Examensarbete

HÖGSKOLAN I SKÖVDE

STYRBETEENDEN FÖR ATT NAVIGERA GRUPPER AV AUTONOMA AGENTER

STEERING BEHAVIORS TO NAVIGATE GROUPS OF AUTONOMOUS AGENTS

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2015

Arvid Backman

Handledare: Mikael Johannesson
Examinator: Anders Dahlbom

Sammanfattning

Nyckelord:

Innehållsförteckning

1 Introduktion

Artificiell intelligens (AI) har alltid varit ett betydande område inom datalogi och det finns fortfarande mycket mer att lära inom området. Inom spel har förekommandet av AI ökat väldigt snabbt under de senaste åren och det läggs större krav på spel och dess AI i den aspekten att agenter ska bete sig trovärdigt. Söktekniker för vägplanering hos agenter är den AI som är vanligast inom spel. En vanlig vägplaneringsalgoritm som används på agenter i spel med statiska hinder är A* (A-stjärna). Utöver en vägplaneringsalgoritm krävs det någon sorts navigering hos agenterna i spelet för att de ska kunna röra sig efter den planerade vägen.

I takt att spel blir större blir även dess AI mer komplex. Hantering och navigering hos stora grupper av agenter i en spelmiljö är ett exempel på detta. Realtidstrategi-genren är en genre som behöver handskas med detta problem men det finns i andra genrer också. Exempelvis i ett FPS där civila människor springer och söker skydd.

Ett sätt att navigera agenter i en miljö är med hjälp av så kallade styrbeteenden. När ett styrbeteende appliceras på en agent kommer den agenten att kunna agera och ta egna besluta. Dessa beslut grundas på den informationen som ges från miljön runtomkring.

Arbetet kommer att analysera två styrbeteenden för att navigera grupper av agenter genom en miljö. Dessa två beteenden är: flödesfälts- och vägföljningsbeteende. Den aspekt som ska analyseras är varje tekniks minneseffektivitet och hur den förändras i olika miljöer och storleken på grupperna.

2 Bakgrund

2.1 Artificiell intelligens

Artificiell intelligens är konsten att skapa maskiner som utför uppgifter som kräver intelligens när de utförs av människor (Kurzweil, 1990). Det är möjligt att programmera en dator att utföra uppgifter som är omöjliga för en människa (eller en grupp människor) att lösa under en rimlig tid, exempelvis: avancerad sökning, utmanande aritmetiska problem, med mera.

Det är dock ett flertal uppgifter som datorer är dåliga på att utföra, som människor finner triviala: bestämma vad som ska göras härnäst, känna igen ansikten och vara kreativa är endast några få exempel. Det är just detta som AI-området utforskar genom att undersöka vilka algoritmer som krävs för att skapa dessa egenskaper hos datorer.

2.1.1 Traditionell AI

Det traditionella AI-området är uppdelat i två, mindre, områden: stark AI och svag AI. Forskningen inom stark AI eftersträvar att skapa ett beteende som efterliknar människors tankeprocess, medan forskningen inom svag AI applicerar AI-teknologier för att lösa problem som finns i den verkliga världen. Dessa två subområden tenderar att fokusera på att lösa ett problem på ett sätt utan att ta större hänsyn till hårdvara eller tidsbegränsningar. Till exempel kan en AI-forskare låta en simulation exekveras i timmar, dagar, eller veckor så länge det ger ett lyckat resultat som kan diskuteras i en artikel (Buckland, 2004).

2.1.2 Spel-AI

Till skillnad från traditionell AI måste AI-system inom spel ta hänsyn till vilka resurser som användaren har, till hur mycket processorkraft eller minne användaren har. Artificiell intelligens har alltid funnits inom datorspel, men *Pacman* (Namco, 1980) var det första spelet med en relativt avancerad AI. Fienderna rörde sig runt omkring banan i samma mån som spelaren. (Millington & Funge, 2009).

En väldigt stor andel av de spel som finns idag har någon sorts AI implementerad. Oavsett om det är en hund som rör sig mellan två olika rum i ett hus, eller om det är en mer avancerad Non-Player Character (NPC) i ett rollspel som rör sig runt i en by, har båda agenterna ett sätt att navigera sig genom den miljön de befinner sig i. Inom datorspel är navigering och rörelse av agenter ett vanligt problem, oftast inom spel där en grupp av agenter ska navigeras tätt intill varandra. Tätt navigerande agenter är väldigt vanligt inom genren realtidstrategi (RTS) så som *Starcraft 2: Wings of Liberty* (Blizzard Entertainment, 2010) och *Warcraft 3: Reign of Chaos* (Blizzard Entertainment, 2002).

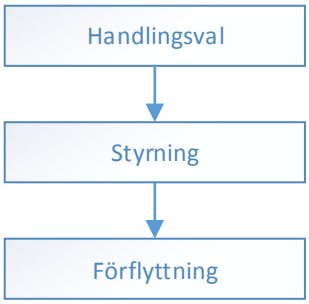
2.1.3 Autonom agent

En autonom agent är en enhet som kan uppfatta miljön runtomkring sig och agera utifrån den informationen, vilket ger agenten funktionaliteten att till viss del improvisera sina beslut. En agents beslut definierar vilket beteenden som en agent ska ha.

Ett beteenden är något som får agenten att utföra olika uppgifter i en miljö. Ett exempel på ett beteenden är att en agent närmar sig en farlig fiende och beslutar sig för att fly från den fienden. En agents beteenden beskrivs av agentfunktioner som mappar en given uppfattning till en mekanism (Russell & Norvig, 2009).

Reynolds beskriver i *Steering Behaviors For Autonomous Characters* (1999) att ett beteende hos en autonom agent kan delas upp i flera lager för att lättare förstå det. Dessa lager är: handlingsval, styrning, och förflyttning (Figur 1).

- Handlingsval: Lagret som bestämmer vilket mål agenten har. Till exempel: ”gå hit”.
- Styrning: Lagret som ansvarar över agentens navigering och ser till att målen från det tidigare lagret uppfylls. Detta uppfylls genom att applicera styrbeteenden hos agenten för att producera en styrkraft som beskriver hur agenten ska röra sig.
- Förflyttning: Det lager som ansvarar för en agents förflyttning. Detta lager konverterar kontrollsignaler från styrningslagret till rörelse av agentens kropp.



Figur 1 Ett beteendes tre lager

Ett exempel på en autonom agents beteende, i ett realtidstrategispel, är att spelaren ger order åt en autonom agent att röra sig till en ny position och den autonoma agenten navigerar sig själv genom miljön. Ett annat exempel är att ha två autonoma agenter, en råtta och en katt. Katten rör sig runt i en miljö medan musen sitter och äter. När musen ser att katten närmar sig flyr den från katten, samtidigt som katten börjar jaga musen. Alla dessa beslut görs utan någon översyn av en programmerare eller spelare.

2.2 Realtidstrategispel

I realtidstrategispel är det användarens jobb att positionera och manövrerar grupper av enheter och byggnader som de har kontroll över. Användaren ska ta över områden i en värld för att sedan förstöra motståndarens tillgångar (Figur 2 visar en skärmdump från ett realtidstrategispel). Via byggnaderna kan användaren bygga nya enheter som alla har olika förmågor. Det kostar olika resurser att skapa byggnader och enheter. Dessa resurser måste användaren samla in på olika sätt.

Figur 2 Skärmdump från realtidstrategi-spelet: *Starcraft 2: Wings of Libery* (Blizzard Entertainment, 2010).

Storleken på de grupper av enheter som användaren manövrerar varierar aktivt mellan små grupper till väldigt stora grupper. Dessa grupper måste navigera sig själv på ett snabbt och effektivt sett för att det ska vara enkelt för användaren att manövrera dem under spelet gång.

Det finns ett flertal sätt att navigera dessa grupper av enheter i miljön. Exempelvis kan man låta enheterna röra sig helt själv utan att ha någon typ av uppfattning om enheterna som befinner sig omkring sig. Det går även att hantera en grupp som en enda stor enhet med hjälp av olika kombinationer av styrbeteenden för att på ett naturligt sätt få enheterna att navigera sig genom miljön.

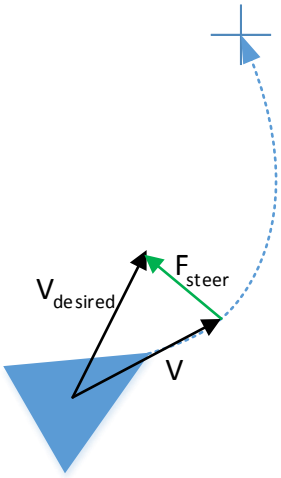
2.3 Styrbeteende

Ett beteende som appliceras för att producera en styrkraft hos en agent, kallas för ett styrbeteende. Det finns en mängd olika styrebeteenden som producerar en styrkraft på olika sätt. Flera av styrbeteendena kan kombineras för att styra den autonoma agenten på ett mer komplext och naturligt sätt. Dessa styrebeteenden presenteras av Reynolds i hans artikel *Steering Behaviors For Autonomous Characters* (1999).

2.3.1 Sök

Sök är ett styrbeteende som används för att styra en agent mot en specificerad position, i global rymd. Figur 3 visar en visualisering av sökbeteendet. Beräkningen för detta styrbeteende är

relativt enkel, först beräknas en önskad hastighet och med den önskade hastigheten kan man enkelt beräkna vilken styrkraft som ska appliceras på agenten.



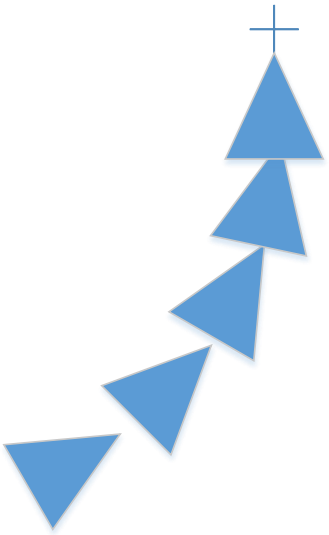
Figur 3 Sökbeteendet

```
vector2D function seek(vector2D target)
{
        var desiredVelocity = target - agentPosition;
        var force = desiredVelocity - agentVelocity;
        return force;
}
```

Pseudokod 1 Pseudokod för sökningsbeteendet.

2.3.2 Ankomst

Styrebeteendet ankomst är identiskt till sök, så länge agenten är långt ifrån den specificerade målpositionen. Det som skiljer ankomstbeteendet från sökbeteendet är att den får en agent att sakta ner när agenten närmar sig sitt mål (se Figur 4).



Figur 4 Ankomstbeteendet

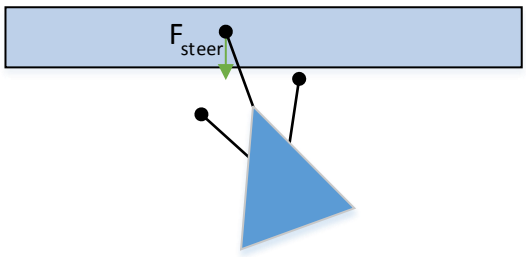
```
vector2D function arrive(vector2D target, float slowingDistance)
{
        var offset = target - agentPosition;
        var magnitude = offset.magnitude;

        float rampedSpeed = agentMaxVelocity * (magnitude / slowingDistance);
        float clippedSpeed = min(rampedSpeed, agentMaxVelocity);
        var desiredVelocity = (clippedSpeed / magnitude) * offset;

        var force = desiredVelocity - agentVelocity;
        return force;
}
```

Pseudokod 2 Pseudokod för ankomstbeteendet.

2.3.3 Väggundvikande

Väggundvikande ger en agent förmågan att styra ifrån potentiella kollisioner med väggar i en miljö. En vägg är ett linjesegment, i 3D en polygon. Detta görs med hjälp av ett antal avkännare hos agenten. Om en avkännare korsar en vägg beräknas en styrkraft genom att beräkna hur mycket avkännaren har penetrerat väggen och sedan skapa en kraft i väggens normalriktning beroende på hur långt avkännaren penetrerade väggen (se Figur 5).

Denna typ av teknik för att undvika väggar går även att använda för att undvika vanliga objekt i en miljö i och med att tekniken endast tittar om en avkännare har korsat en linje. Det ger möjligheten att skapa objekt (exempelvis cirkel- och rektangel-formade objekt) av en mängd linjer i en miljö som en agent kan undvika kollision med.



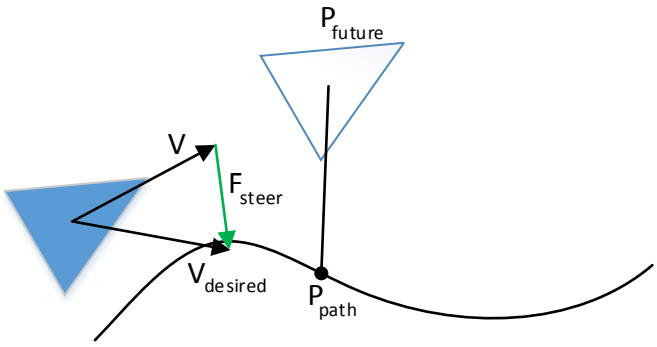
Figur 5 Väggundvikelsebeteendet

```
vector2D function wallAvoidance(array<wall> walls)
{
        var force;
        var closestWall = -1;
        var intersectionPoint;
        var closestPoint;
        var distanceToThisIntersection = 0;
        var distanceToClosestIntersection = MaxNumber;

        foreach(probe in agentProbes)
        {
                foreach(wall in walls)
                {
                        if(lineIntersection(agentPosition, probe, wall.from, wall.to,
                                        distanceToThisIntersection, intersectionPoint)
                        {
                                if(distanceToThisIntersection < distanceToClosestIntersection)
                                {
                                        distanceToClosestIntersection =
                                                                distanceToThisIntersection;
                                        closestWall = wall;
                                        closestPoint = intersectionPoint;
                                }
                        }
                }

                if(closestWall > -1)
                {
                        var overShoot = feeler.position - closestPoint;
                        force = closestWall.normal * overShoot.magnitude;
                }
        }

        return force;
}
```

Pseudokod 3 Pseudokod för väggundvikelsebeteendet.

2.3.4 Vägföljning

Vägföljningsbeteendet gör det möjligt för en agent att styras längs en serie av positioner som formar en väg. Agenten tittar, med hjälp av sin nuvarande position och hastighet, vilken position agenten kommer att ha i framtiden. Från den positionen beräknas den närmsta punkten på vägen. Ett sökbeteende appliceras sedan på agenten, med den närmsta punkten på vägen som målposition. Om målpositionen skulle vara den sista punkten på vägen byts sökbeteendet ut mot ankomstbeteendet (se Figur 6).



Figur 6 Vägföjlningsbeteendet

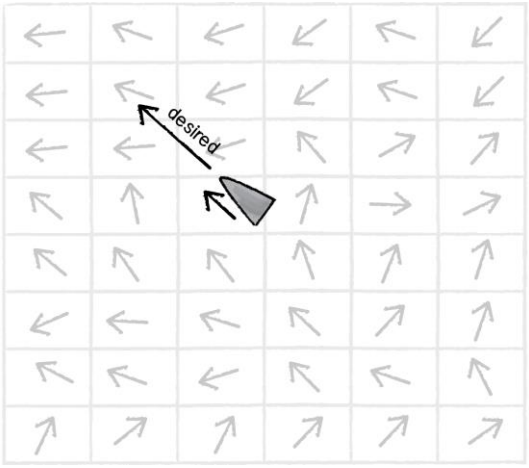
```
vector2D function followPath(Path path)
{
        var predict = agentVelocity;
        var force;
        predict.normalize;
        predict *= 25;
        var predictedPosition = agentPosition + predict;

        var closestDistance = MaxNumber;
        var closestPoint;

        foreach(line in path.line)
        {
                var closestPointOnLine = line.closestPoint(predictedPosition);

                var offset = predictedPosition - closetPointOnLine;
                var magnitude = offest.magnitude;

                if(magnitude < closestDistance)
                {
                        closestDistance = magnitude;
                        closestPoint = closestPointOnLine;
                        var lineDirection = line.to - line.from;
                        lineDirection.normalize();
                        lineDirection *= 10;
                        closestPoint += lineDirection;
                }
        }

        force = seek(closestPoint);

        return force;
}
```

Pseudokod 4 Pseudokod för vägföljningsbeteendet.

2.3.5 Flödesfält

Styrbeteendet flödesfält är ett beteende för navigering av agenter, och kan användas som ett alternativ till vägföljningsbeteendet. Ett flödesfält är ett rutnät, där varje cell i rutnätet innehåller en riktningsvektor. Denna riktningsvektor representerar vilken styrkraft som ska appliceras på en agent när den befinner sig i cellen (se Figur 7). Cellernas riktningsvektorer kan vara statiska, men de kan också uppdateras dynamiskt. Dynamisk uppdatering av flödesfältet är fördelaktigt i spel där hinder för agenter förändras i realtid. Exempelvis realtidstrategispel, där agenterna inte får styras in i varandra och deras position alltid förändras.

Under senare år har det skrivits ett antal artiklar där man tar upp denna typ av styrbeteende för att simulera stora grupper av agenter. Några exempel är *Continuum Crowds* (Treuille et al., 2006), använder flödesfält för att simulera en stadsmiljö med människor och bilar, *Directing Crowd Simulations Using Navigation Fields* (Patil et al., 2011), använder flödesfält för att simulera folkmassor i trånga utrymmen. Graham Pentheny (2013) diskuterar även om möjligheterna att använda flödesfält i spel där stora grupper av agenter måste förflytta sig i en miljö. Ett exempel på ett realtidstrategispel som använder flödesfält för att hantera grupper av agenter är *Planetary Annihilation* (Uber Entertainment, 2014).



Figur 7 Flödesfältsbeteende (Shiffman, 2012)

```
vector2D function flowFieldFollow(FlowField flowField)
{
        var desiredVelocity = flowField.getDirectionFromCell(agentPosition);
        desiredVelocity *= agentMaxVelocity;

        var force = desiredVelocity - agentVelocity;
        return force;
}
```

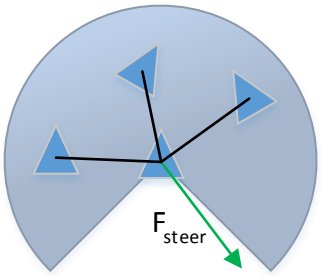
Pseudokod 5 Pseudokod för flödesfältsbeteendet.

2.3.6 Flockbeteende

I *Flocks, Herds, and Schools: A Distributed Behavioral Model* (1987) beskriver Reynolds tre olika styrbeteenden som alla samarbetar för att skapa ett flockbeteende hos grupper av agenter. Dessa tre styrbeteenden är: Separation, sammanhållning, och formering. Styrbeteendena appliceras endast på en agent beroende på de agenter som befinner sig inom sitt närområde. Närområdet definieras med hur långt och brett en agents synfält är (se Figur 8).

Figur 8 En agents närområde

Separationsbeteendet skapar en kraft som styr en agent ifrån andra agenter inom sitt närområde. När detta beteende appliceras på en mängd agenter kommer dom sprida ut sig, och försöka maximera längden från varandra. Styrkraften beräknas genom att beräkna en riktningsvektor mot alla agenter i närområdet. Riktningsvektorerna normaliseras och adderas sedan (se Figur 9). Detta beteende kan användas för att hindra en grupp av agenter att tränga ihop sig.

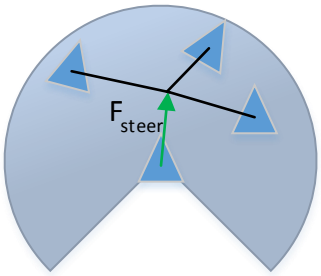


Figur 9 Separationsbeteendet

```
vector2D function separation(array<agents> neighbors)
{
        var force;

        foreach(neighbor in neighbors)
        {
                var offset = agentPosition - neighbor.position;
                force += offset.normalize/offest.magnitude;
        }

        return force;
}
```

Pseudokod 6 Pseudokod för separationsbeteendet.

Sammanhållningsbeteendet ger en agent förmågan att närma och gruppera sig med andra agenter i närområdet. Styrkraften beräknas genom att beräkna medelpositionen hos de närliggande agenterna. Styrkraften kan sedan appliceras i riktningen från agenten och medelpositionen (se Figur 10).

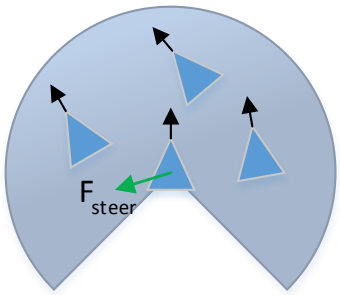


Figur 10 Sammanhållningsbeteendet

```
vector2D function cohesion(array<agents> neighbors)
{
	var force;
	var centerOfMass;
	var neighborCount = neighbor.size;

	foreach(neighbor in neighbors)
	{
		centerOfMass += neighbor.position;
	}
	if(neighborCount > 0)
	{
		centerOfMass = centerOfMass / neighborCount;
		force = seek(centerOfMass);
	}
	return force;
}
```

Pseudokod 7 Pseudokod för sammanhållningsbeteendet.

Formeringsbeteendet ger en agent förmågan att röra sig i samma riktning och hastighet som agenter i närområdet. Styrkraften beräknas genom att först beräkna medelhastigheten hos alla närliggande agenter och sedan subtrahera denna medelhastighet med agentens egen hastighet (se Figur 11).



Figur 11 Formeringsbeteendet

```
vector2D function alignment(array<agents> neighbors)
{
	var force;
	var averageHeading;
	var neighborCount = neighbor.size;

	foreach(neighbor in neighbors)
	{
		averageHeading += neighbor.heading;
	}

	if(neighborCount > 0)
	{
		averageHeading = averageHeading / neighborCount;
		force = averageHeading - agentHeading;
	}

	return force;
}
```

Pseudokod 8 Pseudokod för formeringsbeteendet.

2.4 Beräkningsmodell för kombination av styrbeteenden

Det är sällan som en autonom agent endast styrs av ett enda styrbeteende, utan det är oftast en samling styrbeteenden som styr en autonom agents beteenden. Det krävs till exempel tre styrbeteenden för att åstadkomma ett flockbeteende hos agenter; separation, sammanhållning, och formering. Vill man tillexempel att agenter ska söka sig till ett mål och samtidigt undvika kollision med andra agenter kan en lösning vara att kombinera sök- och separationsbeteendena.

Grunden bakom alla beräkningsmodeller är att addera ihop alla styrkrafter som produceras av styrbeteendena. Denna summa är den totala styrkraften som appliceras på agenten. Denna styrkraft får dock inte vara större än en specificerad maxkraft hos agenten och därför trunkeras alltid agentens kraft med dess maxkraft efter att alla styrkrafter har beräknats. Två beräkningsmodeller beskrivs kort av Reynold i *Steering Behaviors For Autonomous Characters* (1999) medan Buckland (2004) går in på en mer detaljerad nivå när han beskriver dem.

2.4.1 Viktad trunkerad summa

Det enklaste sättet att kombinera styrbeteenden är genom att multiplicera varje styrbeteende med en vikt för att sedan trunkera resultatet med maxkraften hos agenten. Med denna beräkningsmodell medföljer dock några nackdelar (Reynolds, 1999; Buckland, 2004). Ett problem är att varje aktivt styrbeteenden beräknas varje tidssteg, så är den väldigt ineffektiv att utföra. En annan nackdel är att det inte är trivialt att välja hur stor vikterna för alla styrbeteenden ska vara trots det faktum att vikterna justeras väldigt noggrant kan det hända att en del beteenden stängs ute på grund av att agentens totala styrkraft trunkeras.

```
vector2D function weightedTruncatedSum()
{
        var force;
        foreach(behavior in behaviors)
        {
                force += behavior.calculateForce() * behavior.weight;
        }

        force.truncate(agentMaxForce);
        return force;
}
```

Pseudokod 9 Pseudokod för att beräkna en agents totala styrkraft med viktad trunkerad summa.

2.4.2 Viktad trunkerad summa med prioritering

Craig Reynolds (1999) tog fram denna modell för att adressera problemet att beteenden stänger ute varandra. Beräkningsmodell är väldigt lik den förra modellen, men den har två egenskaper som skiljer dem åt. Den första egenskapen är att varje styrbeteende har en prioritet som bestämmer hur viktig den är. Ett styrbeteende med en hög prioritering kommer att uppdateras före de som har låg prioritering. Exempelvis kan det vara viktigare att uppdatera att uppdatera separationsbeteendet, istället för sökbeteendet, hos en grupp agenter som inte ska kollidera med varandra.

Den andra egenskapen som skiljer denna modell ifrån den tidigare är att om den totala styrkraften hos en agent överskrider dess maximala tillåtna styrkraft kommer de resterande styrbeteendena, som har en lägre prioritering, inte att uppdateras. Det innebär att modellen ibland inte uppdaterar vissa styrbeteenden.

```
vector2D function weightedTruncatedSumWithPriority()
{
        var force;
        foreach(behavior in behaviorsPriorityQueue)
        {
                var forceSoFar = force.magnitude;
                var forceRemaining = agentMaxForce - forceSoFar;

                if(forceRemaining <= 0)
                {
                        break;
                }

                var forceToAdd = behavior.calculateForce() * behavior.weight;

                if(forceToAdd.magnitude < forceRemaining)
                {
                        force += forceToAdd;
                }
                else
                {
                        force += forceToAdd.normalize * forceRemaining;
                }
        }

        return force;
}
```

Pseudokod 10 Pseudokod för att beräkna en agents totala styrkraft med viktad trunkerad summa med prioritering.

2.5 Vägplanering

Den vanligaste metoden för att få agenter att navigera sig igenom en miljö, i ett datorspel, är genom vägplanering. Den huvudsakliga uppgiften som vägplaneringen står för, är att hitta den kortaste vägen mellan två definierade punkter. Vägplaneringen tar även hänsyn till statiska objekt i miljön, såsom byggnader, berg osv., när den kortaste vägen mellan punkterna beräknas. A* (Hart et al., 1968) är en av de vanligaste algoritmerna för att beräkna den kortaste vägen.

2.5.1 A*

A*, även kallad A-stjärna, är en sökalgoritm som först framställdes av Peter Hart, Nils Nilsson, och Betram Raphael (1968). Algoritmen är en förbättring av Djikstras (1959) algoritm och det som främst skiljer de två algoritmerna ifrån varandra, är att A* använder sig av heuristik för att hitta den kortaste vägen i en graf. Heuristik innebär att algoritmen gör en uppskattning av vad avståndet från den nuvarande positionen till målpositionen. Funktionen för att beräkna det heuristiska värdet varierar från problem till problem och alla medför för- och nackdelar.

Anledningen till att A* är en vanlig algoritm för vägplanering inom dataspel är på grund av två egenskaper den har. Den första egenskapen är att algoritmen är komplett, vilket innebär att om det finns en väg till målet kommer den vägen att hittas. Den andra egenskapen är att algoritmen alltid kommer att hitta den mest optimala vägen, om det heuristiska värdet inte överskrider den verkliga kostnaden (Hart et al., 1968).

3 Problemformulering

Denna del av examensarbetet är uppdelat i två rubriker: Problembeskrivning och Metodbeskrivning. Problembeskrivning redovisar det problem som examensarbetet är baserat på. Metodbeskrivning redovisar hur examensarbetets frågeställning ska besvaras, undersökas, och utvärderas.

3.1 Problembeskrivning

I takten med att datorspel blir större ökar även komplexiteten hos AI-systemen som de använder (se exempelvis Brian Schwab (2004)). Antalet agenter ökar och komplexiteten på miljöerna dessa agenter rör sig i blir högre. Detta kommer att ha en påverkan på minneseffektiviteten hos det AI-system som navigerar agenterna i spelvärlden. Det är viktigt att AI-systemets minneseffektivitet inte blir för låg, i och med att det endast finns en begränsad resurs att använda sig utav och låg minneseffektivitet leder till att spelet inte kommer köras på ett bra sätt och spelupplevelsen för användaren kommer att försämras.

Som beskrivet i avsnitt 2.3 är styrbeteenden är en samling av olika tekniker vars syfte är att styra en autonom agent och de första av dessa styrbeteenden skapades av Craig Reynolds (1987). Avsnitt 2.3.4 och avsnitt 2.3.5 förklarar vägföljningsbeteende respektive flödesfältsbeteende. Dessa två beteenden har båda som syfte att röra sig från en punkt till en annan punkt i en miljö.

Examensarbetet kommer att fokusera på vägföljnings- och flödesfältsbeteende för att navigera grupper av agenter genom en miljö. Detta är främst applicerbart hos spel som går inom genren realtidstrategispel men kan även användas inom andra spelgenrer, som till exempel rollspel där icke spelarstyrda karaktärer ska navigera sig i en miljö. Det går även att använda dessa tekniker för olika simulationer. Till exempel används flödesfältsbeteende i artikeln *Directing Crowd Simulation Using Navigation Fields* (Patil et al., 2011) för att simulera stora folkmassor i stadsmiljöer.

Den frågeställning som arbetet kommer försöka besvara är:

- Hur jämför sig styrbeteendena flödesfälts- och vägföljnings-beteende för att styra grupper av autonoma agenter i olika miljöer med avseende på minneseffektivitet?

För att det ska vara möjligt att besvara frågeställningen kommer en applikation skapas. Det ska vara möjligt att förändra vissa värden i applikation, som till exempel rutnätets densitet för navigering med flödesfält. Det ska även vara möjligt att välja en miljö som testerna ska utföras på.

Arbetets genomförande är indelat i två delsteg. Det första steget är att implementera en applikation vars syfte är att testa de två teknikerna som arbetet ska utvärdera i ett antal miljöer. Det andra steget är att göra utvärderingar på teknikerna med hjälp av den applikation som tidigare implementerats.

3.1.1 Delmål 1: Implementation

Avsikten med detta delmål är att implementera den applikation som kommer att användas för att utvärdera navigationsteknikerna. Därmed behöver alla styrbeteenden och beräkningsmodeller som ska användas, för utvärderingen, implementeras i applikationen. Det

ska vara möjligt att skapa olika testfall för att enkelt kunna analysera teknikerna. Med olika testfall menas vilken miljö som ska användas, hur många agenter som ska navigera genom vald miljö, och vilken teknik de ska använda.

3.1.2 Delmål 2: Utvärdering

Detta delmål har som har som syfte att utvärdera minnesanvändningen hos de två styrbeteendena. Båda teknikerna kommer att slås ihop med andra styrbeteenden när de utvärderas. Detta kommer att ske med hjälp av en beräkningsmodell.

Det är i detta steg testfallen skapas med hjälp av applikation som implementerats i det tidigare delmålet. Testfallen kommer bestå av grupper av autonoma agenter och storleken på dessa grupper kommer att variera från ett litet antal till ett sjuttiotal agenter mellan de olika testfallen. Eftersom att miljöer i spel kan variera kommer testfallen att köras på ett flertal olika miljöer som lägger fokus på vissa aspekter, som till exempel trånga och fria utrymmen.

Den egenskap som arbetet kommer att utvärdera hos de två teknikerna är deras minneseffektivitet. Den operationella definitionen av minneseffektivitet i detta arbete är minnesanvändningen hos teknikerna. Med minnesanvändning menas hur mycket minne som allokeras och används för att navigera grupperna av agenter.

3.2 Metodbeskrivning

3.2.1 Metod för delmål 1: Implementation

Metoden för detta delmål är att implementera en applikation vars uppgift är att kunna besvara den frågeställning som arbetet har. Den mest vitala delen i detta delmål är implementationen av flödesfält- och vägföljningsnavigering. Styrbeteendena, och den beräkningsmodell, som agenterna kräver för att navigera sig i miljön måste också implementeras. Alla styrbeteenden som används i arbetet grundar sig från Craig Reynolds artikel *Steering Behaviors For Autonomous* Characters (1999). Det krävs även att applikationen har en implementation av sökalgoritmen A*, för att kunna hitta en väg som vägföljningsbeteendet kan använda. För att kunna evaluera minneseffektiviteten hos teknikerna används verktyget *dotMemory* (JetBrains, 2014).

3.2.2 Metod för delmål 2: Utvärdering

Metoden för denna del är att skapa testfall med applikationen som implementerats i det förra steget och sedan utvärdera testfallen. Testfallen går ut på att ett godtyckligt antal agenter i en grupp ska ta sig från en punk till en annan punkt i en miljö. För att göra det möjligt att utvärdera testfallen med avseende på minneseffektivitet kommer det att används ett mått:

- Hur mycket minne krävs för att genomföra testfallet?

För att utvärdera teknikerna baserat på det mått som definierats kommer ett flertal testfall utföras på de två teknikerna. Med hjälp av verktyget *dotMemory* (JetBrains, 2014) kommer resultaten från testfallen att kunna utvärderas. För att kunna mäta minnesanvändningen hos teknikerna kommer varje teknik att testas ett flertal gånger och alla testresultaten kommer sedan utvärderas baserat på deras respektive minnesanvändning där tre aspekter kommer ligga i fokus. Dessa tre aspekter är (i prioritetsordning): minnesanvändningens medelvärdet, minnesanvändningens största värde, och minnesanvändningens minsta värde. Resultaten kommer att jämföras för varje miljö och gruppstorlek.

Anledningen till att aspekterna prioriteras i den ordningen som de görs är för att det är viktigare att ha en jämn minnesanvändning och ett lågt medelvärde istället för att ha flera sekvenser av låg och hög minnesanvändning.

Den motivation som ligger bakom användandet av detta mått för att mäta minneseffektivitet hos de två teknikerna kommer från att AI-system inom spel har begränsade resurser att arbeta med och minneseffektiviteten hos systemen är därför viktigt (Buckland, 2004). Det kommer även viss inspiration från artikeln *CHARM: An efficient algorithm for closed itemset mining* (Mohammed J. Zaki, 2002) och från eget omdöme där det anses vara vettiga mått att använda för att se vilken av teknikerna som lämpar sig bäst (baserat på de valda måtten) för att navigera grupper av agenter under olika förhållanden.

3.3 Metodreflektion

Det går att diskutera den metod som har valts för det första delmålet. Det finns motorer för realtidstrategispel som är öppna och gratis som mycket väl hade kunnat användas för detta arbete. Anledningen till att ett eget ramverk och en egen applikation implementeras är för att få en större kontroll över arbetet och för att applikationen den ska bli mer specificerad mot det mått som används. Därför är det tvunget att en applikation implementeras för att det ska vara möjligt att få fram ett resultat som besvarar arbetets frågeställning.

Det hade varit möjligt att använda stora ordo som mått för att beräkna minneskomplexiteten hos teknikerna. Anledningen till att andra aspekter har valts för att mäta teknikernas minneseffektivitet är för att stora ordo endast ger det värsta fallet av minnesanvändning hos teknikerna. Att jämföra resultaten på flera aspekter ger ett mer konkret resultat över hur effektiviteten hos teknikerna ser ut.

För att få en större helhet över teknikernas minnesanvändning hade det gått att utvärdera flera aspekter. Ett exempel är att testa hur många gånger en teknik kommer över en definierad mängd minnesanvändning. Anledningen till valet att mäta minneseffektiviteten hos teknikerna genom att endast använda medelvärdet, största värdet, och minsta värdet är för att det inte är trivialt att definiera ett bra och konkret värde som teknikerna minnesanvändning inte för överskrida.

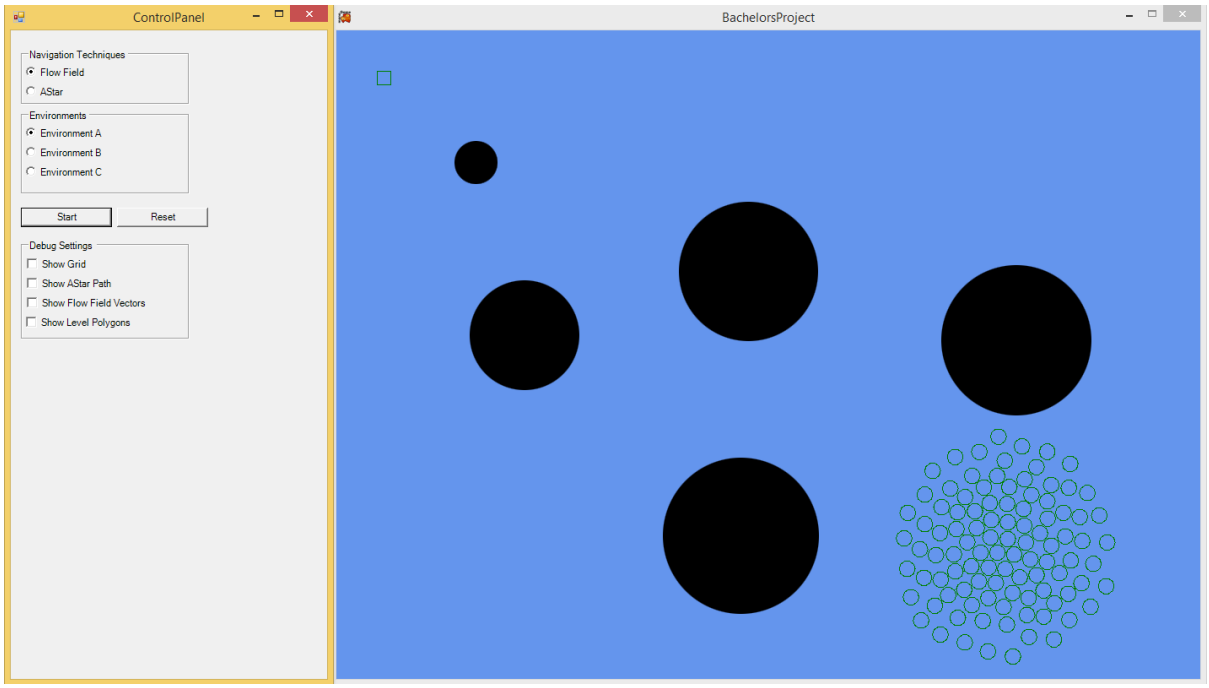
Ett problem som kan uppstå med mätningen av minnesanvändning är att applikationen som utför simuleringen också hanterar andra funktionaliteter (exempelvis rendering) vilket leder till att dessa funktionaliteter också tar upp minne. Detta problem kommer att reduceras genom att teknikerna även kommer att testas utan någon typ av rendering eller några andra funktionaliteter som inte har direkt påverkan på simuleringen och dess testresultat.

4 Implementation

I detta kapitel beskrivs applikationens implementation och design. En beskrivning av de designval och modifikationer som gjorts under applikationens implementation kommer också att tas upp. Förutom att beskriva designen och implementationen hos applikationen kommer de tekniker som används att beskrivas.

4.1 Applikation

Applikationen är den experimentmiljö där testfallen simuleras som består av två separerade fönster. Figur 12 visar ett exempel på hur applikationen ser ut efter uppstart. Det högra fönstret är vad som visualiserar och kör simuleringen av testfallen medan det vänstra fönstret representerar en inställningspanel. I inställningspanelen väljer man vilken teknik som ska användas, antal agenter som ska simuleras, och vilken miljö som ska användas. Ett antal inställningar som inte har någon direkt koppling till testfallet finns också tillgängligt till exempel är det möjligt att rita ut ett flödesfälts alla riktningsvektorer. Applikationen kan simulera testfallet genom att användaren klickar på startknappen.



Figur 12 Appliktationen.

4.1.1 Design

Applikationen är implementerad i programmeringsspråket C# och använder sig av biblioteket MonoGame som är en Open-Source-implementation av Microsoft XNA 4. Microsoft XNA är ett API för att göra det lättare att utveckla spel till Microsofts produkter (exempelvis Windows och XBOX 360). I och med att C# är ett objektorienterat programmeringsspråk följer applikationen också detta programmeringsparadigm.

Nedan är en lista med applikationens klasser och en beskrivning över vad deras syfte är.

- Program

    - En statisk klass vars enda syfte är att skapa en instans av MainApplication och kalla på funktionen *Run()*.
- MainApplication
    - Kärnklassen hos applikationen. Det är denna klass som skapar de fönster som applikationen består av. Det är även denna klass som styr över vilken teknik styr över simuleringen och applicerar alla inställningar som användaren gör.
- ControlPanel
    - Klassen som representerar inställningsfönstret. Har skapats med hjälp av Windows Forms för att enkelt skapa fönstret och dess funktionaliteter.
- Level
    - Basklass för alla miljöer som finns i applikationen. Varje level består av två rutnät. Ett rutnät som används för att generar en väg med A* och ett annat rutnät som representerar miljöns flödesfält. En miljö genereras från en bild där varje svart pixel i bilden representerar en ruta i ett rutnät som en agent inte kan röra sig på.
- BaseGrid
    - En abstrakt klass som representerar ett rutnät. Ett rutnät består av en tvådimensionell array som är fylld med BaseNode-objekt. Både rutnätet som används för att generera en väg med A* och flödesfältsrutnätet ärver av denna basklass.
- BaseNode
    - En abstrakt klass som representerar en nod i ett rutnät. Har som syfte att hålla data som är nödvändig för att utföra funktioner på de rutnät som de ligger i. Två klasser ärver från BaseNode, FlowFieldNode (en nod i flödesfältet) och AStarNode (en nod i rutnätet för A*).
- Agent
    - Efterliknar fordonsmodellen som beskrivs i artikeln *Steering Behaviors For Autonomous Characters* (Reynolds, 1999). Har information och agentens position, hastighet. För att beräkna och applicera de styrkrafter som är nödvändiga har en agent ett SteeringManager-objekt som hanterar detta.
- SteeringManager
    - Klass som hanterar beräkningen av alla styrkrafter som appliceras hos en agent. Implementationen av de styrbeteenden som används beskrivs i kapitel 2.3.

4.2 Utvecklingen av flödesfältsbeteendet

Den första implementationen av flödesbeteendet baserades på den teknik Reynolds (1999) beskriver. Där hämtar agenten den styrkraft som ska appliceras baserat på dess framtida position. Istället för att hämta styrkraften från agentens framtida position hämtas styrkraften från agentens nuvarande position. Anledningen till att denna implementationen av tekniken används är för att det inte gjorde någon större märkbar skillnad hos beteendet och koden blev simplare och mer effektiv om agenten hämtade styrkraften hos flödesfältet baserat på sin nuvarande position.

Genereringen av miljöernas flödesfält är uppdelade i två steg.

1. Skapa ett integreringsfält
2. Skapa riktningsvektorerna för flödesfältet

4.2.1 Integreringsfältet

Det är här som det mesta arbete hos flödesfältsgenereringen sker. Detta görs med hjälp av en modifikation av Djikstras (1959) algoritm. Detta integreringsfält kommer sedan användas för att generera riktningsvektorerna för flödesfältet. Nedan är en lista över de steg som algoritmen går igenom för att beräkna integreringsfältet.

1. Sätt alla noders kostnader till det högsta värdet en integer kan ha.
2. Målnoden får sin kostnad satt till noll och läggs in i en öppen lista.
3. En nuvarande nod blir tilldelad den nod som ligger längst fram i listan.
4. Alla grannar till den nuvarande noden får deras kostnad satt till den nuvarande nodens kostnad plus ett och läggs sedan till längst bak i den öppna listan.
5. Dessa steg fortsätter tills den öppna listan är tom.

4.2.2 Flödesfältsgenerering

Här används de resultat som kommer från integreringsfälts-beräkningarna för att beräkna riktningsvektorerna i flödesfältet. Detta åstadkoms genom att gå igenom alla noder i integreringsfältet för att jämföra alla noders åtta grannar för att hitta den grannen som har minst kostnad. När den grannen med minst kostnad har hittats riktar den nuvarande nodens riktningsvektor mot den grann-noden.

4.3 Utvecklingen av vägföljningsbeteende

En liten modifikation gjordes på det vägföljningsbeteende som används i applikation i jämförelse med den pseudo kod som beskrivs i kapitel 2.3.4. Istället för att agenten konstant applicerar sökbeteendet för att söka sig till den närmsta punkten på linjen i förhållande till sin framtida position appliceras endast sökbeteendet om agenten är på väg att röra sig utanför en definierad längd ifrån vägen. Denna teknik tas upp av Craig Reynolds (1999) och den beskrivs även i boken *The Nature of Code* (Shiffman, 2012).

Den väg som agenterna följer genereras med hjälp av sökalgoritmen A* (Hart et al., 1968). Nedan visas pseduokod över sökalgoritmen är implementerad i applikationen.

```
function AStar(AStarGrid grid) {
    priorityQueue   openList;
    List            closedList;

    var startNode = grid.startNode;
    startNode.g = 0;
    startNode.h = EstimatedDistToGoal(startNode);
    startNode.f = startNode.g + startNode.h;
    startnode.Parent = null;
    push startNode on openList;
    while(openList is not empty) {
        pop node n from openList;
        if(n is the goalNode) {
            construct path
            return path
        }
        foreach(Neighbor n' of n) {
            var newG = n.g + Cost(n', n)
            if(n' is in openList or cloedList and n'.g <= newG) {
                continue;
            }
            n'.Parent = n;
            n'.g = newG;
            n'.h = EstimatedDistToGoal(n');
            n'.f = n'.g + n'.h;
            if(n' is in closedList) {
                remove n from closedList;
            }
            if(n' is not in openList) {
                push n' on openList;
            }
        }
        push n onto closedList;
    }

    return null
}
```

Pseudokod 11 Pseudokod över A*-algoritmen

4.4 Testfallsloopen

Testfallsloopen är alla programsatser som applikationen kommer att exekvera under varje tidssteg för att agenterna ska bete sig korrekt. Pseudokoden nedan är en beskrivning över hur loopen ser ut.

```
foreach(Agent agent in currentLevel) {
    if(agent has not reached its goal) {
        Calculate necessary steering behaviors
        if (Use flow field behavior) {
            agent.FlowFieldUpdate();
        }
        else if (Use path following behavior) {
            agent.PathFollowingUpdate()
        }
        agent.CombineSteeringForces();
        agent.UpdatePosition();
    }
}
```

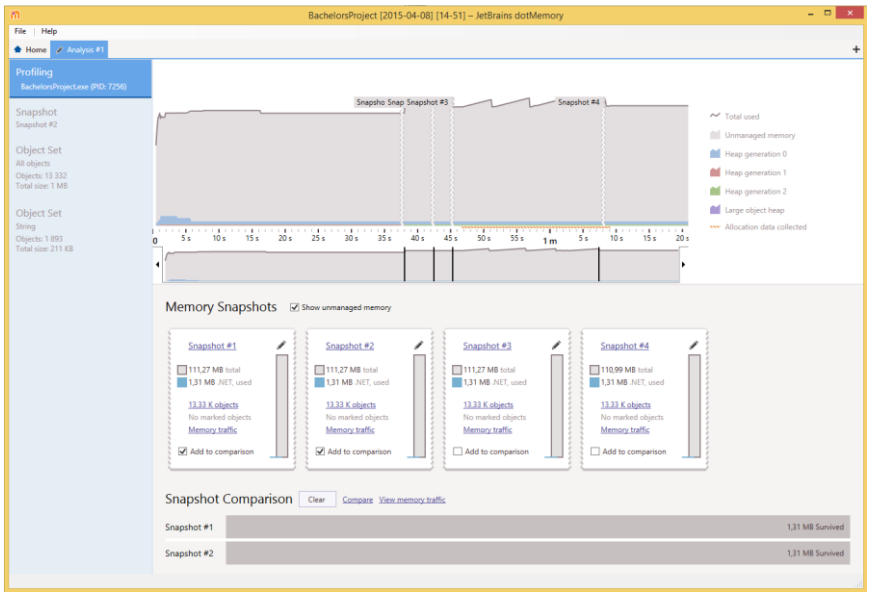
Pseudokod 12 Beskrivning över testfallsloopen

Applikationen göra några tester för att se vilken teknik som ska appliceras på agenterna för att bestämma vilka styrkrafter som ska appliceras på agenterna. Om till exempel testfallet där

agenterna använder flödesfältsbeteende kommer agentens styrbeteendehanterare att beräkna styrkraft baserat på den nuvarande banans flödesfält. En agent beräknar styrkrafter och uppdaterar sin position så länge den inte har nått sin målposition.

4.5 Mätning av minneseffektivitet

För att mäta minneseffektiviteten hos testfallen kommer tre värden att användas. Medelminnesanvändningen, högsta minnesanvändningen, och lägsta minnesanvändningen under testfallen. För att samla in all data som är nödvändig för att göra mätningarna kommer ett tredjepartsprogram att användas. Programmet används för att studera minnesanvändning hos program som är skrivna i C#, programmet heter *dotMemory* (JetBrains, 2014). Figur 13 visar ett exempel på hur utdatan i *dotMemory* (JetBrains, 2014) kan se ut. En mätning av ett testfall pågår så länge testfallsloopen är aktiv hos applikationen.



Figur 13 En skärmdump från *dotMemory* (JetBrains, 2014) som visar utdata från applikationens exekvering.

5 Utvärdering

5.1 Presentation av undersökning

5.2 Analys

5.3 Slutsatser

6 Avslutande diskussion

6.1 Sammanfattning

6.2 Diskussion

6.3 Framtida arbete

Referenser

Blizzard Entertainment (2010). *Starcraft 2: Wings of Liberty* (Version 1.0) [Datorprogram] Blizzard Entertainment.

Blizzard Entertainment (2002). *Warcraft III: Reign of Chaos* (Version 1.0) [Datorprogram] Blizzard Entertainment.

Buckland, M. (2004). *Ai Game Programming by Example*. Wordware Publishing Inc.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*. (http://csl.mendeley.com/styles/252469921/harvard-skovde-university-2). s. 269–271.

Hart, P., Nilsson, N.J. & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*. s. 100–107.

JetBrains (2014). *dotMemory* (Version 4.2) [Datorprogram] JetBrains.

Kurzweil, R. (1990). *The age of intelligent machines*. MIT Press.

Millington, I. & Funge, J. (2009). *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc.

Mohammed J. Zaki, C.H. (2002). *CHARM: An efficient algorithm for closed itemset mining*.

Namco (1980). *Pacman* (Version 1.0) [Datorprogram] Namco.

Patil, S., Berg, J. van den, Curtis, S., Lin, M.C. & Manocha, D. (2011). Directing crowd simulations using navigation fields. *IEEE transactions on visualization and computer graphics*. 17 (2). s. 244–54.

Pentheney, G. (2013). *GDC - The Next Vector: Improvements in AI Steering Behaviors*. 2013. Tillgänglig på Internet: http://www.gdcvault.com/play/1018230/The-Next-Vector-Improvements-in. [Hämtad 30 January 2015].

Reynolds, C. (1999). *Steering Behaviors For Autonomous Characters*.

Reynolds, C.W. (1987). Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*. 1 August 1987, New York, New York, USA: ACM Press, s. 25–34.

Russell, S. & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press.

Schwab, B. (2004). *Ai Game Engine Programming*. Charles River Media, Inc.

Shiffman, D. (2012). *The Nature of Code: Simulating Natural Systems with Processing*.

Treuille, A., Cooper, S. & Popović, Z. (2006). Continuum crowds. *ACM Transactions on Graphics*. s. 1160.

Uber Entertainment (2014). *Planetary Annihilation* (Version 1.0) [Datorprogram] Uber Entertainment.