningsmetoder beroende på den genre vi väljer. Artificiell intelligens och dess beteende i dessa genrer är inget undantag.

I detta avseende behandlar avhandlingen designen och implementeringen av artificiell intelligens som finita tillståndsmaskiner i spel samt jämförelsen mellan den teoretiska modellen och den praktiska. Terminologin definieras i den första delen av avhandlingen varefter vi ser på olika implementeringar av artificiell intelligens samt för och nackdelar med dessa implementationer. I slutet av avhandlingen presenterar vi en implementering av en hårdkodad finit tillståndsmaskin och kommer således till slutledningen.

Nyckelord: Finit tillståndsmaskin, artificiell intelligens, tillstånd, speldesign, genre

Innehållsförteckning

1. Inledning

Spel är stora, komplicerade program som kräver kunskap och talang att utveckla. Även sk. indiespel dvs. spel gjorda av små grupper människor eller spel med en minimal budget kan ta flera år att utveckla. Att utveckla spel innebär att handskas med flera olika utmaningar. Ett spel består alltså av olika system som definieras och skapas efter spelets behov. Exempel på dessa varierar från genre till genre; ett rollspel kräver oftast system som gör spelkaraktärer starkare efter att de klarat av olika utmaningar och pusselspel kommer att kräva ett sätt att kontrollera om en given utmaning är löst. Vilka system som behövs beror således på spelet man skapar, och detta stämmer också för artificiell intelligens.

Artificiell intelligens (kan förkortas AI) används på många olika områden idag. Den används för att utföra enkla uppgifter som människan finner tidskrävande. I detta avseende strävar artificiell intelligens efter att vara så effektiv som möjligt dvs. exekvera på ett sätt som sparar systemets resurser och kommer fram till lösningen så fort som möjligt. I spel används artificiell intelligens i övrigt på samma sätt men med en klar skillnad: Medan AI i spel vill spara resurser, likt AI i andra tillämpningar, kräver vissa spel att AI designas med klara brister[1]. Spel har förstås olika former av AI i sig, men spel använder ofta AI som en form av hinder eller utmaning. En spelare ska kunna se spelets beteende och utnyttja detta för att klara av utmaningar som skapats av spelets designer. I detta avseende får inte AI som är omöjlig att överträffas skapas.

Design är en viktig del av varje spel; den definierar spelupplevelsen och spelets mekanik. Ett spel kan ha flera olika designers beroende på spelets storlek, men i denna avhandling syftar designer på en person som, genom att definiera spelets mekanik, skapar utmaningar som är möjliga för spelaren att klara av. Konceptet om hinder i spel är för abstrakt för att koncentreras på i en avhandling, hinder skulle kunna vara en grop vilken spelaren måste hoppa över för att komma vidare. Men i denna avhandling syftar hinder på en form av datorkontrollerad motståndare.

För avhandlingens syfte lönar det sig att påpeka att alla spel är olika och ofta görs för en viss målgrupp. Detta är dock inte en garanti för att alla inom den målgruppen kommer att tycka om spelet. Vad som klassificeras som bra eller dålig design är beroende av vad spelskaparen vill åstadkomma med spelet och är således nästan omöjligt att objektivt bedöma om man spelar spelet utan skaparens kontext. Detta betyder att en designer sällan kan utgå från något exempel som bra eller dålig design, det är upp till spelarens egen åsikt om designen fungerar eller inte.

Avhandlingen är således en introduktion till teorin om tillståndsmaskiner, artificiell intelligens genom finita tillståndsmaskiner (kan förkortas FSM) och speldesign; samt en jämförelse mellan den teoretiska representationen och den praktiska implementeringen i spel. Det lönar sig här nämna att avhandlingen fokuserar på en viss typ av finita tillståndsmaskiner, ur en praktisk synvinkel kan hela spel definieras som tillståndsmaskiner och alla spelets delar lika så; ämnet är därför för brett för att ta upp i en kandidatavhandling. Istället kommer avhandlingen fokusera på tillståndsmaskiner som explicit designas för att utmana spelare och vilken sorts utmaning som passar ihop med spelets design i denna kontext. Samt presentera finita tillståndsmaskiner ur en teoretisk synvinkel och ge ett enkelt exempel för de som är obekanta med konceptet genom en implementation. Implementationen jämförs sedan med andra alternativa metoder för att utesluta hur effektiv FSM är i spel.

Det är också viktigt att påpeka att avhandlingens innehåll är likt Robert Siréns avhandling: "*Artificiell intelligens i datorspel* " från 2014 där stig sökning, AI och FSM presenteras samt dess roll i olika genrer. I jämförelse kommer avhandlingen att fokusera mera på FSM samt ge vidare insikt i hur FSM fungerar och implementeras.

2 Artificiell intelligens

Med tiden har spel blivit större och extremt krävande. Detta har lett till att arbetsmängden för varje skede i designprocessen har ökat och detta gäller speciellt för AI. När spelens storlek ökar så blir det svårare att programmera exakta algoritmer för att nå ett visst resultat. Detta stycke introducerar AI från en akademisk synvinkel för att sedan ta fram hur AI används i spel.

2.1 Akademisk artificiell intelligens

Artificiell intelligens kan spåras tillbaka i historien till långt innan datorer existerade som de gör idag. Enligt Russell och Norvig [2, 1.2.1] kan AI spåras tillbaka till Aristoteles (384 - 322 före vår tideräkning) som resonerade om system som kunde användas för att nå slutledningar. Russells och Norvigs bok [2] introducerar AI och hur olika fält har haft en inverkan på ämnet som t.ex. Filosofi, matematik, neurovetenskap med mera. Enligt Millington och Funge [3 s.5 - 7] kan akademisk artificiell intelligens delas in i tre olika skeden i historien: den tidiga eran (skrivs i boken: "*the early days*"), den symboliska eran och den moderna eran.

Under den tidiga eran var AI bara en idé om att skapa intelligens som kan jämföras med människans intelligens; en extremt inflytelserik person inom området är Alan Turing. I Turing's artikel *"Computing Machinery and Intelligence"* [4] föreslår han ett test som kan användas för att utesluta om en artificiell intelligens kan misstas för en människa. Testet görs på följande sätt: En förhörare ställer frågor till två personer där en av personerna är en dator som låtsas vara en människa. Om datorn lyckas övertala förhöraren att den är människa så kan man slutleda att datorn är lika intelligent som en människa. I artikeln jämför Turing AI med den mänskliga intelligensen; han anser att artificiell intelligens kan läras lika som barn lär sig från födseln.

Under den symboliska eran uppkom ett sätt att se på problemlösning genom algoritmer. Ett sätt att se på algoritmer är med hjälp av sk. symboliska system ("*symbolic systems*") där en algoritm kan delas in i två delar: en samling med information och ett sätt att använda informationen för att komma fram till en lösning eller mera information. Sättet informationen kan användas för att hitta ny information är genom sökning och detta leder till vad Millington och Funge kallar för "*the golden rule of AI*": Ju mera information om problemet algoritmen har, desto mindre sökning krävs; ju snabbare systemet kan söka efter en lösning, desto mindre information behövs. Olika exempel på symboliska system är t.ex. Stig-sökning, beslutsträd och tillståndsmaskiner.

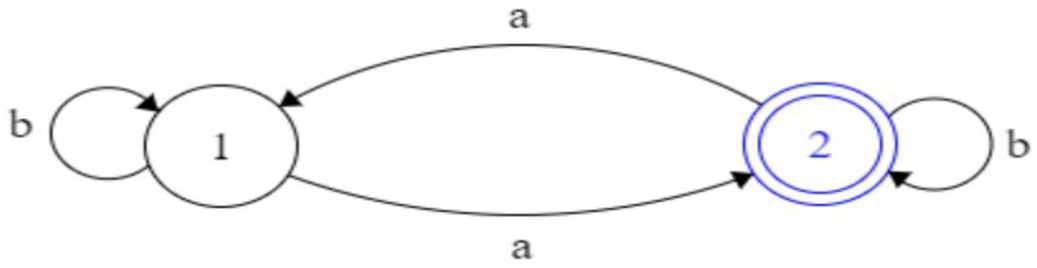
Den moderna eran skiljde sig från den symboliska eran genom att försöka implementera de ovannämnda symboliska systemen. Med fokus på detta blev AI-fältet mera fokuserat på effektiva lösningar än att försöka skapa mänsklig intelligens. Vad Millington och Funge menar är att AI är mest effektiv när det gäller att lösa specifika problem utan att försöka imitera den mänskliga intelligensen.

Den akademiska sidan av AI är viktig och utan dessa framsteg inom den akademiska forskningen skulle spel antagligen inte existera som de gör idag. De symboliska system som nämns i den symboliska eran har ofta använts i spel trots att akademisk AI skiljer sig från spel-AI.

2.2 Tillståndsmaskiner

En tillståndsmaskin är abstrakt definierad av olika tillstånd och övergångar från ett tillstånd till ett annat genom att acceptera indata. Ett tillstånd kan enligt Kozen [5] anses vara en representation av maskinen själv. Genom att definiera tillstånd och övergångar mellan dem kan man skapa automater som har olika funktioner t.ex. figur 1 som endast accepterar strängar som har ett ojämnt antal "a" i sig vilket betyder att den kan garantera att en sträng uppfyller detta villkor genom att ta strängen som indata. Den finita tillståndsmaskinen består enligt Sipser [6 s.35 - 41] formellt av fem olika saker:

(Σ) ett alfabet som innehåller alla möjliga indata, (S) en mängd med alla möjliga tillstånd, (s0) ett bestämt tillstånd varifrån maskinen börjar, (δ) en funktion som möjliggör övergången från ett tillstånd till ett annat och (F) en mängd med slutliga tillstånd där exekveringen kan sluta (en delmängd av S). Att en sträng accepteras betyder att maskinen når ett sluttillstånd i F genom att läsa in ett tecken åt gången från strängen. Om sluttillståndet inte nås på detta sätt nekar maskinen den strängen. Figur 1 är ett exempel på en finit tillståndsmaskin som kan definieras av följande värden: $\Sigma : \{a, b\}$, $S : \{1, 2\}$, $s0 : \{1\}$, $\delta : \Sigma \times S \rightarrow S$, $F : \{2\}$.



Figur 1. En finit tillståndsmaskin som endast accepterar strängar som innehåller ett ojämnt antal "a", s0 = {1}, F = {2}

I figur 1 är siffrorna de numrerade tillstånden som maskinen kan existera i. Pilarna är övergångarna mellan tillstånden och bokstäverna är indata från Σ. För att maskinen ska fungera som definierat måste det första tillståndet vara {1}, om dess funktion var att acceptera de strängar som innehåller ett jämnt antal a:n skulle det första tillståndet vara {2} som även är sluttillståndet eller "det accepterande tillståndet". Mera teori om maskinens uppbyggnad ges nu.

Alfabetet Σ består av den tillåtna mängden indata, i detta fall bokstäverna "a" och "b", maskinen accepterar endast strängar bestående av dessa. De strängar som maskinen accepterar är dock bara en delmängd av alla möjliga finita strängar som kan skapas från Σ. Formellt kan alla möjliga strängar beskrivas med notationen Σ* som är en oändlig mängd med finita strängar; i detta fall bestående av{ϵ (den tomma strängen), a, b, ab, bb, aab,...}. Då kan en möjlig sträng x som maskinen accepterar höra till den oändliga mängden Σ* givet att x innehåller ett ojämnt antal a:n, vilket kan skrivas med notationen: $(x \in \Sigma^* \mid$ x innehåller ett ojämnt antal a:n). En sträng som skulle accepteras är t.ex. "*babaa*" eller "*aaabb*" medan "*abab*" inte skulle accepteras.

Maskinen i figur 1 är även kategoriserad som en deterministisk finit tillståndsmaskin vilket betyder att givet indata från Σ bestående av ett tecken kan maskinen endast nå ett tillstånd från ett annat ty dess beteende är deterministiskt, detta kan förutspås givet indata. En non-deterministisk finit tillståndsmaskin är i jämförelse oförutsägbar då det kan finnas flera övergångar mellan tillstånden med samma indata från Σ vilket betyder att det finns en mängd möjliga tillstånd att nå från ett givet tillstånd, givet indata. Det första tillståndet s0 är en delmängd av S sådant att $s0 \in S$. I detta fall definieras s0 med tillståndet {1}. Maskinen befinner sig i detta tillstånd tills den äntligen får ett "a" som indata vilket gör att tillståndet byter från {1} till {2} osv.

För att förstå varje möjlig övergång som maskinen i figur 1 har måste övergångsfunktionen definieras. Övergångsfunktionen δ definieras som $\Sigma \times S \rightarrow S$ vilket kan läsas som: givet indata från alfabetet Σ och ett tillstånd från mängden S kommer maskinen att nå ett av tillstånden i S. Övergångsfunktionen kan enkelt visualiseras med hjälp av tabell 1.

Indata/Tillstånd	1	2F
a	2	1
b	1	2

Tabell 1. Övergångsfunktionen δ med alfabetet Σ och tillstånden i S

Övergångsfunktionen beskriver maskinens funktion för varje möjlig indata från Σ. Tabeller som beskriver funktionen (tabell 1) kan skrivas genom att se på varje övergång i maskinen och dokumentera vilken indata som leder till vilket tillstånd. Tabellens storlek beror på antalet övergångar och tillstånd som definieras i maskinen. En utvidgad-övergångsfunktion kan användas för att slutleda om en sträng kan nå sluttillståndet dvs. istället för att testa ett tecken från Σ åt gången är det möjligt att ta en finit sträng från Σ* och se om sluttillståndet nås genom att mata in den i den utvidgade-övergångsfunktionen. I detta fall är sluttliståndet F en mängd som består av ett tillstånd: {2}.

Detta är teorin om finita tillståndsmaskiner. Det finns många olika kategorier som dessa kan delas in i som t.ex. deterministiska, non-deterministiska , hierarkiska och hårdkodade tillståndsmaskiner [3. S 318 - 330].

Teorin som presenterats i detta stycke är dock endast grunden för dessa, teorin går mycket djupare än avhandlingen kan presentera. Men givet teorin och de formella definitionerna som kommit ur den akademiska världen blir konceptet klarare då de studeras i spel.

2.3 Artificiell intelligens för spel

Datorspel behandlar ständigt en hel del information, därför är det inte en överraskning att spel använder flera olika metoder och system som uppkommit inom den akademiska forskningen för att effektivt behandla den informationen. Det finns även fall där spel har använts för att undersöka människan själv [7]. I undersökningen ville man veta hur människan lär sig snabbare än artificiell intelligens att spela spel och varför. Slutledningen som artikeln drar är att människan kan använda tidigare kunskap för att lättare kunna begripa spelets funktion och klara av utmaningarna spelet ger medan artificiell intelligens måste lära sig den kunskapen från grunden när den spelar spel med hjälp av sk. "*Deep learning*". Detta innebär att medan spel har kunnat utvecklas med AI tack vare den akademiska världen så kan spel också användas för att forska i olika fält.

Nareyek[1] menar att det finns en stor skillnad mellan akademisk och spel-AI, nämligen syfte; artificiell intelligens i spel designas oftast för att vara rolig att spela mot eller med. I detta avseende designas artificiell intelligens i spel inte för att vara så effektiv och snabb som möjligt utan för att underhålla spelaren. När Nareyek skriver om AI i sin artikel så avser han en artificiell intelligens som styr NPC:er. En NPC (från engelskans "*Non-playable character*") är en karaktär eller figur i spel som inte direkt kontrolleras av spelaren. Dessa finns i spel av olika skäl beroende på spelets genre, t.ex. ett rollspel brukar ofta ha NPC:er som karaktärer i världen som spelet innehåller. Dessa brukar kunna ge information om världen och berättelsen som utspelar sig i spelet. En annan form av NPC är fiende karaktärer som i t.ex. rollspel försöker ta livet av spelaren eller förhindra spelaren från att spela vidare.

För att en NPC ska fungera korrekt i ett spel krävs det olika metoder beroende på den figurens uppgift. I spelet Pac-man används t.ex. tillståndsmaskiner för att ge varje spöke ett visst beteende beroende på dess tillstånd [3, s.7, 19 - 20]. Spöken försöker komma till spelaren i labyrinten på olika sätt vilket ger varje spöke en egen personlighet. Ett spöke kan försöka komma direkt till spelaren medan ett annat spöke helt enkelt går till en slumpmässig plats i labyrinten. Tillståndsmaskiner fungerar bra i dessa fall när spelet behöver olika sorters NPC:er som har olika beteenden. I detta avseende tillåter tillståndsmaskiner programmerare att koda olika beteenden åt varje spöke. På detta sätt kan ett spöke tilldelas en tillståndsmaskin som har en egen metod för att traversera labyrinten och utmana spelaren. När spelaren äter ett piller så byter varje spöke sitt tillstånd från att söka efter Pac-man till att fly från honom. Tillståndsmaskiner gör det möjligt att se på varje NPC i spelet som en samling med tillstånd som varje figur byter emellan dvs. det blir lätt att kategorisera varje spökes beteende genom att klassificera varje beteende som ett tillstånd denne kan existera i.

2.4 Finita tillståndsmaskiner i spel

Det finns olika sätt att skapa beteende i spel; beteende kan vara allt från att kontinuerligt gå mot vänster till att aktivt försöka reducera spelarens liv med hjälp av de handlingar som designen tillåter. För att skapa beteende kan programmeraren hårdkoda beteendet direkt in i spelets NPC:er om hen vet att varje NPC endast kan vara i ett tillstånd som tillåter ett visst beteende. För att tillåta flera olika beteenden för en NPC beroende på dess tillstånd kan tillståndsmaskiner användas för att lättare designa dessa. De olika tillstånd som en tillståndsmaskin består av kan representeras av de beteenden en NPC blir tilldelad. Denne byter nuvarande beteende baserat på indata från spelets olika delar, dess nuvarande tillstånd och spelarens handlingar t.ex. en NPC kan söka efter spelaren medan dess liv är större än tre, om livet går under tre byter tillståndet till att söka efter skydd, om spelaren är synlig för NPCn så kan den skjuta spelaren osv. I spel lönar det sig att bygga en tillståndsmaskin med ett visst antal tillstånd för att inte använda för mycket arbetsminne dvs. mängden av de möjliga tillstånd som en karaktärs maskin består av bör vara ändlig och så liten som möjligt för att ta upp så lite minnesresurser som möjligt.

En programmerare har ofta begränsade resurser att jobba med när hen skapar AI skriver Millington och Funge [3 s. 25]. Att överskrida de tillåtna resurserna betyder att spelet kommer att ha problem att köras stabilt och hålla en koncis uppdateringsfrekvens. En finit tillståndsmaskin (även kallad "finite state automata" på engelska) är en tillståndsmaskin vars mängd med möjliga tillstånd är ändlig eller "finit". En FSM har i teorin inte tillgång till minne, övergångsfunktionen reagerar bara på en bit av indata åt gången. I spel kan detta ignoreras då varje NPC har möjligheten att spara värden i variabler och andra datastrukturer samt ta indata från spelet och spelarens handlingar. En spelprogrammerare kan i detta fall bortse från minnes begränsningarna av en formell tillståndsmaskin och implementera dem i spel.

Millington och Funge tar upp sk. hierarkiska och hårdkodade tillståndsmaskiner [3. s 316 - 330]. En hårdkodad tillståndsmaskin är praktiskt en maskin som kontrollerar nuvarande tillstånd genom att testa olika booleska uttryck. Maskinen byter alltså tillstånd genom att testa om olika fakta är sanna vilket leder till att tillståndet övergår till ett annat t.ex.

```
// exempel på tillståndsövergångar i hårdkodad FSM
if (hungry):  state = EAT;
if (injured): state = TAKECOVER;
```

En hårdkodad tillståndsmaskin använder en liknande if-struktur som enkelt kan kontrollera nuvarande tillstånd och byta mellan dem. En hierarkisk tillståndsmaskin är en maskin som kan bestå av två eller flera tillståndsmaskiner. På detta sätt kan en tillståndsmaskin existera i två eller flera tillstånd samtidigt där ett tillstånd har högre prioritet över det andra t.ex. om en karaktär är i tillståndet "hungrig" så söker denne efter mat. Baserat på ett annat tillstånd som har högre prioritet tilldelad "i fara" så kommer denne att söka efter skydd istället för att söka efter mat. När tillståndet övergår från "i fara" till "i säkerhet" kan tillståndet "hungrig" igen ta prioritet. Enligt Millington och Funge [3 s. 311] finns det så många sätt att implementera tillståndsmaskiner i spel så att det är svårt att fastställa en definitiv implementation av dem. Alfabet och övergångsfunktion är svårare att definiera i spel där indata inte nödvändigtvis ges beroende på vad som händer i spelet och beteenden är inte garanterade att leda till ett nytt tillstånd (vilket är liknande de non-deterministiska tillståndsmaskinerna).

3 Speldesign

Enligt Ernest Adams kan ett spel ha flera olika personer som antar roller i spelutvecklingen [8, s. 54 - 56]. Dessa roller inkluderar bla. programmerare, författare, artister och designers etc. (de flesta beroende på vilken sorts spel det är frågan om). Alla dessa roller påverkar olika delar av spelet en artist ansvarar för de grafiska aspekterna i spelet, författare skriver berättelsen i spelet (om spelet skall ha en sådan) och programmerare bygger ihop spelets logik. Medan alla dessa roller fungerar för att skapa innehåll till spelet kan designerns uppgift anses vara att använda innehållet för att utmana spelaren. Designen beror såklart på vilket spel som utvecklas, men en vanlig tankegång i spelutveckling är att göra spel kul.

För att skapa ett spel som spelaren vill spela föreslår Adams vad han kallar "*player-centric design*"[8 s.32]. Med denna tankegång föreslår Adams att designern tänker på hur spelaren bemöter spelet. En designer bör se på spelet ur en spelares synvinkel för att förstå spelupplevelsen och kunna utvidga det med mekanik som passar spelets behov. Det finns dock missförstånd som en designer måste undvika i spelutvecklingen. Vanliga fel som Adams tar upp är t.ex Passionen en designer har för att göra spel åt sig själv då hen tror att flera konsumenter är intresserade av spelet på basis av sitt eget intresse. Vad som är viktigt i detta fall är att vara professionell dvs. att sälja spel innebär att designa ett spel åt en viss målgrupp och se förbi sina egna preferenser. Naturligtvis kan designern göra spelet åt sig själv, men ur en ekonomisk synvinkel är det bäst att tilltala en så stor målgrupp som möjligt. Det andra missförståndet är att spelaren är designerns motståndare. Detta kan leda till att spelets utmaningar görs avsiktligt svåra för att förlänga spelupplevelsen; något som Adams avråder då det är viktigt att ge spelaren en ärlig chans.

Om design kan anses vara sättet att skapa utmaningar med innehåll så kan det anses vara en del av designerns uppgift att assistera i utvecklingen av spelets olika delar. Att skapa delar av spel med en viss funktion är att skapa spelets mekanik. Mekanik definierar hur spelet fungerar; på många sätt kan mekanik ses som verktyg som ska användas för att klara av spelets utmaningar. Design är att skapa utmaningar som kan avklaras med hjälp av mekaniken för att göra spelet roligt.

Många spel går ut på att besegra en motståndare som är styrd av datorn; motståndaren är på en abstrakt nivå en AI som designas för att utmana och eventuellt segra över spelaren. Det som klassas som AI i spel i denna kontext kan vara allt från hårdkodade procedurer till avancerade system. Spelet Pong, som släpptes på Atari-konsolen år 1972, har en AI som kontrollerar paddelns horisontella position för att möta bollens horisontella position. För att göra datorn mera människolik och ge spelaren möjligheten att vinna skippar datorn var åttonde speluppdatering[9, s.40]. I Pong skulle det vara möjligt att designa en motståndare som inte går att besegra genom att fästa datorns paddel på bollens horisontella position och lämna bort hastighetsbegränsningen. Detta är inte bra för ett spel eftersom datorn nu är oövervinnelig och spelar perfekt varje gång; spelaren kan således inte vinna. Begränsningarna krävs för att spelaren ska ha en ärlig chans att besegra datorn.

Adams bok [8] presenterar många olika uppgifter som ingår i spelutveckling. För att sälja ett spel måste varje del av spelet vara välgjort; detta stämmer även för AI. Detta betyder att projektet kräver speciell AI beroende på genren; den kan användas för många olika saker i spelets design och för att ge spelet en speciell funktion. Några områden där AI är användbar i spel är enligt Adams [8 s.15] strategiskt beteende dvs. Möjligheten för spelet att förse spelaren med utmaningar genom att använda beteende som utmanar spelaren genom att ta spelets tillstånd i beaktande. Exempel på detta skulle vara så kallade RTS eller "*Real time strateg*y" spel som Civilisation-serien där en AI lika som spelaren tar hand om sin egen stad. För att göra detta måste den artificiella intelligensen i spelet kunna hantera anskaffning nya resurser, förhandla med och/eller förklara krig mot spelaren eller en annan AI i spelet och fatta andra strategiska beslut på basen av dess nuvarande tillstånd. På detta sätt får spelaren motståndare och allierade i spelet vilket gör spelet mera utmanande.

Millington och Funge presenterar olika sorters system som artificiell intelligens kan designas med [3 s. 807 - 840]. Enligt dem är det ideala sättet att designa AI att först planera hur spelets olika delar ska fungera för att fastslå de olika system spelet ska använda och sedan implementera dessa. Detta är dock inte alltid möjligt då hinder kan komma från många olika platser i spelutvecklingen; exempel på ett sådant hinder är tidsbegränsning och budget vilket kan orsaka stress och att kod snabbt skrivs för att få ett beteende i spelet (vare sig det är ett idealt beteende eller inte).

Några saker som Millington och Funge tar upp i design av AI är rörelse, nämligen hur NPC:n ska förflytta sig från punkt till punkt och om dess rörelse ska påverkas av andra figurers rörelse. Det är också viktigt att förstå hur den kan fatta beslut i spelet och hurudan data den behöver för att fatta dessa beslut och om den ska kunna jobba tillsammans med andra NPC:er för att nå ett visst mål.

I skjutspel som kan förkortas FPS från engelskans "*first-person shooter*" och TPS "*third-person shooter*" beroende på perspektivet spelaren har, används bland annat algoritmer för rörelse (för att navigera i spelvärlden), beslutfattning (t.ex. för att gömma sig i skydd eller skjut) och uppfattning (hur mycket information som en AI har tillgång till). I sk. plattformsspel (från engelskans "*platformer"*) har AI ibland mönster i sitt beteende för att låta spelaren utnyttja dessa. Plattformsspel har liksom FPS-spel navigering för att kunna röra sig i den värld de sätts in i.

I körspel fokuserar AI mest på rörelse och navigering för att tävla mot spelaren. Navigering är inte nödvändigtvist obligatoriskt för ett körspel då en bil kan fästas vid vägen och helt enkelt följa den. Navigering kan sedan tillämpas i öppna spel med vägar som skiljer sig eller spel där bilar kan köra utanför vägar. Inlärning är också ett sätt att skapa artificiell intelligens. Modellen som Millington och Funge presenterar är bl.a. Artificiella neurala nätverk, vilket låter en intelligens testa sig fram genom att ta indata, försöka agera på de indata och lära sig av sina misstag för att bli mera effektiv. För att bokstavligen lära artificiell intelligens att spela körspel kan även neurala nätverk användas som Togelius et al. 2007 [10] visar med spel som t.ex. Forza Motorsport som använder neurala nätverk för att lära sig hur spelaren kör och agera på basen av det.

Design är utmanande att få rätt och kräver ofta noggrann planering och implementering för att få spelaren att förstå hur spelet fungerar. Om designen inte gör spelet roligt eller intressant kommer de andra delarna att misslyckas, sättet som spelet ser ut eller hur spelet låter har inte lika stor inverkan på spelets mottagande som hur spelet spelas. Att designa AI som ger spelaren utmaningar och möjligheter i ett spel är således extremt viktigt.

4. Implementering Av FSM

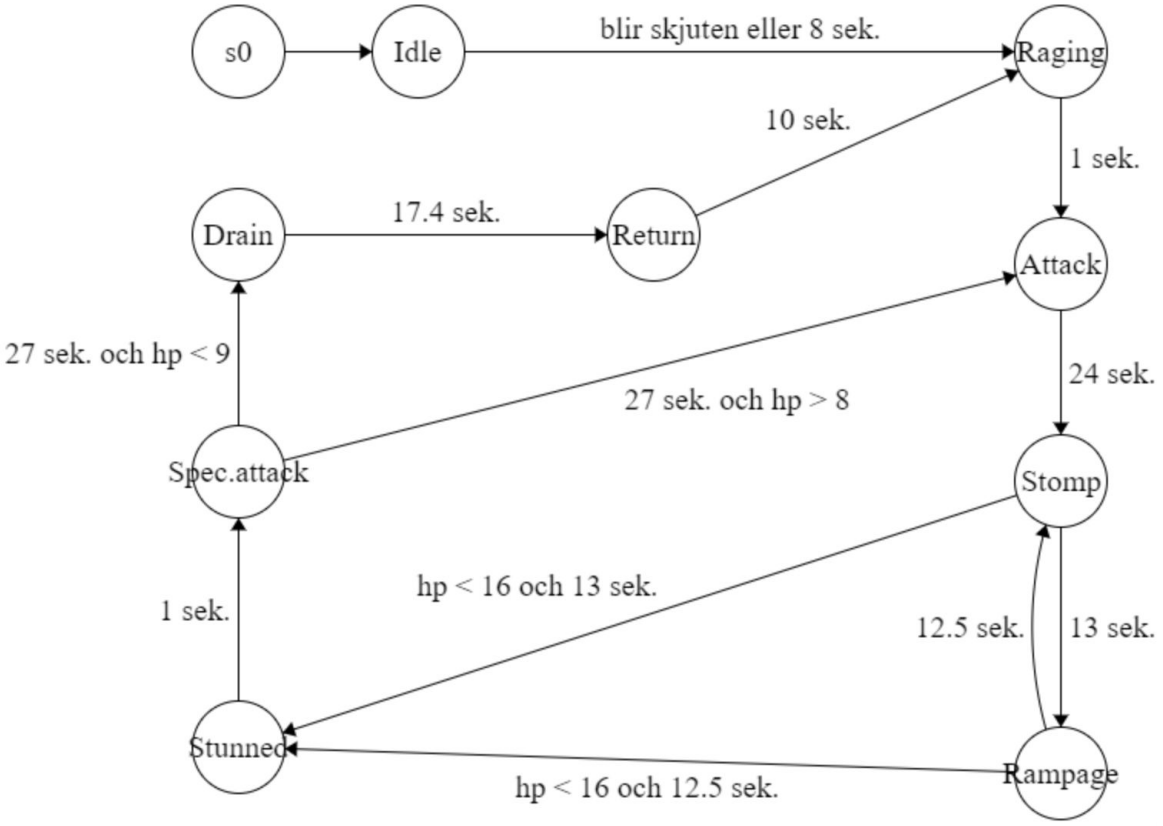
För att tillämpa teorin om tillståndsmaskiner och design ges nu ett exempel på en implementation av en finit tillståndsmaskin från ett spel som heter *Missfortune* vilken jag själv har skapat. Spelet är en "*top-down shooter*" vilket är en genre där spelaren har tillgång till ett skjutvapen och kan skjuta fiender för att undvika och eventuellt döda dem. I spel tillhörande genren visas spelvärlden och spelarens avatar uppifrån-ner därav "*top-down*". I spelet är målet för spelaren att utforska en labyrint som skapas av en slumpgenerator och hitta ammunition till sitt vapen för att komma till en sk. boss (hädanefter fiende); Efter att spelaren har besegrat fienden är spelet slut. Spelet använder sig främst av två dimensioner vilket betyder att spelaren kan röra sig upp, ner, åt vänster och åt höger. Muspekaren används för att rikta vapnet och vänster musknapp används för att skjuta ammunition som spelaren samlat; det finns även en spurt knapp som spelaren kan använda för att hoppa åt det håll som denne rör sig mot.

Spelet är skapat i spelmotorn Unity och all kod är skriven i C#, en länk till spelets källkod och övriga tillgångar finns tillgänglig i bilaga 1. Tillståndsmaskinen i fråga vars implementation ska diskuteras i detta stycke är den som tillhör fienden och är vald för att den är den största som är tillgänglig i spelet.

4.1 Ett praktiskt exempel

För att designa fienden bör dess olika tillstånd definieras, de tillstånd som den slutliga versionen använder visas i bilaga 2. Tillstånden består av en enum (vilket är en förkortning på engelskans "*enumeration"*) som innehåller nio olika tillstånd, på detta sätt kan tillståndet kontrolleras genom att skriva t.ex. *bossState.idle* för att kontrollera om fienden är i idle-tillståndet. I början av designprocessen definieras tillstånden genom att planera vad fienden kommer att göra i spelet. Strukturen på beteendet som fienden har beskrivs i figur 2 som visar övergångarna mellan alla beteenden och dess kriterier.

Tanken är att ge fienden ett beteende som upprepas genom att använda samma tillstånd flera gånger tills endera spelarens eller fiendens *health* värde når noll.



Figur 2. Den finita tillståndsmaskinen som visar fiendens olika tillstånd samt övergångar mellan tillstånden och deras kriterier (sek. läses sekunder har passerat).

Majoriteten av tillstånden som fienden kan existera i är attacker som fokuserar på att reducera spelarens liv. Övergången från tillstånd till tillstånd kontrolleras huvudsakligen av tiden tillståndet varit aktivt t.ex. fienden existerar i Attack-tillståndet tills tjugofyra sekunder har passerat vilket leder till att det nuvarande tillståndet byts till Stomp-tillståndet. Medan varje tillstånd är begränsat med en viss tid finns det även extra predikat som har större inverkan på övergången. Som exempel utförs Stomp i exakt tretton sekunder och byter sedan till Rampage men om tiden passerat och *health*-variabeln är under sexton kommer ett nytt tillstånd att nås. På detta sätt introducerar fienden nya beteenden åt spelaren ju längre striden pågår.

Teorin om finita tillståndsmaskiner och implementeringen av dessa i spel skiljer sig i detta avseende. Det som styr övergången från tillstånd till tillstånd är inte indata ur ett visst alfabet utan varje tillstånd har sparat en egen tid under vilken tillståndets metod utförs.

Kontrolleringen av tillstånd sker i bilaga 3 där switch-satsen kollar varje tillstånd i "*bossState"* och kallar dess respektive metod för att utföra fiendens beteende. Tillståndens design tillåter endast ett aktivt tillstånd; trots detta finns det även en if-sats i bilaga 3 som kontrollerar om spelarens eller fiendens *health*-variabel är noll. Om detta är fallet så kommer spel-objektet som koden är fäst vid att göras inaktiv vilket leder till att koden inte längre exekveras. På detta sätt är tillståndsmaskinen hierarkisk[3] ty *health*-variabeln är en indikator på vare sig figurerna är aktiva spelobjekt eller inte. Således har maskinen en hierarkisk struktur genom att ha ett odefinierat alarmtillstånd (ett tillstånd med prioritet över de andra) genom if-satsen. Denna maskin är då en sk. hårdkodad finit tillståndsmaskin eftersom varje tillstånd kontrolleras för att exekvera rätt beteendemetod.

Varje tillstånd har en metod som definierar det beteende fienden har i varje tillstånd. Ett exempel på detta visas i bilaga 4, "*Attacking()*"-metoden kallas när tillståndet övergår till attack-tillståndet från raging-tillståndet. Varje beteendemetod har en viss tid under vilken den exekveras (sparas i duration-variabeln) och består av två if-satser varav en är en if/else-sats. Den första if-satsen används när metoden först kallas, timer-variabeln är noll och inkrementeras med tiden mellan varje skärm-uppdatering medan beteendemetoden utförs. I if/else-satsen utförs beteendet medan timer-variabeln är mindre än eller lika med duration-variabeln. När {*timer* > *duration*} så utförs koden i else-satsen vilket reducerar timer-variabeln till noll och överför nuvarande tillstånd (state) till nästa enligt figur 2 vilket gör att koden i bilaga 2 kallar en annan beteendemetod.

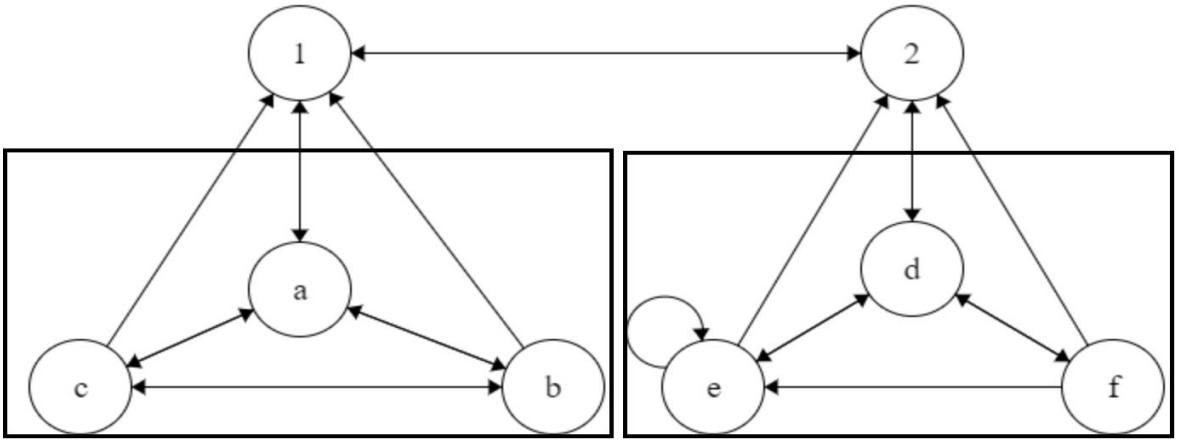
Fiendens tillståndsmaskin i spelet tjänar sitt syfte och fungerar bra då den endast har ett litet antal tillstånd och övergångar. Med den kan fienden ha olika beteenden beroende på hur mycket skada spelaren har lyckats åstadkomma. Beteendemetoderna följer alla samm struktur och kan innehålla mångsidiga beteenden i denna struktur.

Tack varen den hierarkiska struktur maskinen har i bilaga 3 kan maskinen köras så länge som möjligt utan att nå ett sk. Sluttillstånd där exekveringen slutar. Istället görs fienden helt enkelt inaktiv om den eller spelaren dör. Trots detta är modellen inte perfekt eftersom det finns några problem med implementationen och modellen själv.

4.2 Problem med implementeringen

Millington och Funge presenterar några styrkor och svagheter med den hårdkodade modellen [3 s. 318]. Modellen är lätt att skapa och förstå då den följer en relativt trivial struktur (som exempel switch-satsen i bilaga 3). Dock ju större de är blir de svårare att upprätthålla. Maskinen i figur 2 är en relativt liten maskin. Den består av tolv övergångar och nio tillstånd varav ett av dem endast används när spelet börjar (idle-tillståndet). Om nya tillstånd måste inkluderas i modellen efter implementation så skulle detta innebära omdefiniering av övergångarna mellan tillstånden där det nya tillståndet ska tillkomma. Idealt planerar designern hur fienden ska fungera och definierar alla dess tillstånd innan tillståndsmaskinen kodas och läggs till i spelet men detta är ofta inte möjligt då maskiner måste testas i spel. För att lägga till ett tillstånd mellan *stomp* och *rampage* i figur 2 måste deras beteende metoder modifieras genom att tilldela ett nytt värde åt *state*-variabeln i else-satserna när metodernas tid är slut. Ett nytt tillstånd måste definieras i bilaga 2, beteende metoden för det nya tillståndet måste definieras och sedan läggas till i switch-satsen i bilaga 3. Det nya tillståndet är nu en del av maskinen och maskinens komplexitet ökar ju mera tillstånd som inkluderas.

Maskinens alarmtillstånd i if-satsen (bilaga 3) har prioritet över själva maskinen i switch-satsen. Att inkludera alarmtillstånd som har prioritet över andra tillstånd på detta sätt kommer att kräva flera if-satser om designen har behov av andra alarmtillstånd. På detta sätt kan koden bli fylld med if-satser vilket gör koden svår att hantera på samma sätt som antalet tillstånd i switch-satsen kan göra maskinen svår att hantera och upprätthålla. För att framhäva den hierarkiska strukturen föreslår Millington och Funge [3 s.318 - 331] att skapa tillståndsmaskiner i lager dvs. en maskin skapas inne i en annan som i figur 3.



Figur 3. En hierarkisk tillståndsmaskin som kan övergå till alarmtillstånd vid behov

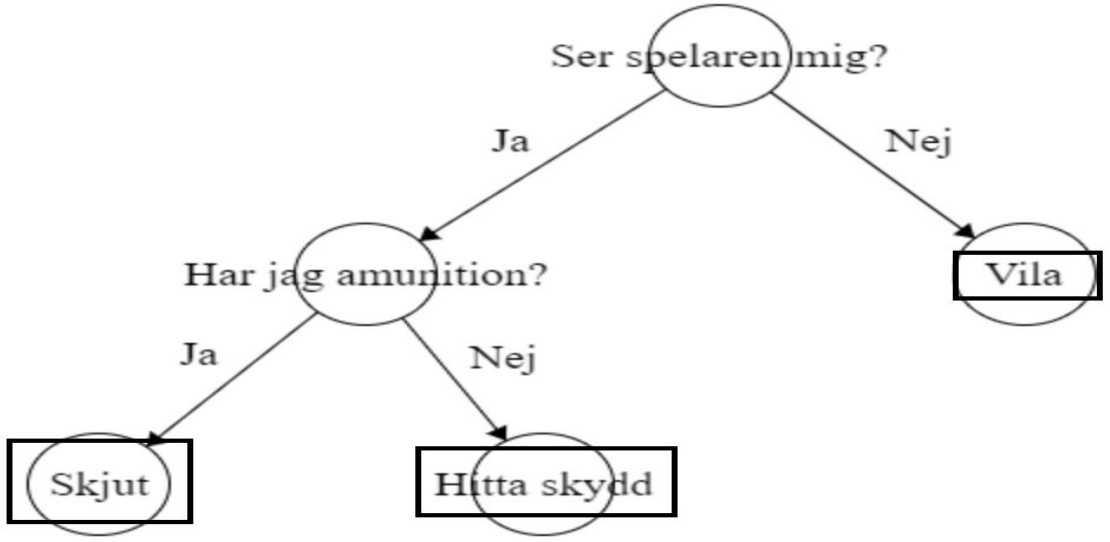
Figur 3 innehåller en hierarkisk tillståndsmaskin som består av en maskin med två tillstånd där varje tillstånd fungerar som en länk till varsin tillståndsmaskin. I vardera maskin har varje tillstånd en övergång till ett av de alarmtillstånd som innehåller den maskinen ({1}, {2}). Detta betyder att oavsett tillstånd kommer maskinen alltid ha möjligheten att hoppa ut ur en maskin och in i en annan. Tillstånden kontrolleras på följande sätt: kontrollera om övergång mellan tillstånd {1} och {2} aktiveras, om inte så gå in i det tillståndet av dessa som är aktiv och fortsätt exekvering därifrån. Som exempel kan maskinen existera i tillståndet [1, c] vilket gör att maskinen utför beteendet i tillstånd c, om övergången från {1} till {2} aktiveras så fortsätter exekveringen från tillstånd [2, d]. Idealt så sparas tillståndet [1, c] vid övergången så att exekvering kan fortsätta därifrån om övergången från {2} till {1} aktiveras.

Ett annat problem med maskinen i figur 2 är den förutsägbara strukturen. Att spela mot fienden länge nog kommer att göra maskinens mönster tydligt för spelaren vilket kan göra spelupplevelsen tråkig och ointressant. En trivial lösning på detta problem skulle vara att ge maskinen en chans att hoppa till olika tillstånd slumpmässigt. Således skulle fienden alltid ha en chans att överraska spelaren genom att bryta upprepningen. Maskinen är dock designad med upprepningen i åtanke, det är tänkt att spelaren ska kunna se och lära sig hur fienden agerar för att effektivt kunna besegra den lika som artificiell intelligens i plattformsspel som Millington och Funge presenterar [3 s.819]. Detta betyder dock inte att upprepningen inte är ett problem bara att det är ett medvetet val, men för vissa spelare kan aktiv och frekvent upprepning vara irriterande.

4.3 Alternativa metoder

Givet problemen ovan är det tydligt att metoden förekommer med sina egna för- och nackdelar. Det förekommer ofta problem från olika håll i varje modell; modellen kanske inte är implementerad på rätt sätt eller så har den begränsningar som en designer måste undvika på något sätt genom att designa maskinen så att begränsningarna inte påverkar spelet. För att förstå hur bra en given modell fungerar i ett spel lönar det sig att utforska andra alternativa modeller och jämföra dessa för att sedan utesluta vilken modell som bäst tillfredsställer spelets behov. Finita tillståndsmaskiner kommer som sagt i olika former i spel; det är därför svårt att fastställa en definitiv version av dessa som fungerar optimalt i spel när en annan version av modellen kanske fungerar bättre. Millington och Funge [3 kap. 5 "*Decision Making*"] presenterar alternativa metoder till finita tillståndsmaskiner samt sätt att utvidga tillståndsmaskiner i spel på olika sätt. Exempel på dessa är t.ex. beslutsträd och "*fuzzy logic*" samt deras kombinationer till tillståndsmaskiner.

Beslutsträd är en modell som abstrakt består av beslut och handlingar. Beslut i dess simplaste form är en fråga och skrivs som en beslutsnod i träddiagrammet. Beroende på svaret på den fråga som noden ställer kommer trädet att övergå till en annan nod som är ett barn till den föregående noden. Figur 4 visar ett simpelt beslutsträd.



Figur 4. Ett beslutsträd med två beslutsnoder och tre handlingar

Trädets "*löv*" dvs. den sista rutan på en gren är handlingar som utförs då de noder som leder till handlingen aktiveras. Trädets beslutsnoder kan ställa frågor som svaras med endera "jo" eller "nej" t.ex. "är talet i x mellan 10 och 15?" eller "har du mindre än två liv kvar?". Millington och Funge ger ett exempel på en sammanslagning av tillståndsmaskiner och beslutsträd [3 s.331] som är möjlig genom att byta ut övergångarna i maskinen med beslut vilket möjliggör mera sofistikerade beslut.

I jämförelse med beslutsträd och tillståndsmaskiner som har deterministiska övergångar introducerar sk. *"fuzzy logic"* grad av slumpmässighet i dessa övergångar. En simpel förklaring av vad fuzzy logic är för något är att se på logik. Logiska värden som sant och falskt kan representeras av siffrorna ett och noll. Ett logiskt predikat är fakta som endera är sant eller falskt. Fuzzy logic ser på detta inte som strikt sant eller falskt men istället som en del värden mellan noll och ett t.ex. Istället för att tilldela sant åt predikatet {det regnar} kan det tilldelas ett värde mellan noll och ett för att representera en viss sannolikhet som exempel 0.7 [3 s. 371]. Denna teori kan även kombineras med tillståndsmaskiner för att skapa sk. *"fuzzy state machines"* för att göra en övergång non-deterministisk (till en viss grad) vilket Millington och Funge ger exempel på i sin bok [3 kap. 5.5 "Fuzzy Logic"].

Dessa är dock endast en del av de olika metoderna som kan användas med/istället för tillståndsmaskiner; ett annat exempel skulle vara t.ex. att hårdkoda beteende åt NPC:er så att dessa fungerar på samma sätt varje gång. Dessa metoder för förståss med sig egna problem som kan användas för spelets fördel bara designen tillåter det. En tillståndsmaskin som använder t.ex. fuzzy logic kanske inte är ideal om dess NPC ska ha ett förutsägbart beteende. Spelindustrin är fylld med många sådana system (somliga vilka kan vara betydligt mera komplexa) och är därför inte nödvändiga att förstå utan och innan, vad som är viktigt är att veta att dessa system existerar och möjligheterna de medföljer är för dyrbara för att ignorera.

5. Slutsats

Efter att ha läst de olika metoderna Millington och Funge presenterar och de andra systemen som presenteras i källorna (vilka finns i stycke 6.) tycker jag att finita tillståndsmaskiner är en gammal men fortfarande användbar metod för att skapa beteenden i spel. Innan jag skrev denna avhandling undrade jag hur mycket teorin skiljer sig från den praktiska implementationen då jag själv skapat en finit tillståndsmaskin (se bilagorna) utan att veta någonting om teorin. Vad jag funnit är att medan modellen som presenteras i teorin har överförts till praktiken i spel har den formella teorin inte nödvändigtvis tagits bort men ändå inte helt tillämpats i implementationerna. Som tidigare nämnts är en definitiv implementation av dem i spel så gott som omöjlig att definiera då alla spel kräver olika saker av dem. En implementering med alfabet och sluttillstånd hittades aldrig och mitt antagande vad gäller detta är att spel ofta inte har möjligheten att fungera som de teoretiska maskinerna gör. Detta betyder inte att det är omöjligt att implementera dem i spel men strukturer där predikat kan användas som t.ex. {hp < 16} och {hungrig == True} verkar enligt mig vara lättare att arbeta med än att på något sätt definiera ett spel-alfabet och använda de som input. På detta sätt finner jag att skillnaden mellan modeller som t.ex. Beslutsträd och tillståndsmaskiner är extremt godtycklig då de kan definieras på sätt som gör dem praktiskt taget oskiljaktiga (även om beslutsträd är strukturerade som träd).

Mitt intresse i detta fall ligger i artificiell intelligens lika väl som spel och de andra metoderna skulle likaväl kunna utforskas vidare för att skriva andra uppsatser som exempel inlärning i spel och de olika sättet man åstadkommer detta och slumpgenerering i spel etc.

Spel är även en nästan perfekt plattform för att testa och träna AI. Jag hittade två rapporter som skrivits i detta sammanhang nämligen Dubey et al [7] och Togelius et al. 2007 [10] vars forskning gör mig intresserad av att själv utveckla och använda spel i detta syfte. AI har hjälp spel att utvecklas och vise versa och jag ser fram emot att få se vad som utvecklas näst.

6. Referenser

[1] Nareyek, Alexander. 2004. “AI in Computer Games.” *Queueing Systems. Theory and Applications* 1 (10): 58.

[2] Russell, Stuart, and Peter Norvig. 2013. *Artificial Intelligence: Pearson New International Edition: A Modern Approach*. Pearson Higher Ed.

[3] Millington, Ian, and John Funge. 2016. *Artificial Intelligence for Games*. CRC Press.

[4] Turing, Alan M. 2009. “Computing Machinery and Intelligence.” In *Parsing the Turing Test*, 23–65.

[5] Kozen, Dexter C. 1977. *Automata and Computability*.

[6] Fortnow, Lance. 1999. “Sipser Michael. Introduction to the Theory of Computation. PWS Publishing Company, Boston Etc. 1997, Xv 396 Pp.” *Journal of Symbolic Logic* 64 (01): 403.

[7] Dubey et al, “*Investigating human priors from playing video games*”, University of California, Berkeley

[8] Adams, Ernest. 2013. *Fundamentals of Game Design*. New Riders.

[9] Budziszewski, Konrad. 2011. “Racing the Beam: The Atari Video Computer System, by Nick Montfort and Ian Bogost.” *Popular Communication* 9 (1): 60–62.

[10] Togelius, Julian, Simon M. Lucas, and Renzo De Nardi. 2007. “Computational Intelligence in Racing Games.” In *Studies in Computational Intelligence*, 39–69.

Bilagor

Bilaga 1.

https://github.com/JoelSjoberg/Project-unlucky/tree/Prototype2

Bilaga 2.

```
public enum bossState
{
    idle,
    raging,
    attacking,
    stomping,
    rampaging,
    specialAttacking,
    stunned,
    draining,
    returning
}
```

Bilaga 3.

```
void Update () {
        if (baseBehaviour.player.health <= 0 || baseBehaviour.health <=0)
        {
            gameObject.SetActive(false);
            FindObjectOfType<AudioController>().playTheme("LevelEnd");
        }
        switch(state)
        {
            case bossState.idle:
                Idle();
                break;
            case bossState.raging:
                Raging();
                break;
            case bossState.attacking:
                Attacking();
                break;
            case bossState.stomping:
                Stomping();
                break;
            case bossState.rampaging:
                Rampaging();
                break;
            case bossState.specialAttacking:
                specialAttack();
                break;
            case bossState.stunned:
                Stunned();
                break;
            case bossState.draining:
                Draining();
                break;
            case bossState.returning:
                returning();
                break;
        }
}
```

Bilaga 4.

```
Vector3 playerPos;
private void Attacking()
{
    if (timer == 0)
    {
        duration = 24;
        FindObjectOfType<AudioController>().playTheme("Attacking");
        playerPos = baseBehaviour.player.transform.position;
        baseBehaviour.speed = 150;
    }
    // Update state, this is the attack itself: charge, blast away and repeat
    if (timer <= duration)
    {
        timer += Time.deltaTime;
        if(( (Vector3)(playerPos - transform.position)).magnitude > 5)
            moveTowardsPoint(playerPos);
        else
        {
            if (baseBehaviour.collidingWithPlayer)
            {
                 baseBehaviour.player.takeDamage(baseBehaviour.damage);
            }

            if (!wait(1f))
            {
                playerPos = baseBehaviour.player.transform.position;
            }
        }
    }
    else
    {
        timer = 0;
        state = bossState.stomping;
    }
}
```