NTNU
Det skapende universitet

Visualisering av store mengder data i brannvarslingssystem

Gaute Nilsson

Master i teknisk kybernetikk
Oppgaven levert: Juni 2011
Hovedveileder: Sverre Hendseth, ITK

Norges teknisk-naturvitenskapelige universitet

Visualisering av store mengder data i brannvarslingssystem

Gaute Nilsson

13. juni 2011

Oppgaveteksten

Autronica ønsker å se nærmere på muligheten til å presentere systemets totale tilstand i sanntid. Tanken er at operatøren skal ha mulighet til å se den overordnede tilstanden til systemet, og det bør være mulig å ta et "klinisk blikk" av hele systemet kun ved å se på et skjermbilde. Med andre ord: å presentere systemets overordnede tilstand i sanntid på én skjerm. Det er ønskelig å se på om en flokksimulering kan benyttes til oppgaven med å visualisere data.

Dette innebærer:

- Se nærmere på flocking-algorimter og emergente egenskaper.
- Utvikle en testbenk for å eksperimentere med parametere til en slik flokksimulering.
- Lage en visualisering av en flokksimulering
- Vurdere om en slik flokksimulering kan benyttes å visualisere data i store brannalarmsystem.

Forord

Arbeidet med denne oppgaven har foregått ved Institutt for Teknisk Kyernetikk ved NTNU.

I løpet av oppgaven har jeg lest meg opp på mange tema som er litt på sidelinjen av studiet, jeg har tilegnet meg god kunskap innenfor emnene grafikkprogrammering og parallellprogrammering av GPU'er.

Det rettes en takk til Autronica som har stått for kontorplass i arbeidsperioden.

Jeg vil også rette en stor takk til ekstern veileder i Autronica, Øyvind Teig som har tatt seg tid til å hjelpe til med veiledning til oppgaven. Øyvind har under hele prosessen vært en inspirasjonskilde og har bidratt med stor entusiasme.

Til slutt vil jeg takke Sverre Hendseth for god veiledning og mange konstruktive innspill det siste året. Sverre skal ha mye av æren for at oppgaven har kommet i mål.

Trondheim, 13. juni 2011

Gaute Nilsson

Sammendrag

På broen i et moderne cruise-skip kryr det av teknologiske innretninger for styring av forskjellige tekniske detaljer ved båten. Støynivået skal være lavt og utstyr som genererer unødvendig støy skal ikke forekomme. Derfor foretrekkes visuelle presentasjonssystemer som er rasjonelle og som ikke stresser mannskapet.

Mesteparten av båtens tekniske systemer presenterer en helhetlig oversikt over systemet via en skjerm, og muligens et pekredskap/tastatur oppe på broen. Ettersom moderne skip i dag er meget høyteknologiske resulterer dette i et stort antall systemer med tilhørende informasjonspresentasjonssystemer som alle krever oppmerksomhet fra tid til annen.

Autronica som ledene aktør innenfor branndeteksjon ønsker å se nærmere på alternative måter å skaffe en rask og helhetlig oversikt over deres system.

Oppgaven tar for seg et eksperiment hvor det studeres om det er mulig å presentere store datamengder ved å påtrykke spesielle parametere til et komplekst dynamisk system.

Innhold

1 Innledning **1**
- 1.1 Autronicas bakgrunn for oppgaven . . . 1
- 1.2 Relevans . . . 2
- 1.3 Avgrensninger . . . 2
- 1.4 Organisering av rapporten . . . 3

2 Bakgrunn **5**
- 2.1 Introduksjon til brannalarmanlegg . . . 5
  - 2.1.1 Brannsentral . . . 5
  - 2.1.2 Detektorer . . . 7
  - 2.1.3 Alarmorganer . . . 8
- 2.2 Kort om emergens og emergente egenskaper . . . 9
- 2.3 Parallellprogrammering . . . 12
- 2.4 Partikkelsystem . . . 12
- 2.5 GPU-programmering . . . 14
  - 2.5.1 GPU-utvikling . . . 14
  - 2.5.2 Programeringsinterface mot GPU . . . 14
- 2.6 OpenCL . . . 15
  - 2.6.1 Terminologi . . . 16
  - 2.6.2 Minnemodell . . . 17
  - 2.6.3 Kjøremodell . . . 18
  - 2.6.4 Platform layer API . . . 19
- 2.7 CUDA . . . 20
- 2.8 Numerisk integrasjon . . . 21
- 2.9 Romlige datastrukturer . . . 22
  - 2.9.1 Grid . . . 23
  - 2.9.2 Quad-trær og oct-trær . . . 24
  - 2.9.3 kd-trær . . . 25
- 2.10 Flokksimulering . . . 25

2.10.1 Boids . . . 25
2.10.2 En boids nærområde . . . 26
2.10.3 Autonome agenter og steering . . . 26
2.10.4 Separation . . . 27
2.10.5 Alignment . . . 28
2.10.6 Cohesion . . . 29
2.10.7 Flocking . . . 30
2.11 Programmeringsmiljø for kreativ programmering . . . 31
2.11.1 Processing . . . 32
2.11.2 OpenFrameworks . . . 32
2.11.3 Cinder . . . 32
2.11.4 Field . . . 33
2.11.5 NodeBox . . . 33
2.11.6 Impromptu . . . 34

3 Utvikling av testbenk til boid-eksperimenter 37
3.1 Noen eksisterende simuleringer . . . 37
3.1.1 Occoids . . . 38
3.1.2 xboids . . . 40
3.1.3 Tutorial Processing . . . 40
3.1.4 OpenSteerDemo . . . 41
3.1.5 Tutorial Cinder . . . 42
3.2 Valg av programmeringsmiljø . . . 42
3.3 Algoritmiske usikkerheter . . . 43
3.4 Maskinvare . . . 47
3.5 Litt om implementasjonen . . . 48
3.5.1 Grensebetingelser . . . 49
3.5.2 Hastighetsbegrensing . . . 50
3.5.3 Kraftbegrensing . . . 50
3.5.4 Screen tearing . . . 50
3.5.5 Normalisering av steering-vektorene . . . 51
3.5.6 Trails . . . 51
3.6 Optimalisering av boids-algoritmen . . . 52
3.6.1 Tidligere arbeid . . . 52
3.6.2 Lokasjonsoppslag . . . 52
3.6.3 skipThink . . . 55
3.6.4 Parallellisering . . . 56
3.7 OpenCL . . . 57
3.8 Loggdata . . . 59
3.9 Sanntidsdata . . . 60

4 Eksperimenter og resultater **61**
4.1 OpenCL 61
4.2 Testrigg 62
4.3 Resultater fra testriggen 62
4.3.1 Separation angle 63
4.3.2 Alignment angle 63
4.3.3 Cohesion angle 63
4.3.4 Separation weight 64
4.3.5 Alignment weight 64
4.3.6 Cohesion weight 64
4.3.7 Separation radius 65
4.3.8 Alignment radius 65
4.3.9 Cohesion radius 65
4.3.10 Endring av deler av flokkparameterene 66
4.4 Loggdata 66

5 Diskusjon **69**
5.1 Endring av parametere 69
5.2 Andre algoritmer 70
5.3 Boider brukt for å fange interesse 70

6 Konklusjon **71**
6.1 Videre arbeid 72

Figurer

2.1 Eksempel på en *glider* i Conway's Game of Life 11
2.2 Eksempel på bruk av partikkelsystem 13
2.3 OpenCL grunnprinsipp 15
2.4 Modell av en OpenCL-plattform [15] 16
2.5 Modell av minnehåndtering i OpenCL [15] 17
2.6 Synkronisering i OpenCL 19
2.7 Addisjon av to vektorer i OpenCL 19
2.8 Metoder for numerisk integrasjon 22
2.9 Bin-lattice database 24
2.10 Eksempel på quad-tre 25
2.11 En boids nærområde 26
2.12 Separation-oppførsel 27
2.13 Alignment-oppførsel 29
2.14 Cohesion-oppførsel 30
2.15 "Capture the Flag"-plugin i OpenSteerDemo 35

3.1 Skjermbilde fra Occoids 38
3.2 En boids synsfelt i Occoids 39
3.3 Skjermbilde fra xboids 40
3.4 Flocking i Processing 41
3.5 Boids plug-in i OpenSteer 42
3.6 Skjermbilde fra flock-tutorial i Cinder 43
3.7 Illustrasjon av separation-vektor 44
3.8 Sammenlikning av separation-vekting 46
3.9 Eksempel på Screen tearing (fra wikipedia) 51
3.10 *PSCrowd* 53
3.11 *Pigeons in the Park* 53
3.12 *Bin-lattice*-algoritme 55
3.13 *K nearest neighbors*-algoritme 55

3.14 *skipThink*-algoritme 56
3.15 Flokksimulering med OpenCL 58
3.16 Feilvisning av trails 59

4.1 Data over ett år fra noen utvalgte detektorer 67

Kapittel 1

Innledning

I større brannvarslingsanlegg er det behov for et toppnivå-system for å presentere data fra brannsentralene. Slike toppnivåsystem kjører som regel på egen maskinvare uavhengig av resten av brannvarslingsanlegget. Toppnivåsystem har som hovedoppgave å presentere alle detaljer rundt tilstanden til anlegget, inkludert data fra hver detektor i sytemet.

Å se en stor fugleflokk i naturen er en vakker og intressant opplevelse. Selv om en flokk av og til består av hundrevis eller kanskje tusenvis av fugler virker det som om flokken beveger seg som om den er én stor organisme. Enkeltindivider kolliderer aldri, selv om de bevger seg relativt nære hverandre. Bevegelsesmønsteret er meget dynamisk og det er alltid noe nytt å se på, man kan se på slike fugleflokker i lang tid uten at det blir kjedelig.

Hvis en flokksimulering kan benyttes til å presentere nyttige data samtidig som den er innbydende å se på, kan simuleringen benyttes som en diskré visulalisering til en operatør som kan ha interesse av dataene systemet har å presentere, men som ikke ønsker eller har mulighet til enhver tid å sette seg ned og studere systemet for å se om det inneholder noen intressante data og evt. av hvilken art de er.

1.1 Autronicas bakgrunn for oppgaven

I dag finnes et presentasjonssystem for styring og overvåkning av områder som er under branndeteksjon. Systemet tilbyr gjennom et grafisk brukergrensesnitt muligheter for kontroll og overvåkning av hvert eneste punkt i alle sløyfer som er inkludert i brannalarmanlegget. For visualisering av intressante tilstander vil en dynamisk farge tildeles det korresponderende området på en plantegning, og ved en eventuell alarm vil lokasjonen enkelt

kunne identifiseres på plantegningen.

Dagens system fungerer godt til å presentere data på detaljnivå. Det gir enkel og nøyaktig visning av enkelthendelser i systemet samtidig som det oppgir alle intressante detaljer omkring hendelsen. Systemet er ikke utviklet med tanke på å kunne gi en raskt og effektiv oversikt av tilstanden til hele systemet. For å skaffe seg en slik oversikt i dag baserer man seg gjerne på automatisk generete tilstandsrapporter. Eksempler på dette kan være:

- Servicerapporter som forteller hvilke detektorer som er nedstøvet og trenger og byttes.

- Rapporter over "problemdetektorer", dette kan være punkter som ofte går inn og ut av forvarsel eller melder falsk alarm hyppig. Disse punktene og deres respektive alarmer oppleves gjerne som støy, og problemet er som regel at punktene har feil konfigurasjon, er dårlig plassert eller rett og slett er av feil type i forhold til applikasjonen.

- Rapporter over ustabile detektorer, dette kan være detektorer som sporadisk melder feil eller punkter som ofte melder teknisk feil, for eksempel branndører som går tregt og bruker for lang tid på å lukke.

1.2 Relevans

Autronica legger store ressurser på å være ledende i markedet innenfor branndeteksjon og varsling. Noe av det som gjør Autronica markedsledende er deres toppnivåsystem som tilbyr god oversikt over områdene som overvåkes. Dette systemet er under kontinuerlig utvikling for å sikre at det står i stil med dagens strenge krav og høye forventninger blant kundene. Som en naturlig del denne utviklingen vil det være intressant å se på nye måter å fremstille data fra anlegget som kan være nyttige for en operatør.

En bonus ved en slik nyvinnig vil være at den lett tiltrekker seg oppmerksomhet på messer o.l. fordi en slik visualisering ikke forventes å sees i forbindelse med toppnivåsystemer for brannvarsling.

1.3 Avgrensninger

Oppgaven skal ikke fokusere på den visuelle kvaliteten ved arbeidet. Arbeidet skal først og fremst fokusere på i hvilken grad en slik visualisering egner seg til å visualisere data.

Det skal ikke være behov for operatør og interaktere med visualiseringen for å hente ut data.

Visualiseringen skal i første omgang begrense seg til å foregå i to dimensjoner.

1.4 Organisering av rapporten

Denne oppgaven består seks kapitler: Kapittel 1 er innledningen til rapporten og sier noe om oppgavens aktualitet og inneholder Autronicas bakgrunn for oppgaven.

Kapittel 2 presenterer relevant teoretisk bakgrunn og fungere som fundament for resten av oppgaven.

Kapittel 3 beskriver utvikling av en testbenk for å utføre eksperimenter på flokksimulering.

Kapittel 4 presenterer resultater fra utvikling av testbenken og eksperimenter gjort med testbenken.

Kapittel 5 tar for seg diskusjon rundt eksperimentene som er gjort i testbenken og egene tanker om hvorvidt en slik flokksimulering er egnet til å visualisere data.

Til slutt presenteres konklusjonen av oppgaven og videre arbeid i kapittel 6.

Kapittel 2

Bakgrunn

2.1 Introduksjon til brannalarmanlegg

Følgende er i hovedsak hentet fra [4].

Hovedoppgaven til et brannalarmanlegg er å kjenne igjen en brannsituasjon innenfor rimelig tid etter den har oppstått, for så å varsle om en nødsituasjon. Brannalarmanlegg kan tilby ulike funksjoner basert på grad av alvor ved brann, type bygning og dens bruksområde, antall folk i bygningen osv. For det første skal et brannalarmanlegg kunne oppdage en brann via manuelle eller automatiske meldere, og for det andre skal det gi beskjed til folk i bygningen om at det har oppstått en brann og at bygningen må evakueres. Det er også vanlig at et brannalarmanlegg tilbyr ekstern varsling, som f.eks. varsel til brannvesen eller andre innstanser som er i beredskap. I tillegg kan et større brannalarmanlegg også styre bygningstekniske funksjoner.

2.1.1 Brannsentral

Brannsentralen er "hjernen" brannalarmanlegget. Den er ansvarlig for å overvåke de forskjellige innenhetene slik som manuelle og automatiske brannmeldere, for så og aktivere utenhetene slik som klokker, sirener, nødlys og bygningstekniske finesser ved en eventuell brann. Slike system kan strekke seg fra "dumme" panel med én inngangs- og utgangssone til store mikroprosessorstyrte anlegg med tusenvis av detektorer. Brannsentraler er i hovedsak bygd på to forskjellige teknologier: konvensjonell og addreserbar topologi. Konvensjonelle brannalarmanlegg har i mange år vær industristandarden innenfor brannvarsling. I et konvesjonelt system legges det en sløyfe gjennom området som er under overvåkning. Langs sløyfen hektes det på én eller flere brannmeldere. Som regel pleier sløyfene å være delt opp i naturlige so-

ner, for eksempel én sone for hver etasje i mindre næringsbygg, eller én sone for et område avgrenset av branndører.

Et *punkt* er en hvilken som helst enhet som er koblet til sløyfen. Hvis det oppstår en brann vil én eller flere punkter aktiveres. Dette slutter detektorsløyfen og brannsentralen gjenkjenner dette som en nødsituasjon. Den vil da aktivere de tilhørende alarmutgangene hvor klokker eller sirener som regel er plassert. For å sikre at det ikke er brudd på noen av sløyfene vil brannsentralen sende en hvilestrøm gjennom hver sløyfe. Ved et kabelbrudd vil hvilestrømmen opphøre og brannsentralen kjenner igjen situasjonen som en feil. Den vil da indikere at den aktuelle sløyfen inneholder en feil og muligens er ute av stand til å varsle en brann.

Ulempen med konvensjonelle system er at de ikke kan si hvilken enhet på sløyfen som er blitt aktivert eller i tilfelle feil, hvor i sløyfen feilen ligger. Konvensjonelle system er derfor tidkrevende å vedlikeholde. Regler sier at automatiske detektorer skal testes med jevne mellomrom for å sikre at de fungerer. Alle detektorene må rutinemessig renses og rekalibreres for å sikre at systemet fungerer optimalt. Ved en feil må servicepersonell systematisk gjennomgå hele sløyfen til feilen er funnet.

Addresserbare eller "intelligente" system er i dag den ledende teknologien med tanke på funksjonalitet og skalerbarhet. I likhet med konvensjonelle system vil anlegget være bygd opp av en eller flere sløyfer som inneholder enhetene som inngår i systemet. Sløyfene i et addresserbart system kan også inneholde alarmorganer, i motsetning til et konvensjonelt system der disse er plassert på en egen sløyfe.

Hovedforskjellen til et konvensjonelt system er at hvert punkt på sløyfen kan identifiseres med sin unike addresse. Informasjon om hvert enkelt punkt som type enhet, plassering, nødvendig respons ved aktivering osv. er koblet opp mot punktets respektive addrese i brannsentralen. Sentralen mottar data kontinuerlig slik at en statusendring på sløyfen umiddelbart vil bli fanget opp. Et addresserbart system vil også overvåke sløyfen slik at eventuelle feil blir detektert. En stor fordel med slike system er at de gir mulighet til å lokalisere feil ute på sløyfen. Istedet for å kun rapportere om en feil på sløyfen, kan de også identfisere på hvilket punkt, eller mellom hvilke punkter feilen ligger.

Når en detektor blir nedstøvet vil systemet forsøke å kompensere for dette ved å øke eller minke sensitiviteten til detektoren. Når det er kompensert såpass mye at signal-/støyforhold er under en gitt grense, vil sentralen melde fra om dette, slik at det kun vil bli foretatt service på detektorer som trenger det. Hvis et addresserbart system skal utvides innebærer dette at man legger opp en ny sløyfe eller utvider en gammel sløyfe der det er mulig.

Brannsentralen må da omprogrameres så den kjenner igjen de nye enhetene.

2.1.2 Detektorer

Mennesket er utmerket til å oppdage en brann. En normal person vil være i stand til å kjenne igjen flere karakteristiske trekk ved en brann, inkludert røyk, flamme, varme og lukter. Derfor er de fleste brannalarmsystemer utstyrt med en eller flere manuellmeldere som skal brukes om man oppdager brann før brannalarmanlegget. Desverre kan manuell brannmelding være upålitelig, spesielt når mennesker ikke er tilstede når en brann starter, eller ikke er i stand til å kjenne igjen karakteristiske trekk ved en brann (nedsatt syn, osv.). På bakgrunn av dette er det utviklet mange typer automatiske branndetektorer. Disse kan reagere på varme, røyk, flammer, spesielle gasser, osv. Når det velges riktig detektor i forhold til applikasjonen kan disse være en meget pålitelig måte og oppdage brann på.

Manuell deteksjon er den eldste formen for brannvarsling. I sin simpleste form vil en person som roper ut gi et brannvarsel. I en bygning derimot vil ikke lyden trenge gjennom til alle deler av bygningen. Manuellmeldere gir folk i bygningen mulighet til å løse ut brannalarmsystemet, som vil gi brannvarsel til alle nødvendige deler av bygningen. Ulempen med manuellmeldere er at de har ingen verdi hvis lokalet er tomt for folk, de kan også brukes til å gi falske alarmer med vilje. Likevel er manuellmeldere en viktig komponent i et hvert brannalarmsystem.

Varmedetektorer er den eldste typen automatiske detektorer, og ble tatt i bruk på midten av 1800-tallet. Den mest vanlige typen er detektorer som løser ut når temperaturen passerer en gitt grense (vanligvis rundt 57 °C - 74 °C). En annen vanlig type varmedetektor løser ut når temperatur*endringen* overskrider en gitt grense, det vil si når den registrerer en unormalt rask temperaturstigning. Disse typene er punktdetektorer, noe som gjør at de gjerne må plasseres i taket eller høyt på en vegg. Det finnes også et tredje prinsipp som ikke lenger er i utstarkt bruk i dag. Dette prinsippet baserer seg på en lang kabel med to ledere i. Isolasjonen mellom lederene er konstruert for å bryte sammen når temperaturen overskrider en gitt grense. Varmedetektorer er meget robuste og billige og vedlikeholde. Ulempen med varmedetektorer er at de ikke løser ut før omgivelsene har nådd en kritisk temperatur, hvor brannen sansynligvis er godt i gang og hvor skadene øker eksponensielt.

Røykdetektorer baserer seg på en nyere teknologi. Disse ble utbredt rundt 1970-tallet. Den mest vanlige røykdetektoren er punktdetektoren, disse plasseres i taket eller høyt på en vegg i likhet med varmedetektorer. Slike

detektorer benytter i hovedsak to prinsipper for å måle røyknivå: optisk eller ionisering. Begge prinsippene har hver sine spesielle fordeler under forskjellige omstendigheter. For store og åpne lokaler kan såkalte linjedetektorer benyttes. Slike detektorer består av to deler, en sender og en mottaker som er montert et stykke fra hverandre (opp til ca. 100 m). Senderen gir ut en lysstråle som mottakeren analyserer. Hvis det oppstår røyk vil ikke mottakeren ta i mot lysstrålen med full styrke og dermed gå i alarm. En tredje type røykdetektor som brukes i spesielt krevende miljø er en aspirasjonsdetektor. Denne består av en hovedenhet med aspirasjonsvifte samt et kammer for analysering av luften, og et nettverk av små rør for å trekke inn luft gjennom. Luft som blir hentet gjennom de forskjellige rørene blir sugd inn til deteksjonskammeret og analysert for røykinnhold før den blir blåst ut tilbake til omgivelsene. Slike detektorer er ekstremt følsomme, og er den raskeste av alle detektorer til å oppdage røyk. Aspirasjonsdetektorer kan benyttes i lokaler der det legges større vekt på estetikk da rørene er små og lettere å skjule sammenlignet med andre detektorer.

Hovedfordelen med røykdetektorer er at de kan oppdage en brann som fortsatt er i startfasen. Dette gjør at utrykningsmannskaper raskere kan komme til og få kontroll på brannen før det oppstår større skader. Røykdetektorer er den foretrukne detektortypen når liv og helse står på spill. Ulempen med røykdetektorer er at de er dyrere og har lettere for å gi falske alarmer sammenlignet med varmedetektorer. Men når de er korrekt plassert og har riktig sensitivitet er røykdetektorer en meget pålitelig enhet med liten relativt lav sjanse for å gi falsk alarm.

Flammedetektorer er en gruppe detektorer som benytter et *pyroelektrisk element* til å detektere flammer. Slike detektorer ser etter karakteristiske trekk ved en brann i det infrarøde spekteret. Enheten kjenner igjen den spesielle infrarøde frekvenssignaturen en flamme avgir. Fordelen med flammedetektorer er at de er meget robuste i krevende miljø. Typsike bruksområder er i maskinrom, oljeraffinerier, gruver og andre steder hvor det gjerne er varmt og støvete. Ulempen med flammedetektorer er at de er dyre og krever ofte vedlikehold (fjerning av støv fra linsen). I tillegg må flammedetektorer peke direkte mot brannkilden i motsetning til varme- og røykdetektorer.

2.1.3 Alarmorganer

Når brannsentralen har mottatt en melding om brann må den fortelle at en nødsituasjon er under utvikling. Alarmorganer består som oftest av forskjellige lyd- og lysenheter. Brannklokker er mest vanlig og gir en velkjent alarm. Sirener er også vanlige alarmorganer som benyttes når mindre områder skal

varsles. Ved små områder som for eksempel soverom eller andre steder det er ikke behov for kraftig alarm kan det benyttes alarmsummere. Høytalere benyttes hvor det er ønskelig at en forhåndsinnspilt stemme forteller om situasjonen og hvordan evakueringen skal foregå. Dette foretrekkes i store lokaler, cruise-skip eller andre bygninger hvor det er nødvendig med evakuering i flere steg.

Når det gjelder visuelle alarmer benyttes det som regel blinkende lys eller strober. Slike enheter benyttes i områder hvor det er høy bakgrunnsstøy, områder hvor det benyttes hørselvern eller i områder det kan oppholde seg folk med nedsatt hørsel.

En viktig funksjon til brannsentralen er muligheten til å gi ekstern varsling til en brannstasjon eller et beredskapsteam. Dette gjøres vanligvis via mobilnettet eller en egen dedikert fast linje.

Andre responsfunksjoner ved en brannalarm kan være å sende alle heiser til bakkenivå, slå av ventilasjon (stenge tilførsel av frisk luft), åpne røykluker, dempe lydanlegg (f.eks. i et diskotek), stoppe flyt av kjemikalier inn til en alarmsone (f.eks. ved et oljeraffineri), osv. Det kan også være aktuelt å aktivere bygningens sprinklersystem eller andre dedikerte slukkesystem.

2.2 Kort om emergens og emergente egenskaper

En emergent oppførsel eller emergent egenskap kan opptre når et antall enheter (agenter) opererer i et miljø og hvor de tilsammen utøver en mer kompleks oppførsel som en helhet.

Ordet kompleks er et synonym for komplisert, men det brukes også i en mer teknisk forstand til å henvise til systemer med et relativt lite utvalg av relativt homogene agenter som virker på hverandre og følger enkle lokale regler for interaksjon og komunikasjon uten global eller sentralisert styring [37].

Aristoteles [3] har en av de tidligste (350 f.k.) definisjonene på det vi i dag kaller emergente systemer:

> *"Things which have several parts and in which the totality is not, as it were, a mere heap, but the whole is something beside the parts"*

I dag blir dette vanligvis formulert som: "Helheten er mer enn summen av delene" [37].

Global økonomi, bikuber, finansmarkedet, gresshoppesvermer, massehysteri, vegnett, trafikkorker, bakteriell infeksjon, byplanlegging, evolusjon og

Internett er alle eksempler på emergente fenomen hvor en samling enkeltindivider gjør lokale handlinger uten noen form for overordnet styring og hvor "helheten" av handlingene ikke er uttrykkelig planlagt.

Et simpelt eksempel fra naturen kan være termitter som vi tenker oss følger disse to enkle reglene [18]:

- Hvis du ikke bærer en treflis og finner en, plukk den opp.
- Hvis du bærer en treflis og finner en annen, legg din ned.

Med dette vil en treflis som har blitt lagt ned av en tremitt gi et signal til etterfølgende termitter om å legge deres trefliser ned. Hver ny flis gjør haugen mer iøyenfallende og sannsynligheten for at haugen blir oppdaget av andre termitter øker raskt.

Typiske egenskaper hos systemer med emergent oppførsel [5]:

- Emergente system er robuste. Det finnes ikke noe svakt ledd som kan lede til at systemet feiler. Hvis en bit av systemet feiler eller blir fjernet vil systenet fortsatt fungere og utøve emergente egenskaper.
- De er godt tilpasset et "rotete" miljø med mange usikkerheter. Andre menneskelagde system kan være "optimale", men krever ofte mye tid til utvikling og er gjerne sårbare hvis omgivelsene divergerer mye fra utgangspunktet. Emergente system trenger ingen helhetlig oversikt eller kjennskap for å kunne nå et mål på et høyere plan.
- De finner en rimelig løsning raskt for så og optimalisere. I den virkelige verden må valg gjøres raskt mens de fremdeles er relevante. Typiske datamaskinalgoritmer pleier ikke å komme opp med et brukbart svar før de er kjørt ferdig.

Conway's Game of Life er et simpelt sett med regler laget for *Cellulær Automata*[1] og ble funnet opp av John Horton Conway i 1970. "Spillet" er et *zero-player* spill, som betyr at progresjonen i spillet bestemt av starttilstanden [9]. Spillet foregår på et uendelig todimensjonalt rutenett med firkantede celler som hver kan ha to mulige tilstander, *levende* eller *død*. Alle cellene ser på sine åtte naboer. For hvert steg i simuleringen følges disse reglene.

1. Hver levende celle med ingen eller én nabo dør (av ensomhet)

[1] En *cellulær automat* ("cellular automata") er en diskret modell i form av et (ofte uendelig) rutenett med celler, hvor én celle har et endelig antall tilstander, f.eks. "av" og "på" [47].

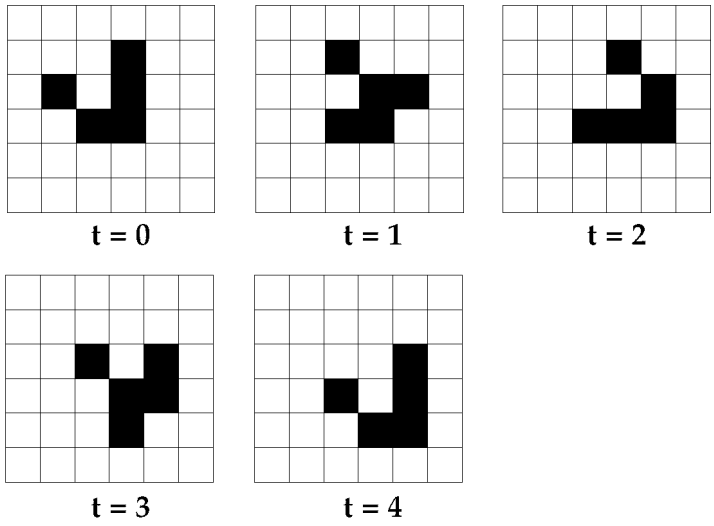
2. Hver levende celle med fire eller flere naboer dør (av overbefolkning)

3. Hver levende celle med to eller tre naboer overlever[2]

4. Hver død celle med tre naboer kommer til live

Reglene utføres samtidig på alle celler i systemet.

Kun ved å følge disse enkle reglene kan det utarte seg meget komplekse mønstre. I figur 2.1 vises et eksempel på en *glider* som brer seg utover i spillet.



Figur 2.1: Eksempel på en *glider* i Conway's Game of Life

Conway's Game of Life har fått mye oppmerksomhet grunnet de overaskende måtene mønstre i spillet utvikles på. Dataforskere, fysikere, biologer, økonomer, matematikere, genforskere, filosofer og flere vitenskapsgrener har rettet mye forskning rundt spillet for å studere hvordan komplekse mønstre *emergerer* ut fra implementasjonen av de enkle reglene. Nyere forskning har til og emd bevist at Game of Life er Turing-komplett[3] [6], dette vil si at alt som kan regnes ut algoritmisk kan regnes ut i Game of Life.

[2]Denne regelen er redundant, og er gitt av regel 1 og 2

[3]Et system i stand til å beregne alle funksjoner en Turing-maskin kan, inkludert addisjon, subtraksjon, multiplikasjon og betinget iterering

2.3 Parallellprogrammering

I sin enkleste form betyr parallprogrammering å benytte flere ressursser samtidig for å løse et gitt problem. Flere ressurser vil i denne konteksten bety flere prosessorer som kan utføre beregninger.

Noen problem er bedre egnet for dette enn andre. For eksempel hvis et steg i algoritmen er avhengig av svaret fra tidligere steg for å kunne fortsette, vil dette gjøre at punktet må vente til dette svaret er tilgjengelig. Denne delen av algoritmen er sekvensiell og krever en synkronisering mellom prosessorene. Amdahl's lov [2] sier at maksimal hastighetsøkning ved å parallellisere et problem er gitt av:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{2.1}$$

Hvor P er andelen av programmet som er parallelisérbar og N er antall prosessorer. $(1-P)$ er da den sekvensielle biten av programmet. Som et eksempel: hvis P er 90 %, så vil $(1-P)$ være 10 % og hastighetsøkningen vil maksimalt være 10, uavhengig av antall prosessorer som benyttes. Loven gir oss at: uavhengig av antall prosessorer som benyttes vil programmet aldri kjøre raskere enn den sekvensielle delen av programmet. Andelen av programmet som er *parallelliserbar* har derfor stor innvirkning på hvilken hastighetsøkning som kan oppnås.

2.4 Partikkelsystem

Partikkelsystem kan benyttes til mer eller mindre hva som helst, og som en generell regel vil et partikkelsystem bestå av et antall enheter, relaterte eller urelaterte til hverandre, som beveger seg i henhold til et sett med regler [7].

Et eksempel på hva et partikkelsystem kan benyttes til er visualisering av vann i en fontene, som vist i figur 2.2. Én partikkel representerer én vanndråpe. Partiklene har en utgangsfart fra senter av fontenen og beveger seg i forhold til gravitasjonskraften.

Det fins to typer partikkelsystem: tilstandsløse og tilstandsbevarende (*stateless* og *state-preserving*) system [20]. Tilstandsløse system lagrer ikke nåværende posisjon og andre data om partiklene. Oppdatert posisjon til hver partikkel blir regnet ut av en matematisk funksjon på bakgrunn av startposisjon og simuleringstiden. Tilstandsbevarende system muliggjør bruk av numeriske integrasjonsmetoder for å regne ut oppdaterte verdier for partiklene, basert på tidligere data. Ved å bruke denne teknikken er det også

mulig å utføre kollisjonsdeteksjon med faste objekter i scenen eller regne ut innbyrdes interaksjon mellom partikler.

Figur 2.2: Eksempel på partikkelsystem, her brukt til å simulere vann i en fontene [25]

Hver ny partikkel som genereres har gjerne et sett med faste parametere. Disse kan være: startposisjon, utgangsfart, farge ved start, startstørrelse, gjennomsiktighet ved start, form og levetid [27].

Etter en partikkel er laget kan kan man for eksempel justere fargen med en grad-av-fargeendring parameter, likedan med gjennomsiktigheten.

Ofte har partikler i et partikkelsystem et begrenset livsløp. Når en partikkel blir laget kan det spesifiseres en levetid. For hvert steg i simuleringen vil denne parameteren minke. Når levetiden er null, blir partikkelen slettet. En annen teknikk kan være å se på farge/gjennomsiktighet til partikkelen. Hvis denne går under et gitt nivå blir partikkelen slettet. Nok en teknikk kan være å slette partikkelen hvis den har beveget seg utenfor interesseområdet til simuleringen eller etter farten har avtatt til et gitt nivå. Som regel når en partikkel slettes vil en ny bli laget og sendt ut fra en bestemt startposisjon, ofte kalt *partikkelemitter*.

For å simulere bevegelsene til en partikkel må det regnes ut krefter som virker på partikkelen. Globale og lokale krefter legges sammen til én vektor

slik at akselerasjonen kan finnes ut fra klassisk fysikk som vist i ligning (2.2).

$$\boldsymbol{a} = \frac{\boldsymbol{F}}{m} \tag{2.2}$$

Deretter blir akselerasjonsvektoren numerisk integrert og man finner ny fart og posisjon. Dette er beskrevet nærmere i avsnitt 2.8.

Til slutt blir hver partikkel tegnet på skjermen med sin oppdaterte posisjon og i henhold til spesifiserte attributter som for eksempel farge/gjennomsiktighet og størrelse.

Avhengig av måten hver partikkel tegnes på kan partikler skjule andre partikler som ligger bak dem, og de kan kaste skygger på andre partikler. En mye brukt teknikk er å la partiklene oppføre seg som lyskilder. Da tegnes partiklene additivt på skjermen i henhold til deres verdier for farge og gjenomsiktighet. Dette eliminerer problem med at noen partikler skjuler andre partikler fordi to overlappende partikler vil da kun legge til mer lys til en gitt piksel.

2.5 GPU-programmering

GPU-programmering eller GPGPU (General-Purpose computing on Graphics Processing Units) betyr å ta i bruk en GPU for å utføre beregninger til generelle formål, det vil si andre ting enn grafikk.

Filosofien bak GPU-programmering er å la CPU og GPU arbeide sammen og la den sekvensielle delen av programmet kjøre på CPU'en mens den regnetunge (og paralleliserbare) delen tas hånd om av et eller flere grafikkort.

Prossessen med å rendre en 3D scene innebærer mange gjentatte transformasjoner og fargeberegninger på vertex'er og piksler. Fordi at dette er individuelle operasjoner er det mulig å dele disse opp og "streame" operasjonene mellom prosesser. Stream prosessering er form for parallellprossesering (SIMD) hvor separate prossessorer ikke behøver å kommunisere eller synkronisere. For å utnytte denne egenskapen blir GPUer bygd med mange prosessorer (gjerne hundretalls).

2.5.1 GPU-utvikling

2.5.2 Programeringsinterface mot GPU

Nvidia lanserte CUDA (Compute Unified Device Architecture) i februar 2007. Dette var et av de første GPGPU-grensesnittene som åpnet for høynivå komunikasjon med GPU. Tidligere var slik interfacing blitt gjort gjennom

OpenGL eller DirectX som var utviklet med fokus på 3D-grafikk og ikke GPGPU.

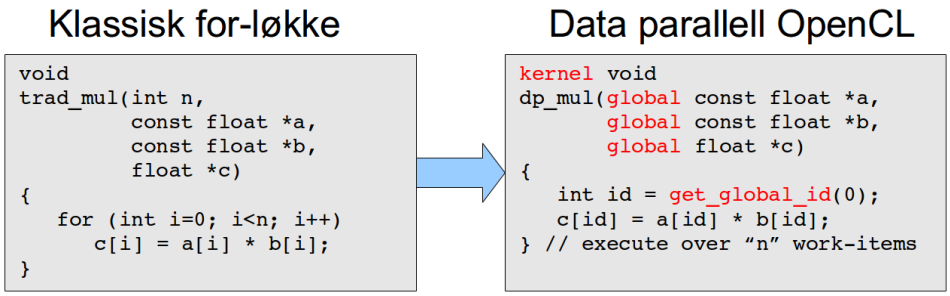
Ikke lenge etter Nvidia hadde lansert CUDA, tok AMD sitt GPGPU-rammeverk (som da het "Close to Metal") ut av beta og lanserte ATI Stream i desember 2007.

I juni 2008 gikk representanter fra bl.a. AMD, IBM, Intel og Nvidia sammen og dannet Khronos Group. Khronos Group og Apple samarbeidet så om å utvikle et rammeverk for å skrive og kjøre programmer på tvers av hetrogene plattformer som består av CPU'er, GPU'er og andre prosessorer [44]. Fem måneder senere var spesifikasjonene til OpenCL (Open Computing Language) klare.

I august 2009 lanserte Apple Mac OS X Snow Leopard som inneholder en full implementasjon av OpenCL. OpenCL var da Apples svar på et rammeverk for å kunne utnytte all den tilgjengelige prosessorkapasiteten som ligger i GPU'en [44].

2.6 OpenCL

Følgende er hentet fra [16], [13], [15] og [1].
Khronos har med OpenCL følgende forretningsmessige mål:

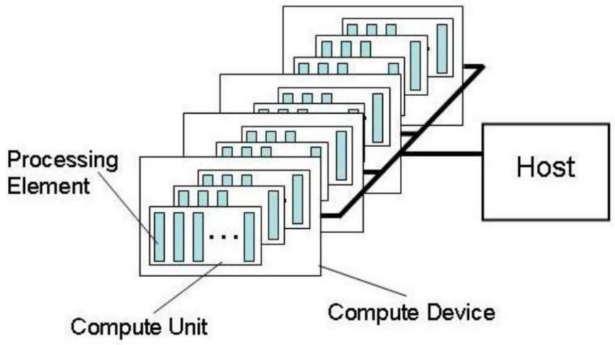


Figur 2.3: Grunnprinsippet for OpenCL, istedet for prosessering i en for-løkke foregår databehandlingen i parallelle tråder med hver sin unike id

- Øke markedsområdet for parallell databehandling
- Lage et grunnleggende utgangspunkt for parallell databehandling
- Være en helhetlig programmeringsmodell for CPU'er GPU'er Cell, DSP og andre prosessorer

- Støtte et bredt spekter av applikasjoner, fra mobile enheter til høyytelses databehandling (tungregning)
- Portabilitet av programmer mellom flere systemer fra forskjellige produsenter
- Være tett integrert med prosessorer helt ned til silisumsnivå

2.6.1 Terminologi



Figur 2.4: Modell av en OpenCL-plattform [15]

OpenCL-standarden navngir en hvilken som helst enhet som er i stand til å utføre beregninger som en *compute device*. Hver compute device består av én eller flere regneenheter, disse kalles *compute units*. Antall compute units tilsvarer antall uavhengige instruksjoner en compute device kan utføre samtidig. En firekjernes CPU består av fire compute units.

En CPU i dag består som regel av to til åtte compute units, mens en GPU består av mange flere compute units, vanligvis fra et titall på små GPU'er og opp til mange hundre på litt krafigere GPU'er. Om det er en CPU med åtte compute units eller en GPU med 100 compute units regnes hver av disse kun som én compute device. Hierarkiet er illustert i figur 2.4.

Programmet som kjører OpenCL-funksjonene og styrer den overordnede prosessen blir kjørt på en *host*-maskin. Det befinner seg kun én host i en OpenCL-applikasjon.

En *kernel* er et sett med instruksjoner skrevet i OpenCL C-language som er tiltenkt å kompilere og kjøre på en compute device. En kernel blir lagt i en *command queue* av host, men må kompileres separat slik at maskinkoden kan bli tilpasset enheten den skal kjøre på. Man kan skrive Kernel-kildekoden som en egen fil, eller inkludere den inline i kildekoden til host-programmet.

En instans av en kjørende kernel kalles et *work item*. Et slikt work item er i praksis en kjørende tråd som er tildelt sin unike id. Flere like work items grupperes i en *work group*. I CUDA sin terminologi kalles et work item for *thread*, og en work group for *thread block*.

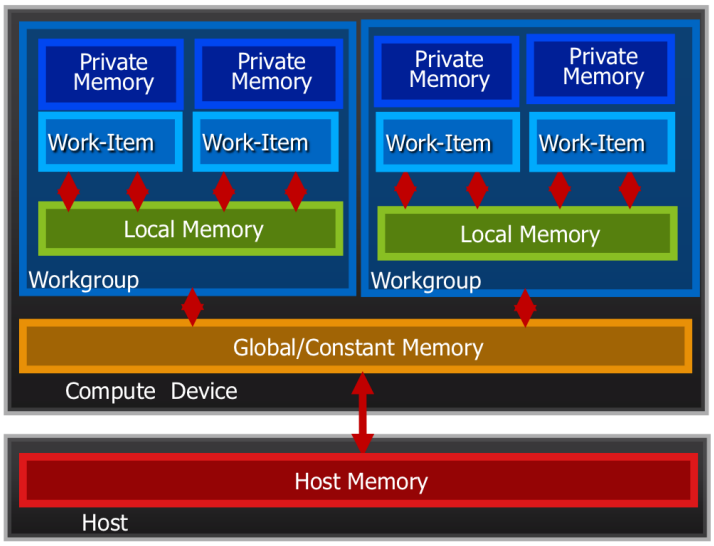
Det er nødvendig å ha en *command queues* for å sende arbeid til en compute device. Køene bestemmer rekkefølgen på kjøring av kernels og minneoperasjoner.

En *context* er miljøet som work items blir kjørt i. En context inneholder et antall compute devices med sine respektive minneområder og command queues. Det er nødvendig å sette opp en context for at man kan dele minne mellom compute devices.

Program objects er en datatype som representerer en OpenCL-kernel som skal kjøres. Den inneholder informasjon om context'en en kernel kjøres på, programkoden til kernelen og en liste over hvilke compute devices kernelen kan kjøres på.

2.6.2 Minnemodell

OpenCL benytter en *relaxed memory consistency model*. Modellen klassifiserer de forskjellige typene minne som er tilgjengelig i fem forskjellige klasser. En oversikt over minneområdene er gitt i figur 2.5.



Figur 2.5: Modell av minnehåndtering i OpenCL [15]

- *Host memory* er minne som ligger på host-maskinen og er adskilt fra compute devices.
- *Global memory* er tilgjengelig fra alle work items i alle work groups. Dette tilsvarer off-chip minne på en compute-device.
- *Constant memory* som er et område i global memory som er avsatt til read-only aksess fra work items og vil bli holdt read-only under kjøring av en kernel.
- *Local memory* er tilgjengelig for alle work items i en work group og benyttes til å lagre data som deles mellom work itmes.
- *Private memory* er minne som kun brukes i ett enkelt work item. Dette tilsvarer registre i en enkel compute unit eller CPU-kjerne.

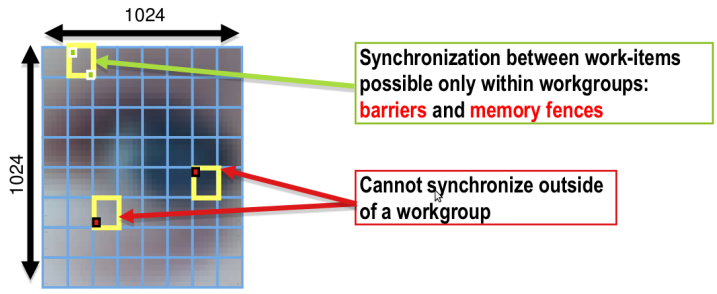
All minnehåndtering i OpenCL er eksplisitt. Data må flyttes fra host → global → local → ... og tilbake igjen til host når prosesseringen er ferdig. Det er kun Global/Constant-minnet som kan aksesseres direkte fra host.

2.6.3 Kjøremodell

Kjøring av et OpenCL-program innebærer parallell kjøring av instanser av en kernel på en eller flere OpenCL-devices som blir styrt av et host-program. Hver kernel gjør de samme operasjonene, men på forskjellige deler av datasettet. Når man setter opp command queues for kjøring av kernels, setter man opp antall instanser som skal kjøre for å behandle alle data. Dette kalles et *index space*. OpenCL støtter index spaces i opp til tre dimensjoner. For eksempel hvis det skal behandles et bilde på 64x64 piksler vil man gjerne sette opp et todimensjonalt index space med størrelse 64x64, som resulterer i 4096 work items. Antall work items er praktisk talt ubegrenset, man bruker det antallet som passer best til algoritmen som kjøres.

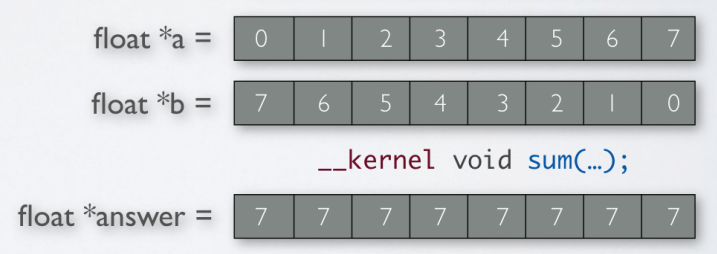
OpenCL støtter synkronisering av work items innad i en work group, men tillater ikke synkronisering mellom forskjellige work groups. Dette er illustrert i figur 2.6. Start av kernels og eventuelle videre handlinger som ligger i command queues kan synkroniseres fra host. Hvert work item har sin unike *global ID* som tilsvarer plasseringen i index spacet. Eksempelvis har et work item i et todimensjonalt index space som er nr 9 på x-aksen og 3 på y-aksen en global ID: [9,3]. På samme måte har hver work group også sin unike *work group ID*.

I figur 2.7 vises et eksempel på en kernel som adderer et endimensjonalt datasett bestående av to vektorer av dimensjon 8. Her kjøres åtte work items



Figur 2.6: Synkroniskering av work-items og work groups i OpenCL, som her opererer på et bilde på 1024x1024 piksler [15]

parallelt. Kernel-koden er gitt i listing 2.1. Hvert work item leser inn data fra global (constant) memory, utfører en addisjon for så og legge data tilbake i global memory. `id`-variabelen blir lagret i private memory. Siden dette er et endimensjonalt problem ber vi om global ID i første dimensjon (0) i index spacet.



Figur 2.7: Addisjon av to vektorer i OpenCL

2.6.4 Platform layer API

Host-maskinen må ta seg av valg av compute devices og sette opp disse rikig. Et host-system kan gjerne bestå av flere compute devices som har forskjellige egenskaper (DSP, FPGA, CPU, GPU, ...) og det er viktig å velge den enheten som er best egnet til å løse det aktuelle problemet.

På host-siden gjøres disse stegene med API-et for å sette opp et OpenCL-program:

1. Søk etter compute devices
2. Lag contexts

Listing 2.1: Kernel-kode skrevet i OpenCL C for vektoraddisjon

```
1 kernel void sum(global const float *a,
2                 global const float *b,
3                 global float *answer)
4 {
5    size_t id = get_global_id(0);
6    answer[id] = a[id] + b[id];
7 }
```

3. Lag minneobjekter som blir assosiert med contexts
4. Kopier data som skal prosesseres over til device
5. Kompiler og flytt over kernel program objects
6. Send ut kommandoer til command queue (kjøring av kernels, synkronisering, ...)
7. Kopier tilbake prosesserte data
8. Frigi OpenCL-ressurser

I den nyeste OpenCL-spesifikasjonen tillates også at flere host-programmer interfacer samme OpenCL-device samtidig.

2.7 CUDA

CUDA er navnet på regnemotoren i Nvidia's grafikkort som er tilgjengelig for bruk med GPGPU-programmering. Nvidia har utviklet "C for CUDA" som er et vanlig C-kodespråk, men med utvidelser for CUDA-arkitekturen.

All kode som blir skrevet for CUDA, både API-kall og kernels kompileres med Nvidias egen C-kompilator: `nvcc.exe`. I første generasjon av CUDA-arkitekturen, G80, var det kun støtte for standard C-kode noe som resulterte i flere problemer når CUDA-funksjonalitet skulle integreres i C++-applikasjoner. Nvidia har gitt ut et simpelt eksempel på hvordan CUDA kan integreres med C++-applikasjoner, men integrasjonen hadde flere ulemper som bl.a. er diskutert i [8].

Dagens CUDA-generasjon, Fermi, har blitt redesignet fra grunnen av for å kunne tilby native støtte til flere programmeringssråk, deriblant C++.

Kode som er skrevet i C for CUDA er kun kompatibel med Nvidias GPUer. Det eksisterer også forskjellige grader av CUDA-kompatibilitet innad mellom GPU'ene til Nvidia, omtalt som *compute capability*. Eksempelvis

så vil dagens GPUer som støtter CUDA compute capability 2.0 ha bredere funksjonalitet enn de første CUDA-akselererte GPUene som kom i 2006 (G80-arkitektur) som da hadde compute capability 1.0.

I følge [17] er den praktiske ytelsen til CUDA høyere enn OpenCL under minnetransaksjoner mellom host og device. Dette skyldes at det er behov for litt større transaksjoner med OpenCL, sannsynligvis fordi OpenCL følger mindre plattformspesifikke prosedyrer enn CUDA. Selve ytelsen til kernels under regneoperasjonene er relativt lik.

2.8 Numerisk integrasjon

I et system som skal simulere fysiske egenskaper som posisjon, fart og akselerasjon kreves det numerisk integrasjon av bevegelseslikningene for at simuleringen skal gå glatt og nøyaktig. Den enkleste og mest effektive teknikken er *Forward Euler*-integrasjon som vist i (2.3), vanligvis omtalt som Euler-metoden.

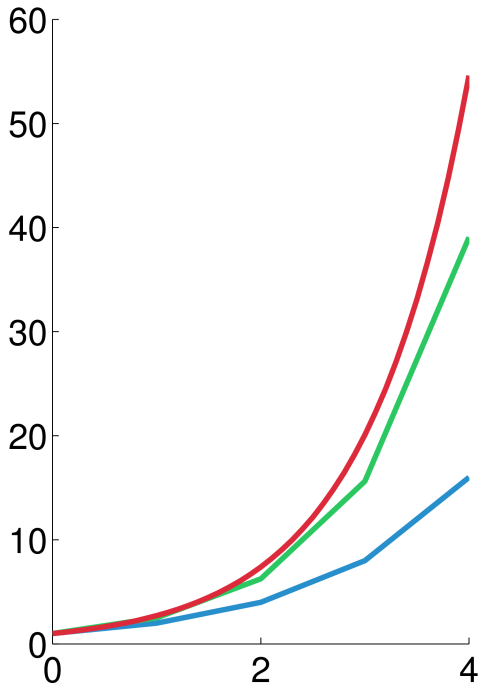
$$\begin{aligned} V_{n+1} &= V_n + hA_n \\ P_{n+1} &= P_n + hV_n \end{aligned} \tag{2.3}$$

Her er P posisjonen, V er farten, A er akselerasjon og h er steglengden, dvs. tiden mellom n og $n+1$. Euler-metoden vil regne ut nøyaktig fart ved konstant akselerasjon og nøyaktig posisjon ved konstant fart, men blir meget unøyaktig når akselerasjonen eller farten varierer eller hvis steglengden h blir stor [19], som vist i figur 2.8.

Improved Euler, også kjent som midtpunktsmetoden er en annen numerisk integrasjonsmetode som benytter snittet av gradienten mellom n og $n+1$ istedet for gradienten ved n til å regne ut neste steg og har derfor bedre nøyaktighet. Likningene for midtpunktsmetoden er gitt i (2.4).

$$\begin{aligned} V_{n+1} &= V_n + hA_n \\ P_{n+1} &= P_n + h\frac{1}{2}(V_n + V_{n+1}) \\ &= P_n + h\frac{1}{2}(V_n + V_n + hA_n) \\ &= P_n + h(V_n + \frac{h}{2}A_n) \end{aligned} \tag{2.4}$$

Avviket med denne metoden er mye lavere enn ved forover Euler-integrasjon som vist i figur 2.8.



Figur 2.8: Numerisk integrasjon av likningen: $y' = y, y(0) = 1$. Blå linje: Forward Euler, grønn linje: midtpunktmetoden, rød linje: eksakt løsning [41]

Forholdet i ytelse mellom disse to integrasjonsteknikkene er konstant. Det vil si at valg mellom disse metodene kun utgjør et konstantledd i kjøretiden, gitt at størrelsen på partikkelsystemet (antall partikler) er lik.

Både Euler-metoden og midtpunktsmetoden er eksplisitte Runge-Kutta-metoder, henholdsvis av 1. og 2. orden. Det er selvsagt mulig å benytte metoder av høyere orden for å oppnå bedre nøyaktighet, f.eks. RK4-metoden. Høyere ordens metoder er imidlertid tyngre å regne ut og om økningen i nøyaktighet kan forsvares til fordel for raskere prosessering (og dermed kortere steglengde), må vurderes i hvert enkelt tilfelle.

2.9 Romlige datastrukturer

En viktig del av simuleringen av interacting particle systems er å identifisere og finne avstanden til nærliggende partikler uten å regne ut avstanden til *alle* partiklene i systemet. En romlig datastruktur vil organisere romlige

data, og benyttes som regel til å optimalisere *romlige oppslag*. Noen vanlige spørringer til en slik database kan være:

- Avstand mellom to objekter
- Nærmeste objekt
- Nærliggende objekter
- Krysser to objekter hverandre?
- Lengden av et objekt
- Areal av objekt
- Sentroide av objekter

Det eksisterer mange teknikker og strukturer for å organisere romlige data, alle med hver sine fordeler og ulemper avhengig av bruksområdene til databasen.

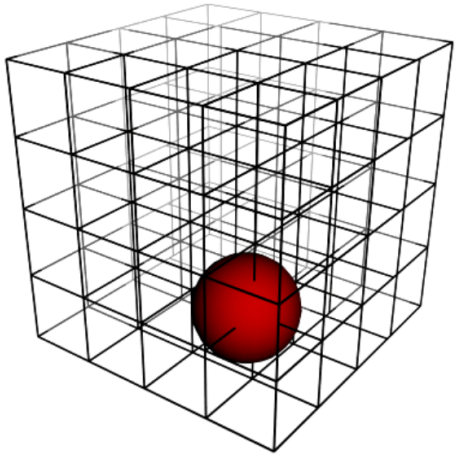
2.9.1 Grid

En datastruktur av typen grid deler rommet opp i tilstøtende celler som tildeles en indeks og hvor man kan slå opp innholdet i cellen basert på dens respektive indeks. Cellene i en gridstruktur vil som regel ha en form som egner seg til applikasjonen. Heksagonale (sekskantede) celler for eksempel, egner seg for romlige oppslag på en kuleflate. Den enkleste og kanskje mest opplagte inndelingen er et uniformt rutenett som strekker seg i hver dimensjon.

I et tredimensjonalt miljø resulterer dette i en matrise med celler som fyller rommet. En enkelt volumcelle kalles voxel (volumetric pixel). Hver voxel har oversikt over tilstøtende celler. Dette muliggjør et svært effektive oppslag på nærliggende objekter ettersom man kun trenger å teste på innholdet i de nærliggende voxlene.

Grid-strukturen fungerer best når objekter er uniformt fordelt i rommet. Hvis objektene har en meget ujevn fordeling vil det være en utfording og justere størrelsen på rutenettet. Hvis rutenettet er for grovt vil hver celle inneholde for mange objekter og ved å minke størrelsen på rutenettet vil det gå med mange celler til å beskrive ingenting.

Reynolds benytter denne typen struktur i [30] hvor den omtales som *Bin-Lattice spatial subdivision*.



Figur 2.9: Illustrasjon av romlig database med grid-struktur [30]

2.9.2 Quad-trær og oct-trær

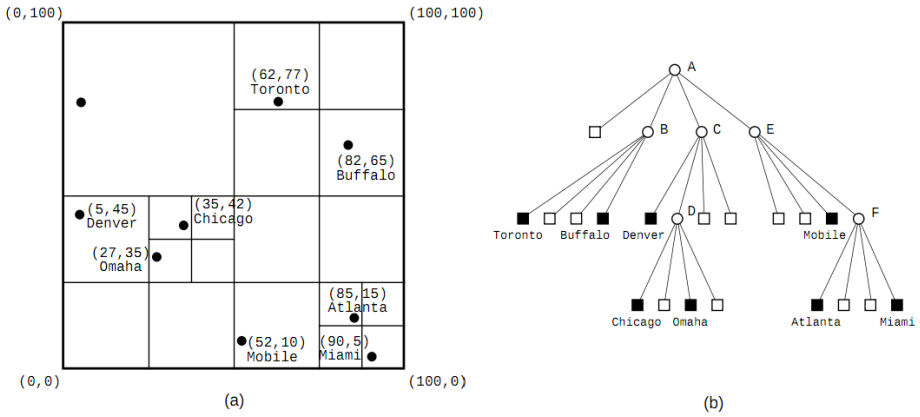
Et binærtre er en tredatastruktur hvor hver node kan ha nøyaktig to barn hvis den ikke er en løvnode. Hvis man utvider denne trestrukturen slik at hver node kan ha nøyaktig fire barn hvis den ikke er en løvnode, kalles dette et quad-tre. Et oct-tre er en tredatastruktur hvor hver node kan ha nøyaktig åtte barn hvis den ikke er en løvnode.

Disse trestrukturene er spesielt godt egnet til å representere romlige data i miljø med forskjellig grad av dimensjon. 1D: binærtre, 2D: quad-tre og 3D: oct-tre. I høyere dimensjoner er det foretrukket å benytte et kd-tre [33] som er beskrevet nærmere i avsnitt 2.9.3.

I et quad-tre vil hver celle (node) i treet representere et kvadrat i rommet. I et oct-tre vil hver celle representere en kube i rommet.

Hvis et quad- eller oct-tre skal benyttes til å representere punktdata vil treet deles opp i flere noder avhengig av antall objekter som ligger i hver celle. I 2.10 vises et eksempel på et quad-tre hvor hver celle har maksimalt ett punkt.

Ulempen med et slikt quad-tre er at antall spaltinger/delinger avhenger av minimumsavstanden mellom to objekter. Hvis to objekter ligger veldig nære hverandre kan dette resultere i veldig dyp spalting av treet. Dette kan løses ved å gi hver av cellene kapasitet c og bare spalte ned treet hvis en celle inneholder flere enn c objekter [33].



Figur 2.10: Et quad-tre som representerer todimensjonale punktdata [33]

2.9.3 kd-trær

Et kd-tre (k-dimensjonalt tre) er et binærtre som deler rommet hvor der det behøves [19]. Hvis en løvnode har for mange objekter vil treet dele denne cellen i to ved senter av objektene i cellen. For hver gren i treet vil deleaksen rotere (første deling langs x-aksen, andre deling langs y-aksen osv). Fordi treet deles i senter av objektene i en celle har det ingen betydning om objektene er uniformt fordelt i rommet eller ikke.

2.10 Flokksimulering

2.10.1 Boids

I 1987 [28] introduserte Craig Reynolds en modell over oppførselen til et individ i en flokk. Modellen av et slikt individ ble kalt boid, en forkortelse for "bird-oid", en blanding av bird og droid. Ved å la flere boider bevege seg i samme miljø vil man kunne observere flokkoppførsel som man kjenner igjen fra virkeligheten.

Hver boid beveger seg ut i fra et sett med svært enkle regler: Collision Avoidance, Velocity Matching og Flock Centering [28]. Disse har Reynolds i senere tid valgt å kalle henholdsvis Separation, Alignment og Cohesion [29].

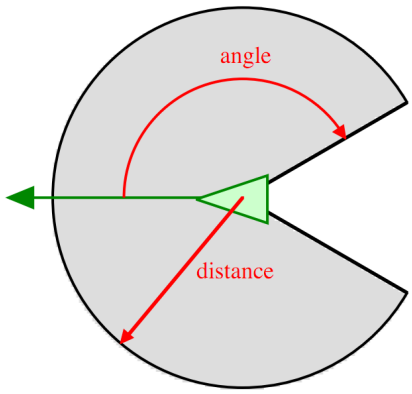
- *Separation*: Unngå kollisjoner med nærliggende boider
- *Alignment*: Følg fartsvektoren til nærliggende boider

- *Cohesion*: Hold seg i nærheten av nærliggende boider

Separation- og Alignment-oppførsel er i utgangspunktet kun supplerende krefter. Sammen sikrer de at medlemmer av flokken kan fly i overfylte omgivelser uten å kollidere med hverandre.

2.10.2 En boids nærområde

Modellen til Reynolds prøver å gi hver boid tilgang til ca. den samme informasjonen et virkelig dyr har mens det utøver flokkoppførsel. En ekte fugls informasjon om omverdenen er gjerne begrenset fordi den ikke kan se uendelig langt, og fugler i nærheten vil dekke til andre fugler lenger unna. Reynolds skriver også i [28] at oppførselen som vi kjenner igjen som flokkoppførsel **avhenger** av et begrenset lokalt syn på omgivelsene. Hver enkelt boid har kun mulighet til å se andre boider som befinner seg i dens *nærområde*. Dette nærområdet er spesifisert av en synslengde og en synsvinkel, som vist i figur 2.11.



Figur 2.11: Grafisk fremstilling av en boids nærområde [29].

2.10.3 Autonome agenter og steering

Begrepet *steering* henviser til realistisk navigering av autonome *agenter* [8]. Pattie Maes [22] definerer autonome agent som: "Autonome agenter er beregningsorienterte systemer som lever i et komplekst dynamisk miljø, og som oppfatter og handler i dette miljøet, og ved å gjøre dette realiserer et antall oppgaver eller mål som de er konstruert for å gjøre." Som regel vil slike agenter simuleres vet at de utfører en *steering behaviour* som igjen bestemmer oppførselen til agenten. Steering behaviours bestemmer ikke oppførselen

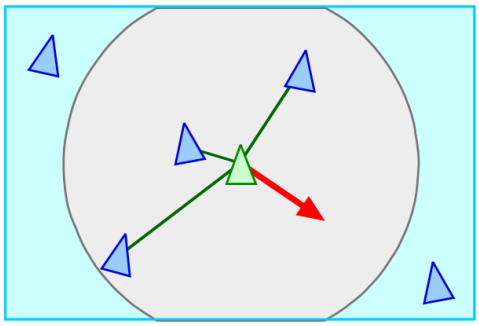
til en gruppe agenter, men oppførselen til én enkelt agent i gruppen basert på dens nærmiljø. Gruppeoppførselen er et emergent fenomen og resulterer i "intelligent" gruppeoppførsel. I en boidsimulering blir én boid simulert av én agent. Når simuleringen er i gang vil agenten kontinuerlig forsøke å bevege seg i en gitt retning. Denne retningen er representert med en steering-vektor. En slik vektor har en retning som gir hvilken vei agenten ønsker å bevege seg, og lengde som gir ønsket størrelse på akselerasjonen.

2.10.4 Separation

Separation er en steering behaviour som gir en agent mulighet til å styre vekk fra en overhengende kollisjon. Oppførselen benyttes også til å hindre at flere agenter klumper seg tett sammen. Størrelsen på kraften er basert på den relative avstanden mellom agentene.

For å regne ut steering-vektor for separation må en agent først finne andre agenter som befinner seg i sitt nærområde. En frastøtende kraft blir regnet ut på bakgrunn av avstandsvektoren mellom den simulerte agenten og agentene i nærområdet.

Utregningen av separation-vektor er som følger: Først finnes avstandsvektoren mellom den simulerte agenten og en nabo. Vektoren blir normalisert slik at den har lengde 1. Agenter som er langt unna skal virke med lavere kraft så den normaliserte avstandsvektoren blir delt på lengden til den opprinnelige avstandsvektoren. Nå vil kraften være mindre for agenter som befinner seg lengre unna. Bidrag fra alle agenter i nærområdet blir lagt sammen og returnert som en steer-vektor [29]. Se forøvrig figur 2.12 og pseudokode 2.1.



Figur 2.12: Grafisk fremstilling av *separation*-oppførsel. Den røde vektoren er beregnet steer-vektor [29]

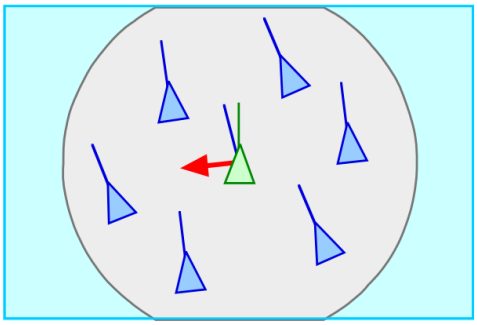
Pseudokode 2.1 Pseudokode for separation-oppførsel

```
Vector Separation() {
    Agent neighborhood[] = neighbor_search()
    Vector steering = 0

    for each (Agent a in neighborhood) {
        // find the offset vector
        Vector offset = me.position - a.position

        // divided to get 1/r falloff
        steering += offset.normalize() / offset.length()
    }

    return steering
}
```

2.10.5 Alignment

Alignment er en steering behaviour som gir en agent egenskapen til å stille seg i samme retning (og/eller samme fartsvektor) som agenter i nærområdet. Denne oppførselen gjør at en agent vil holde avstanden til nærliggende agenter mer eller mindre konstant. Mens separation vil etablere en minimumsavstand mellom agenter vil alignment forsøke å opprettholde den [28].

For å finne steer-vektor for alignment-oppførsel regner man ut gjennomsnittet av fartsvektoren (eventuelt enhets heading-vektor) til alle agenter i nærområdet. Dette er *ønsket fartsvektor*. Den endelige steering-vektoren blir da forskjellen mellom nåværende og ønsket fartsvektor (eventuelt enhets heading-vektor). Denne vektoren vil vende den simulerte agenten slik at den er på linje med agentene i nærområdet. Se forøvrig figur 2.13 og pseudokode 2.2.



Figur 2.13: Grafisk fremstilling av *alignment*-oppførsel. Den røde vektoren er beregnet steer-vektor [29]

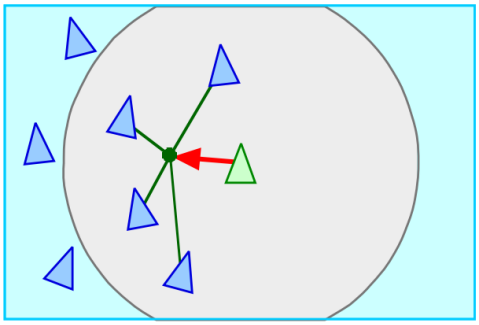
Pseudokode 2.2 Pseudokode for Pseudokode for alignment-oppførsel

```
Vector Alignment() {
    Agent neighborhood[] = neighbor_search()
    Vector steering = 0

    for each (Agent a in neighborhood) {
        // accumulate sum of neighbor's velocity
        steering += a.velocity
    }

    if(neighborhood.count > 0) {
        // steering is the difference between the average
        // velocity and our current velocity
        steering = (steering / neighborhood.count) - me.velocity
    }

    return steering
}
```

2.10.6 Cohesion

Cohesion gjør at en agent ønsker å være nær senter av flokken. Men fordi hver agent har et avgrenset syn på omgivelsene så vil senter av flokken bety senter av agentene i nærområdet. Hvis en agent befinner seg midt inne i en overfylt flokk så vil tettheten av agenter i nærområdet være relativt homogen. I dette tilfellet vil senter av flokken ligge ca midt i den simulerte agentens nærområde og "sentreringstrangen" vil være liten.

En agent finner steer-vektor for cohesion ved å regne ut gjennomsnittet av posisjonene til agenter i nærområdet for så og trekke fra sin egen posisjon. Steer-vektoren er illustrert i figur 2.14 og utregningen beskrevet i pseudokode 2.3.



Figur 2.14: Grafisk fremstilling av *cohesion*-oppførsel. Den røde vektoren er beregnet steer-vektor [29]

Pseudokode 2.3 Pseudokode for cohesion-oppførsel

```
Vector Cohesion() {
    Agent neighborhood[] = neighbor_search()
    Vector steering = 0

    for each (Agent a in neighborhood)
        // accumulate sum of neighbors positions
        steering += a.position
    }

    if(neighborhood.count > 0) {
        // steering is towards the average position of neighbors
        steering = (steering / neighborhood.count) - me.position
    }

    return steering
}
```

2.10.7 Flocking

Ved å kombinere separation-, alignment-, og cohesion-behaviours vil man få det som Reynolds kaller en Boid-modell av flokk- og stimoppførsel [28]. I

noen tilfeller er det tilstrekkelig å kun summere de tre kreftene for å produsere én enkel steer-vektor. For bedre kontroll er det derimot praktisk å normalisere de tre komponentene, for så og skalere dem med hver sin faktor før de summeres, som vist i pseudokode 2.4.

Som et resultat av dette er Boid-flocking-oppførsel bestemt av ni skalarer: En skaleringsfaktor, en avstand og en vinkel (for å definere nærområdet) for hver av de tre oppførselene.

I [28] argumenterte Reynolds for at oppførselene burde prioriteres. Separation hadde høyest prioritet, alignment hadde middels prioritet og cohesion hadde lavest prioritet. Reynolds kalte dette for *prioritized acceleration*. Tanken bak dette var at en agent har kun en gitt mengde akselerasjon tilgjengelig, så i krevende situasjoner ble akselerasjonen brukt til de mest de mest kritiske behaviours først. Hvis akselerasjonen ble "brukt opp" ville de mindre kritiske behaviours bli midlertidig oversett. Måten dette ble gjort på var å legge de tre steering-vektorene i prioritert rekkefølge til i en akkumulator. Lengden på de tre vektorene ble målt og lagt til i en ny akkumulator. Dette fortsatte helt til all tilgjengelig akselerasjon var brukt opp. Den siste vektor som er lagt i akkumulatoren ble trimmet ned for å kompensere for overskuddet av summert akselerasjon.

Reynolds har i senere tid [29] skrevet at en enkel linær kombinasjon av de tre oppførslene har vist segå være tilstrekkelig for å gi realistisk flokkoppførsel.

Pseudokode 2.4 Pseudokode for flocking-oppførsel

```
Vector Flock() {
    Vector sepSteer = Separation().normalized() * sepWeight
    Vector aliSteer = Alignment().normalized() * aliWeight
    Vector cohSteer = Cohesion().normalized() * cohWeight
    Vector flockSteer = sepSteer + aliSteer + cohSteer
    return flockSteer
}
```

2.11 Programmeringsmiljø for kreativ programmering

Et typisk programeringsmiljø/rammeverk er gjerne utviklet med tanke på at det skal være effektivt og allsidig,fte kommer brukervennligheten i andre rekke. Jo mer et programmeringsmiljø er spesialisert for en oppgave, dessto

lettere er det som regel å benytte. Ulempen med spesialiserte programmeringsmiljø er at det kan være unødvendig komplisert å gjøre oppgaver som strekker seg litt utenfor det programmeringsmiljøet var utviklet for.

En gren av slike programmeringsmiljø er de som er utviklet for "Creative Coding". Her er filosofien at man skal bruke minst mulig tid på programmering, og mer tid på den kreative fasen. På bakgrunn av dette er det blitt undersøkt noen programmeringsmiljø som kan forenkle designprosessen.

2.11.1 Processing

Processing er et programmeringsspråk og integrert utvklingsmiljø (IDE) som er laget for utvikling av digital kunst og bruk til læring av programmering i visuell kontekst. Prosjektet ble startet i 2001 av Casey Reas og Benjamin Fry. Et av hovedmålene for Processing er at det skal være et verktøy for å få ikke-programmerere i gang med programmering gjennom å gi umiddelbar visuell respons til kompilert kode. Språket er bygd på Java, men benytter en forenklet syntaks. Processing er gratis og open-source. [40], [46].

2.11.2 OpenFrameworks

OpenFrameworks er et bibliotek til C++, og i er følge nettsiden: "laget for å fremme den kreative prosessen til artister gjennom å tilby et enkelt og intuitivt rammeverk som gjør det lettere å eksperimentere".

Biblioteket er utviklet av Zach Lieberman, Theodore Watson og Arturo Castro med hjelp fra brukere av OpenFrameworks' nettsamfunn. Kode skrevet i OpenFrameworks kjører på Windows, Mac OS X, Linux, iOS og Android. Biblioteket syr sammen flere biblioteker og tilbyr et ryddig grensesnitt: OpenGL for grafikk, rtAudio for lyd, freeType til fonter, freeImage for bildebehandlig og QuickTime for avspilling og opptak av video.

Utviklingen av OpenFrameworks er sterkt påvirket av Processing. API'et er laget for å være minimalistisk og lett å forstå. Det er få klasser, og hver av klassene inneholder få funksjoner. Man kan likevel få stor funksjonalitet ved å legge til *addons*. Nye addons utvikles kontinuerlig av nettsamfunnet til OpenFrameworks og av utviklerene til OpenFrameworks selv. Eksempler på slike addons er bibliotek for matriser, vektoralgebra, OpenCL, OpenCV, vektorgrafikk, GUI, osv. [39], [45].

2.11.3 Cinder

Cinder er nok et gratis open-source C++-bibliotek for kreativ programmering. Det tilbyr en kraftig og intuitiv verktøykasse for programmering av

blant annet grafikk, lyd, video og bildebehandling. Cinder er kryssplatformsbibliotek, og identisk kode fungerer under Windows, Mac OS X og stadig flere andre plattformer, inkludert iPhone og iPad. Biblioteket er utviklet med tanke på å dra nytte av hardware-akselerasjon der dette er tilgjengelig. Cinder-prosjektet ble startet av "The Barbarian Group" som et internt prosjekt laget for å produsere interaktiv reklame, men ble senere endret til et open-source-prosjekt [38].

2.11.4 Field

Field er et open-source software-prosjekt som opprinnelig ble startet av "OpenEnded Group" som et verktøy for å lage digital kunst. Det er et utviklingsmiljø hvor det er lagt fokus på rask og eksperimentell produksjon av kode for å kunne sette sammen og utforske algoritmiske system. Prosjektet ble startet ved MIT Media Lab, og ble utviklet internt i rundt seks år før det ble frigitt som open-source. For øyeblikket er prosjektet kun støttet av Mac OS X.

Field prøver å lenke så mange biblioteker, programmeringsspråk og måter å gjøre ting på som mulig. Man trenger ikke å velge mellom dataflytsystemer [4], grafiske brukergrensesnitt eller ren tekstbasert programmering. Man kan benytte alle tre metodene sammen og hente de beste egenskapene ut fra hver av de. Field benytter Python som programmeringsspråk, men har full støtte for prosjekter laget i Processing som da er skrevet i Java. At man kan benytte forskjellige programmeringsspråk blir sett på som en frihet til programmereren som skal ha fokus på det kreative og ikke det tekniske. Field benytter også såkalt "live coding". Software kan skrives i sanntid uten behov for kompilering. Dette gjør at man umiddelbart vil se endringer i kreasjonen etterhvert som man forandrer kildekoden [24].

2.11.5 NodeBox

NodeBox er et open-source prosjekt som ble startet i 2002. Programmet lar brukeren lage statisk, animert eller interaktiv 2D-grafikk ved å programmere i Python. Den nyeste versjonen, NodeBox 2.0 kjører på kjører på Mac OS X og Windows, med eksperimentell støtte for enkelte Linux-distribusjoner. Utviklerene av NodeBox har hentet inspirasjon fra teknologier som OpenGL og PostScrpt. Prosjektet er basert på DrawBot, et liknende open-source prosjekt som er utviklet for pedagogiske formål [14], [43].

[4] som f.eks LabView

2.11.6 Impromptu

Impromptu er et programmeringsspråk og utviklingsmiljø for Mac OS X. Det er tiltenkt musikere, VJer[5] og grafikere med interesse for live- eller interaktiv programmering. Impromptu er et programmeringsmiljø rundt språket *Scheme* som er en dialekt av Lisp.

I likhet med Field muligjør Impromptu såkalt live-coding. Man kan endre hvilken som helst del av kildekoden mens programmet kjører, dette betyr at man kan definere og endre variabler eller funksjoner live og endringene trer i kraft umiddelbart. Det er mulig å skrive kode som blir schedulet til fremtiden, dette kan for eksempel være noter eller grafiske effekter. Funksjoner som skal loopes implementeres ved å benytte *temporal recursion*, dvs funksjonen scheduler et fremtidig kall til seg selv som siste handling før den avslutter. Bibliotekne er tett integret med Mac OS X. Man har mulighet for å kalle Objective-C-kode fra editoren. Impromptu er også fullt kompatibel med OpenGL.

Kompilatoren benytter LLVM[6] for backend kompilering til x86. Impromptu er ikke open-source, men gratis å bruke [36], [42].

OpenSteer

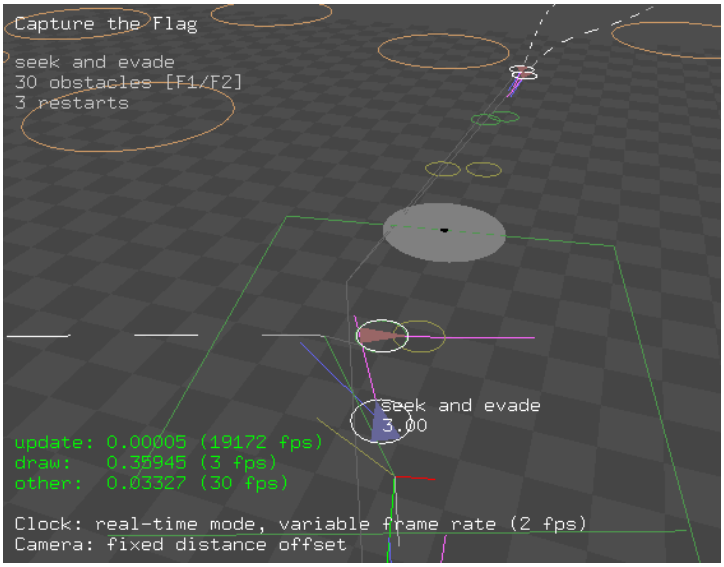
OpenSteer er et open-source bibliotek som er laget med tanke på å assistere utviklingen av *steering behaviours* for autonome agenter i spill og andre simuleringer. Craig Reynolds har selv vært med på å utvikle dette biblioteket. OpenSteer ble opprinnelig laget til Linux, men ble senere portet til Windows og Mac OS X. Det er skrevet i C++ og benytter OpenGL-bibliotekene [32].

OpenSteer tilbyr et interaktivt program, kalt OpenSteerDemo som demonstrerer noen steering behaviours i et tredimensjonalt miljø. Programmet er bygd på en Plug-In-arkitektur, dvs. et tomt rammeverk hvor flere Plug-Ins kan ble utviklet og lagt til etter tur. Programmet kommer med et knippe intressante Plug-Ins ferdig utviklet og lagt til, deriblant:

- Capture the Flag
- MapDrive
- Pedestrians
- Boids
- Soccer

[5] samlebetegnelse for artister som utøver visuell kreativitet i sanntid
[6] Low Level Virtual Machine

Figur 2.15 viser kjøring av "Capture the Flag"-plugin i OpenSteerDemo.



Figur 2.15: "Capture the Flag"-plugin i OpenSteerDemo

Kapittel 3

Utvikling av testbenk til boid-eksperimenter

Dette kapittelet vil gjennomgå oppsett og utvikling av en testbenk for eksperimentering med flocking-egenskaper.

Reynolds har i [29] definert hans algoritmer for de tre flocking-komponentene. Disse ble beskrevet i nærmere detalj i avsnitt 2.10.1. Flocking er et populært tema og i de siste årene har regnekraften i PCer økt såpass mye at man kjøre interaktive flock-simuleringer i sanntid. Da Reynolds ga ut det opprinnelige boid-paperet [28] i 1987 brukte den originale LISP-implementeringen på en Symbolics 3600-maskin 95 sekunder for hvert steg i en simulering med 80 boider. En video på ti sekunder (300 steg) tok rundt åtte timer å produsere [21].

Dagens datamaskiner er betydelig raskere og med riktig optimalisering kan man simulere flere tusen agenter i sanntid på en vanlig kontor-PC.

Siden 1987 har det blitt laget en rekke implementeringer og tolkninger av Reynolds opprinnelige paper fra 1987 [28].

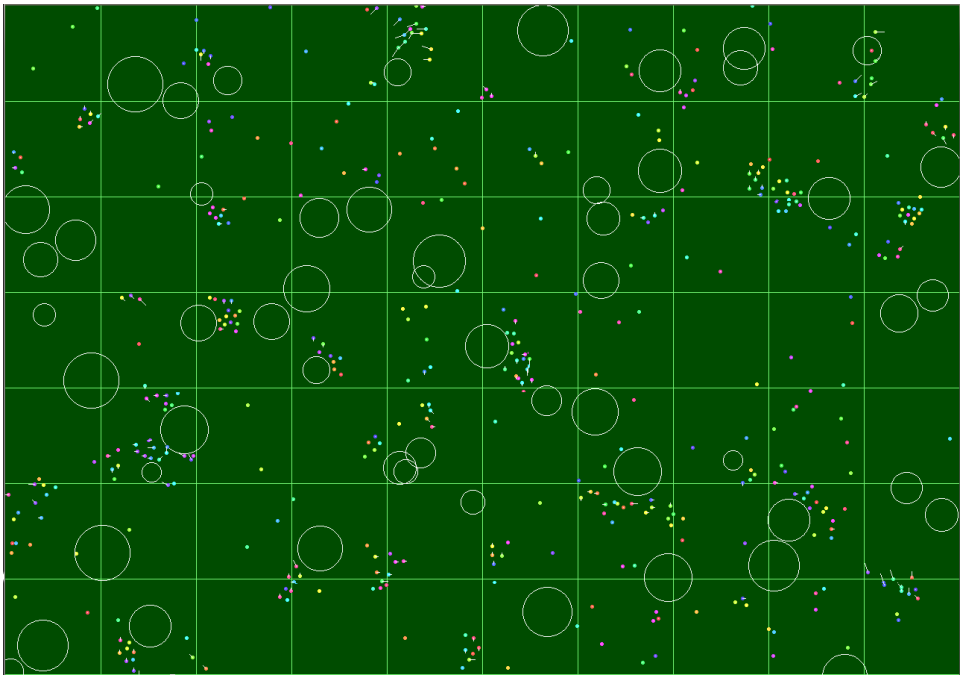
3.1 Noen eksisterende simuleringer

Før man selv begynner å lage et forslag til implementering av flocking-algoritmen vil det være intressant å studere liknende simuleringer som er gjort for å muligens høste andres erfaringer rundt temaet.

3.1.1 Occoids

CoSMoS står får *Complex Systems Modelling and Simulation* og er et rammeverk som er utviklet for å støtte modellering og analyse av komplekse systemer, og som et hjelpemiddel til å konstruere og validere slike systemer.

Et av de første studiene av CoSMoS var Reynolds' boider. Den første implementasjonen var *Occoids*, skrevet i occam-π. Boids var et meget intressant studie fordi simuleringen tar for seg mange viktige egenskaper til komplekse systemer. Man kan se mange intressante egenskaper ved å justere og utvide de tre grunnreglene. Det er enkelt å se om implementasjonen er korrekt fordi kun små feil gir markant endring i oppførsel og helt forskjellige flokkmønster. Mesteparten av kompleksiteten i Occoids ligger i implemente-



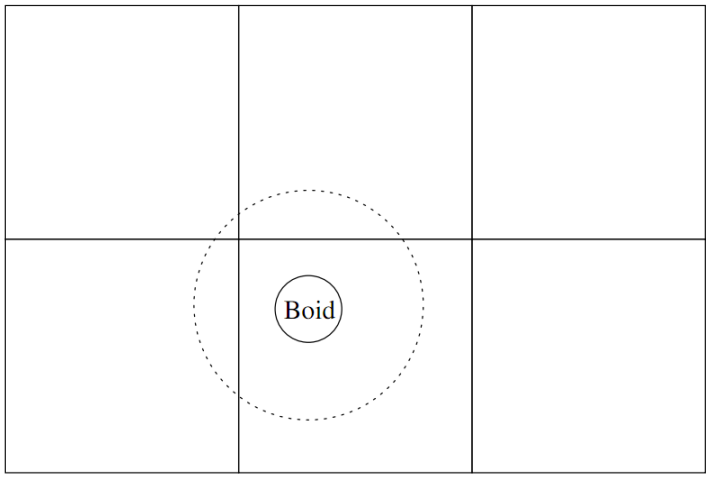
Figur 3.1: Skjermbilde fra Occoids

ringen av *space model*, altså den modellen av rommet hvor boidene beveger seg, og hvordan boidene forholder seg til denne. Modellen er utviklet med tanke på rask, skalerbar og parallell simulering.

Rommet er delt i et antall områder hvor hvert område til styrt av en *Location*-prosess. Hvert område innheholder et antall agent-prosesser (boider og eventuelle hindringer) og har oversikt over posisjonen til disse. Location-

prosessene er satt opp et grid og har delte kanaler som hver av de nærliggende prosessene har tilgang til.

For hvert steg i simuleringen må hver enkelt boid se etter andre agenter i nærområdet. Dette blir gjort ved skaffe oversikt over innholdet i nærliggende Locations. Agentene har et begrenset synsfelt til en sirkel med diameter på maksimalt én location. Dette gjør at hver agent kun trenger å lete i de ni nærmeste locations. Alle agenter som ligger i samme location vil ha behov for å se i de samme location's, derfor er det innført en *viewer*-prosess. Det eksisterer én dedikert viewer per location, og prosessen har som oppgave å lage en oversikt over agenter i nærliggende locations [34].



Figur 3.2: En boids synsfelt i Occoids

For å sikre at alle agentene har et konsistent bilde av verden må det forsikres at posisjonen til alle agentene blir lest ut for hvert steg i simuleringen, og før de oppdaterer sin egen posisjon. På bakgrunn av dette er hvert steg i simuleringen delt opp i to faser:

- Fase 1, viewer-prosessene henter info fra nærliggende locations.
- Fase 2, agentene henter info fra viewer-prosessene, regner ut ny fartsvektor og sender denne til sin location-prosess.

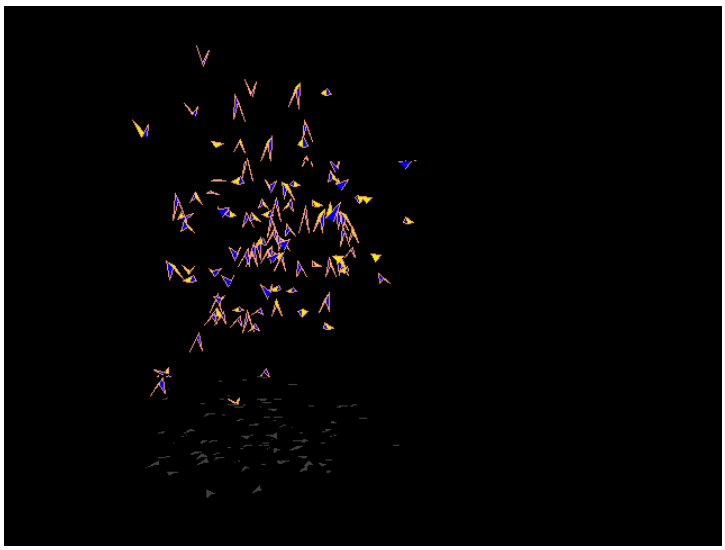
Når en location tar imot en fartsvektor fra en agent vil den oppdatere posisjonen til agenten. Hvis posisjonen til agenten nå ligger utenfor sin location, vil den overføre ansvaret for agenten til den nye location-prosessen.

Implementasjonen ble først utviklet til å kjøre på én PC, men er i senere tid utvidet til å kunne kjøre på et større cluster av maskiner.

3.1.2 xboids

På hans hjemmeside [26] har Conrad Parker beskrevet sitt forslag til pseudokode til en flocking-implementasjon. Denne pseudokoden er svært ofte nevnt og referert til på forskjellige nettsamfunn som diskuterer enkel implementering av en flocking-algoritme.

Pseudokoden er basert på hans egen flock-simulering ved navn xboids. Figur 3.3 viser et skjermbilde fra simuleringen.

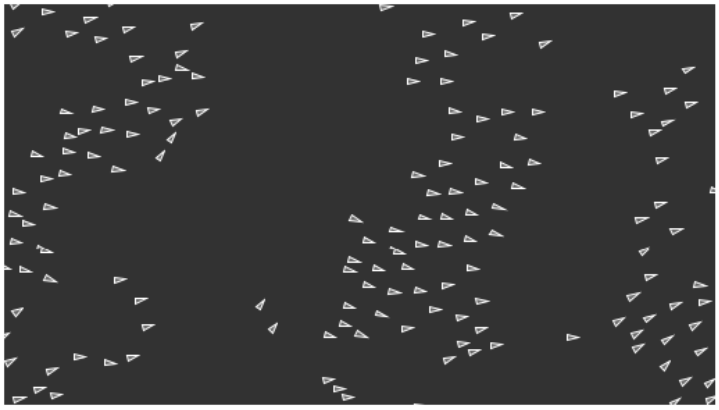


Figur 3.3: Skjermbilde fra xboids

3.1.3 Tutorial Processing

På nettsidene til Processing ligger en implementasjon av flocking-algoritmen til Reynolds som benyttes i en tutorial innenfor emnet "simulering med Processing". Koden er laget av Daniel Shiffman, og er skrevet meget enkelt uten noen form for optimalisering da det er læreprosessen som står i fokus her. Et skjermbilde fra simuleringen er gitt i figur 3.4. Det er mulig å legge til nye boider i simuleringen ved å klikke ved musen. Dette er den ene interaktive delen som er implementert. Implementasjonen "wrapper" boidene rundt skjermen slik at boider som beveger seg ut av skjermbildet kommer inn i bildet igjen på motsatt side.

Strukturen på simuleringsalgoritmen minner om pseudo-koden som tidligere er beskrevet i kapittel 2.10.1, men de tre steering behaviours blir reg-



Figur 3.4: Skjermbilde fra Shiffmans flock-simulering i Processing

net ut på andre måter. Når total steer-vektor er funnet foretas en Euler-integrasjon og posisjonen blir oppdatert som beskrevet i kapittel 2.8. Algoritmen ser ut til å fungere korrekt da boidene utøver plausibel flokkoppførsel.

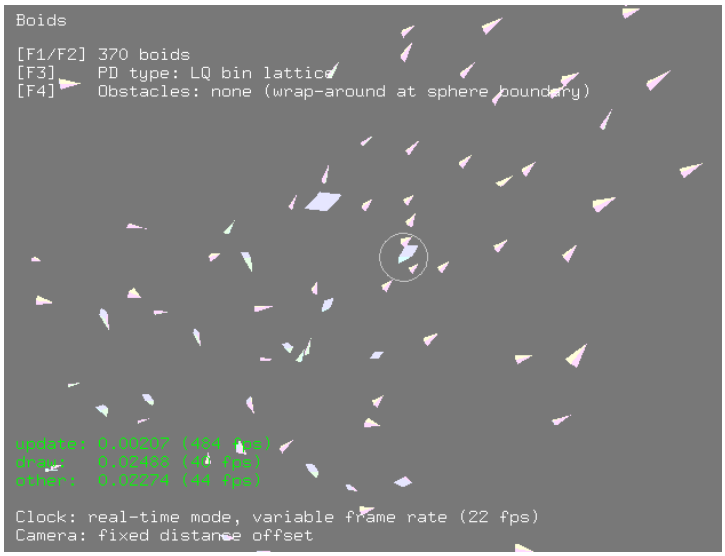
3.1.4 OpenSteerDemo

Craig Reynolds har aktivt vært med å utvikle OpenSteer. Man kan muligens se på OpenSteerDemo som en referanseimplementasjon av flocking-algoritmen til Reynolds. Boid Plug-in'et er satt opp med 200 boids i én flokk, antall boider kan økes og minskes mens simuleringen foregår. Modellen benytter separation-, alignment- og cohesion-reglene. Man kan velge om boider skal foreta lokasjonsoppslag på andre boider gjennom en romlig database eller om de skal traversere gjennom alle boidene i flokken. Flokken beveger seg inni en sfære og man kan velge mellom to forskjellige grensebetingelser:

- "Steer back when outside", når en boid beveger seg utenfor sfæren vil den foreta en *seek*-manøver[1] mot senter til sfæren.
- "Wrap around (teleport)", en boid som beveger seg utenfor sfæren vil umiddelbart bli plassert til motsatt side av sfæren.

Etter steer-vektorene fra hver av reglene er vektet og lagt sammen blir den samlede steer-vektoren kjørt gjennom en rekke filtre. Først justeres vektoren slik at den blir holdt innenfor en maksimal vinkel i forhold til fartsvektoren når boiden holder lav hastighet. Deretter begrenses størrelsen på kraften

[1] en steering behaviour der agenten prøver å bevege seg mot et fast punkt i rommet



Figur 3.5: Boids plug-in i OpenSteer

til boidens maksimalkraft. Av kraften finnes det akselerasjonvektoren Akselerasjonen blir lavpassfiltrert for å jevne raske endringer fra steg til steg i simuleringen. Så tilslutt foretas Euler-integrasjon og fart og posisjon blir regnet ut og oppdatert.

Et bilde fra simuleringen er gitt i figur 3.5.

3.1.5 Tutorial Cinder

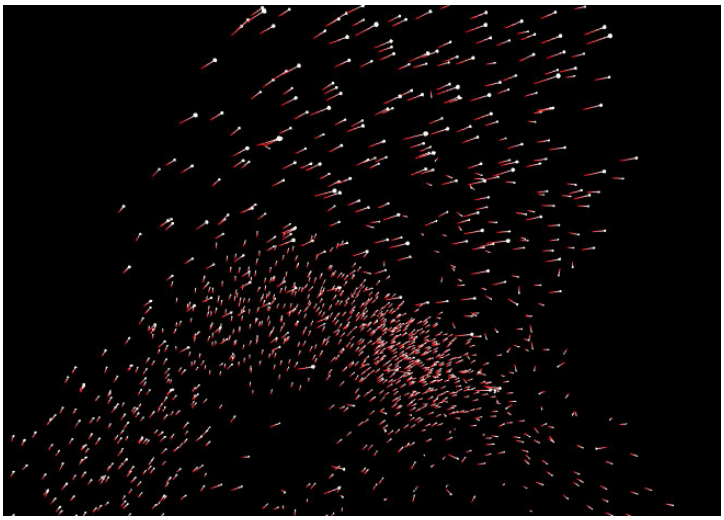
Cinder har i likhet med processing også en tutorial på nettsidene deres som omhandler flocking. Denne tutorialen er skrevet for visualisering i 3D med all ekstra kompleksitet dette innebærer. Her er 20 % av tutorialen dedikert til oppsett av kamera og perspektivinnstillinger.

Figur 3.6 viser et skjermbilde fra resultatet av tutorialen.

3.2 Valg av programmeringsmiljø

Det er ønskelig å se på forskjellige implementasjoner og nøyansene rundt disse. Derfor legges det stor vekt på å konstruere en modulbasert og pluggbar testrigg som gjør det enkelt å teste og veie de forskjelle implementasjonene opp mot hverandre.

Ingen av de nevnte flokksimuleringene var spesielt egnet til å *utforske*



Figur 3.6: Skjermbilde fra flock-tutorial i Cinder

flocking-algoritmen. Om man skal justere på parametere må koden rekompileres for hver gang.

Det er derfor nødvendig å lage en egen testbenk eller utvide en eksisterende simulering slik at det legges til rette for utstrakt testing. I første omgang er ikke ytelsen prioritert. Ved å konstruere implementasjonen fleksibel og pluggbar kan eventuell optimalisering foregå på et senere tidspunkt.

OpenFrameworks og Cinder har identisk designfilosofi, begge er tett integrert med OpenGL og har C++-biblioteker. Den største forskjellen er at Cinder ikke har biblioteker mot Linux.

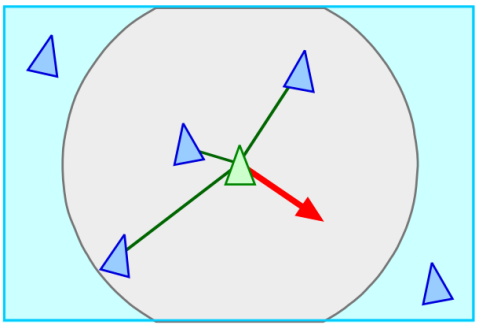
Det er valgt OpenFrameworks som utgangspunkt for testingen. OpenFrameworks benytter seg av plattformuavhengig biblioteker slik at det er mulig å lage en implementering som fungerer i Linux, og senere ha mulighet til smertefritt å kompilere den samme koden til andre plattformer. De mest intressante plattformene er Linux, Windows, Mac OS X og mulignes Ipad. Fordelen med OpenFrameworks kontra Processing er at det er basert på direkte interfacing mot OpenGL-grensesnittet og alle bibliotekene er skrevet i C++. Dette gjør rammeverket vesentlig raskere enn Processing.

3.3 Algoritmiske usikkerheter

Alle de fire ovennevnte flokksimuleringene har ulike måter å regne ut steeringvektorer på. Derfor må også flocking-parameterene justeres ulikt over de

forskjellige simuleringene.

Den delen av flock-algoritmen til Reynolds som er minst intuitiv er utregning av steer-vektor for separation-oppførsel som vist i figur 3.7.



Figur 3.7: Illustrasjon av separation-vektor

Fra [29] har vi at denne er summen av frastøtende avstandsvektorer fra neighbors, normaliset, og deretter vektet med en faktor $1/r$ hvor r er avstand til neigbour. Reynolds skriver at $1/r$ er kun en vekt som har fungert greit, og ikke en fundametal verdi. Dermed er bordet duket for mange tolkninger og usikkerheter rundt separation-delen.

I listing 3.1 vises hvordan Conrad Parkers xboids finner separation. Listing 3.2 viser hvordan OpenSteer og Reynolds regner ut denne vektoren. Cinder-tutorialen finner separation som vist i listing 3.3, og tilslutt viser listing 3.4 hvordan Processing-tutorialen løser dette.

Listing 3.1: Separation-implementering i xboids

```
1  Vector sep(boid bJ) {
2     Vector c = 0;
3     FOR EACH BOID b {
4        IF b != bJ THEN { // not look at ourself
5           // check if neighbor
6           IF |b.position - bJ.position| < 100 THEN {
7              c = c - (b.position - bJ.position)
8           }
9        }
10    }
11    return c
12 }
```

Listing 3.2: Separation-implementering i OpenSteer

```
1     Vec3 steering = 0;;
```

```
    for (otherVehicle over boidList) {
        if (inBoidNeighborhood (otherVehicle) )
        {
            Vec3 offset = otherVehicle.pos() - this.pos();
         // normalize and weight by 1/r in one operation
            steering += offset / -offset.lengthSquared();
        }
    }
    return steering.normalized();
```

Listing 3.3: Separation-implementering i Cinder

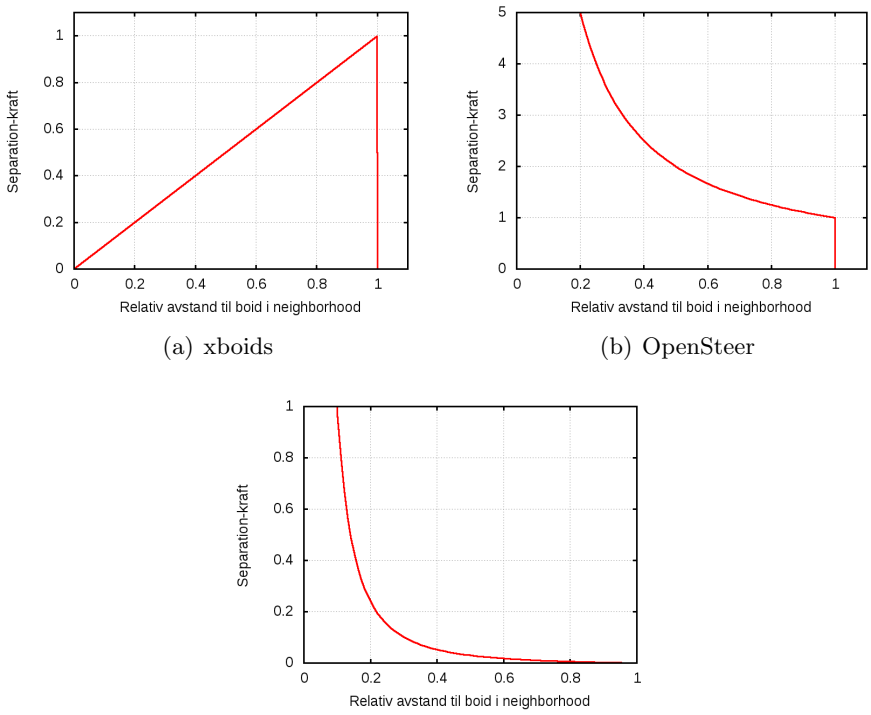
```
for(p1 over mParticles) {
   p2 = p1;
   for( ++p2; p2 != mParticles.end(); ++p2 ) {
      Vec3f dir = p1->mPos - p2->mPos; // offset vector
      float distSqrd = dir.lengthSquared();

      if( distSqrd <= zoneRadiusSqrd ) { // check if neighbor
         float F = ( zoneRadiusSqrd/distSqrd - 1.0f ) * 0.01f;
         dir = dir.normalized() * F;
         p1->mAcc += dir;
         p2->mAcc -= dir;
      }
   }
}
```

Listing 3.4: Separation-implementering i xboids

```
Vector separate (ArrayList boids) {
   float desiredseparation = 20.0;
   Vector steer = 0;
   int count = 0;

   for (int i = 0 ; i < boids.size(); i++) {
      Boid other = boids.get(i);
      // find offset vector
      float d = Vector.dist(pos,other.pos);
      // check if neighbor
      if ((d > 0) && (d < desiredseparation)) {
         Vector diff = Vector.sub(pos,other.pos);
         diff.normalize();
         diff.div(d);        // Weight by 1/r
         steer.add(diff);
         count++;
      }
   }

    if (count > 0) {   // Average
```

```
21        steer.div((float)count);
22      }
23
24      // As long as the vector is greater than 0
25      if (steer.mag() > 0) {
26        // Implement Reynolds: Steering = Desired - Velocity
27        steer.scale(maxspeed);
28        steer.sub(vel);
29        steer.limit(maxforce);
30      }
31      return steer;
32    }
```

Som man ser har alle disse implementasjonene forskjellige måter å regne ut vektingen av separation-vektoren. Processing-utgaven blander også inn boidens fartsvektor inn i bildet. De forskjellige vektingene er illustrert i figur 3.8. På grunn av at vekten i Processing-tutorialen er avhengig maksimalfart, fartsvektor og maksimal steer-kraft, kan ikke denne illusteres like enkelt.



(a) xboids

(b) OpenSteer

(c) Cinder-tutorial

Figur 3.8: Sammenlikning av separation-vekting

Hvis vi først ser på xboids-utgaven, legger en merke til at separation-kraften blir svakere desto nærmere en nabo kommer. Dette virker ulogisk, men Parker argumenterer for dette med å si at hvis to boider kommer veldig nær hverandre vil de ikke frastøtes umiddelbart, men gjennom en jevn akselerasjon [26].

Reynolds vekting i OpenSteer er $1/r$. Dette produserer i utgangspunktet en jevn vekting ut i fra avstanden til hver nabo, men siden hver boid har et simulert synsfelt blir denne kurven bratt avbrutt i randen av synsfeltet. Dette resulterer i en vekting som vist i figur 3.8(b). Til slutt gjøres en normalisering av den samlede steer-vektoren slik at man får bedre kontrol over separation når behaviours skal vektes [29].

I Cinder er det valgt å kompensere for diskontinuiteten i separation-vektoren. Resultatet er en noe mer regnetung algoritme (én ekstra sqrt-operasjon). Her benyttes ingen normalisering så man vil få meget høye separation-vektorer når boider har kommet nær hverandre. Dette kan for eksempel skje på grunn av regnefeil som introduseres av den numeriske integrasjonen av systemet.

Første del av separation-algoritmen i Processing fungerer likt med Reynolds kode i OpenSteer. Deretter blir separation-vektoren vektet ned med antall separation-vektorer lagt til fra neighborhood.

Så følger en snedig og uventet vekting av separation-vektoren. I linje 24 til 31 i listing 3.4 blir steer-vektoren (som har enhet: kraft) skalert til toppfarten, deretter trekkes den nåværende fartsvektoren fra, tilslutt blir steer-vektoren begrenset til maksimal kraft for boiden. Dette gir lite mening og virker svært ulogisk. En mulig forklaring på dette kan være at Shiffmann har feilaktig blandet inn en del fra alignmen-algoritmen til Reynolds, som beskrevet i listing 2.2 på side 29.

3.4 Maskinvare

Forsøkene har i hovedsak foregått på to maskiner. Disse maskinene kan sies å være ytterpunkter på ytelsesskalaen for PC'er som er i bruk i dag.

Maskin 1, selvbygg fra 2011:

- CPU: Intel i7 2600K, 4 cores @ 4 GHz
- RAM: 8 GB DDR3 @ 1600 MHz
- GPU: Nvidia GTX 560 Ti, 1024 MB RAM, 384 CUDA cores

Maskin 2, Dell Precision 380 fra 2006:

- CPU: Intel P4, 1 core @ 3,2 GHz
- RAM: 1 GB DDR2 @ 533 MHz
- GPU: Nvidia Quadro FX 1400

Utviklingen har i hovedsak foregått med Ubuntu 10.10 og 11.04 som operativsystem. Litt ut i testingen ble det klart at grafikkytelsen var markant bedre under Windows. Når det skulle gjøres optimaliseringer og tester på programvarens ytelse ble maskin 1 med Windows 7 Proffesional, 64-bit benyttet.

3.5 Litt om implementasjonen

I testriggen velges det å basere implementasjonen på OpenSteerDemo. Dette er gjort fordi OpenSteer-biblioteket har størst troverdighet da Reynolds selv har vært med på å utvikle det. Hoveddelene i programmet er illustrert i listing 3.5 og 3.6. Innholdet i `flock()`-funksjonen er beskrevet i pseudokode 2.4 på side 31. Det er nødvendig og kjøre oppdateringen av boidene i to runder for å sikre at data om posisjon og fart er konsistent under kjøring av flock-algoritmen.

Listing 3.5: Flock-implementasjon, hovedløkke

```
1  void flockApp::update() {
2      for(int i=0; i<boids.size(); i++)
3          boids[i]->updateBoid();
4
5      for(int i=0; i<boids.size(); i++)
6          boids[i]->updateVehicle();
7  }
8
9  void flockApp::draw() {
10     for(int i=0; i<boids.size(); i++)
11         boids[i]->draw();
12 }
```

Listing 3.6: Flock-implementasjon, boid-kode

```
1  void Boid::updateBoid() {
2      Vector steerForce = flock();
3      acceleration = steerForce / mass;
4  }
5
6  void Boid::updateVehicle() {
7      velocity += acceleration * ofGetLastFrameTime();
```

```
8       position += velocity * ofGetLastFrameTime();
9   }
```

Simuleringen kjører greit med denne minimalistiske implementeringen men har noen svakheter. Det er nødvendig å innføre utvidelser for at simuleringen skal ha noen verdi.

3.5.1 Grensebetingelser

For å holde flokken innenfor et gitt område, for eksempel i skjermbildet, må det introduseres kode som bidrar til å holde hver enkelt boid innenfor gitte grenser. En løsning er å la de sprette tilbake hvis de treffer en av grensene, akkurat som en bilijardkule som treffer en kant. Dette er simpelt, men gir uønskede resultater fordi boidene vil få en urealistisk oppførsel og vil oppleves som masseløse.

En mer elegant løsning er å legge til en regel som "oppfordrer" boidene å holde seg innenfor grensene. En slik løsning er vist i pseudokode 3.5.1. Denne regelen kan vektes og legges til resten av de tre flock-reglene slik at man får en samlet steer-vektor som vil holde boidene innenfor gitte grenser.

Pseudokode 3.1 Forslag til grensebetingelse

```
Vector Boundaries(Boid b) {
    Vector steer = 0

    IF b.position.x < x_min THEN
        steer.x = 1.0
    ELSE IF b.position.x > x_max THEN
        steer.x = -1.0
    END IF

    IF b.position.y < y_min THEN
        steer.y = 1.0
    ELSE IF b.position.y > y_max THEN
        steer.y = -1.0
    END IF

    RETURN steer
}
```

En tredje løsning kan være å la boidene "wrappe" rundt scenen. Det vil si at hvis en boid for eksempel beveger seg ut av skjermbildet mot høyre vil den bli reposisjonert til venstre side av skjermen, hvor den vil fortsette å bevege seg med samme fartsvektor. Denne teknikken benyttes bl.a i OpenSteerDemo.

3.5.2 Hastighetsbegrensing

Det kan være greit å sette en øvre grense på størrelsen til fartsvektoren for hver av boidene. Dette gjør at de ikke beveger seg for raskt. Man regner også med at dyr har en topphastighet de kan bevege seg med, derfor er det rimelig å begrense hastigheten til boidene.

3.5.3 Kraftbegrensing

På samme måte som fartsvektoren blir begrenset kan det være ønskelig å sette en øvre grense på akselerasjonvektoren til hver av boidene. Dette gjør at de ikke har mulighet til å utføre raske bevegelser, akkurat som fugler ikke kan bråstoppe mens de flyr i luften.

3.5.4 Screen tearing

Under simuleringen opplevdes det tendenser til*Screen tearing*. Dette er en visuell forstyrrelse i skjermbildet når to eller flere rammer blir lagt til skjermenbufferet under én enkel skjermopptegning.

Dette skjer når skriving av skjermbufferet ikke er synkronisert med skjermoppdateringsfrekvensen. Forstyrrelsen vises ved at bildet får en horisontal forstyrrelse som vist i figur 3.9. Problemet kan løses ved at man synkroniserer oppdateringen av skjermbufferet med tegningen av skjermbildet, som regel omtalt som V-Sync. Denne synkroniseringen foregår ved at skjermkortet låser skjermbufferet mens det er i gang med å tegne et bilde på skjermen. Når skjermbildet er ferdig tegnet vil skjermkortet kopiere data fra et off-screenbuffer og til det aktive skjermbufferet før det går i gang med å tegne et nytt bilde. Med denne synkroniseringen gjør dette at hastigheten på tegneoppdateringer i en simulering blir begrenset til skjermens oppdateringshastighet. Dette vil i praksis si at man prøver å låse oppdateringshastigheten til 60 rammer i sekundet.

Figur 3.9: Eksempel på Screen tearing (fra wikipedia)

3.5.5 Normalisering av steering-vektorene

Reynolds skriver i [29] at det ofte kan være tilstrekkelig å summere de tre vektede steer-vektorene sammen til en samlet flock-vektor for å oppnå tilstrekkelige resultater. Det kan imdlertid være nyttig å normalisere disse vektorene før de legges sammen slik at man har bedre kontroll på størrelsesforholdet mellom de. Dette gjør at vektingen blir enklere fordi alle behaviours har vekttall som likner hverandre. Ulempen med slik normalisering er at man mister litt av dynamikken som ligger i de forskjellige oppførslene.

3.5.6 Trails

For at bevegelsen til boider skal bli lettere å fange opp implementeres det en *trail* som følger dem. Dette kan betraktes som et spor der de tidligere har bevegd seg. For at skjermen ikke skal fylles med trails er det nødvendig å sette en maksimal lengde på sporet.

Implementasjonen av dette sporet er løst ved å lage en liste over tidligere posisjoner i et ringbuffer. Når posisjonene til alle boids i simuleringen skal legges til et buffer, 60 ganger i sekundet blir det naturligvis mye minneaksessering. I første omgang ble bufferet løst med C++'s STL-biblioteker, men dette viste seg å være en stor flaskehals for ytelsen. Ved å lage en egen minimalistisk ringbuffer-klasse uten index-sjekk eller annen feilhåndtering

steg ytelsen betraktelig.

For å unngå unødvendig store bufre og mye minneaksess er det lagt til en begrensing slik at det posisjonen ikke blir lagret mer enn 20 ganger per sekund. Med denne begrensningen unngår man problemer når simuleringen når ekstreme hastigheter (1000 rammer i sekundet) hvis v-sync er deaktivert og scenen består av få boids.

3.6 Optimalisering av boids-algoritmen

En naiv implementasjon av boids-algoritmen på maskin 1 klarer å simulere opptil 5-600 boider før simuleringshastigheten går under 30 rammer i sekundet. En tanke bak eksperimentet var at hver boid skal representere et punkt i et brannalarmsystem. For at simuleringen skal ha noen verdi må det gjøres optimaliseringer av algoritmen slik at det er mulig å simulere vesentlig flere agenter med akseptabel oppdateringsfrekvens.

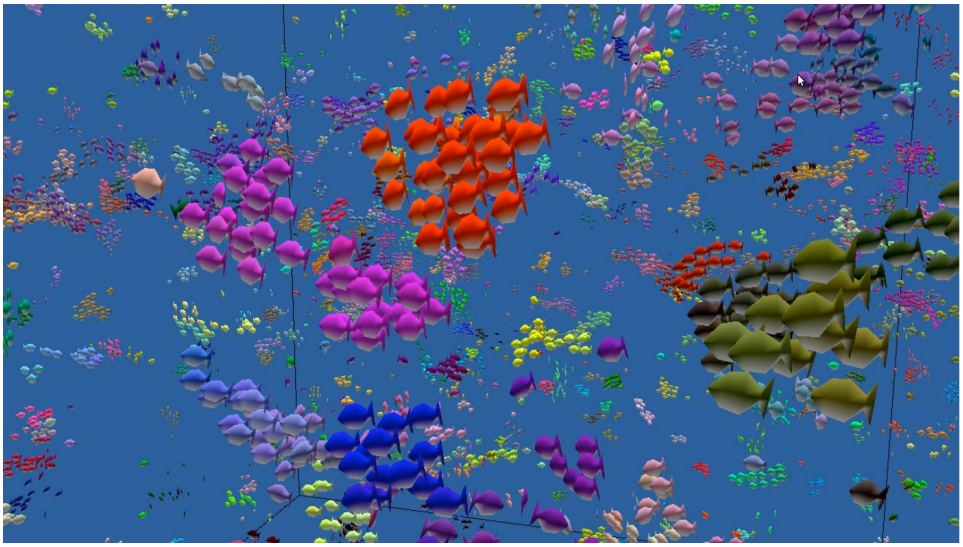
3.6.1 Tidligere arbeid

I 2006 ga Craig Reynolds ut et paper: "Big Fast Crowds on PS3" [31] som beskriver en teknikk for å kjøre store agentbaserte flokksimuleringer ved å benytte paralellprosessering for å oppnå høy ytelse. Det diskuteres en implementasjon for Playstation 3, kalt "PSCrowd". PSCrowd muliggjør simulering av opp til 15 000 agenter ved 60 fps.

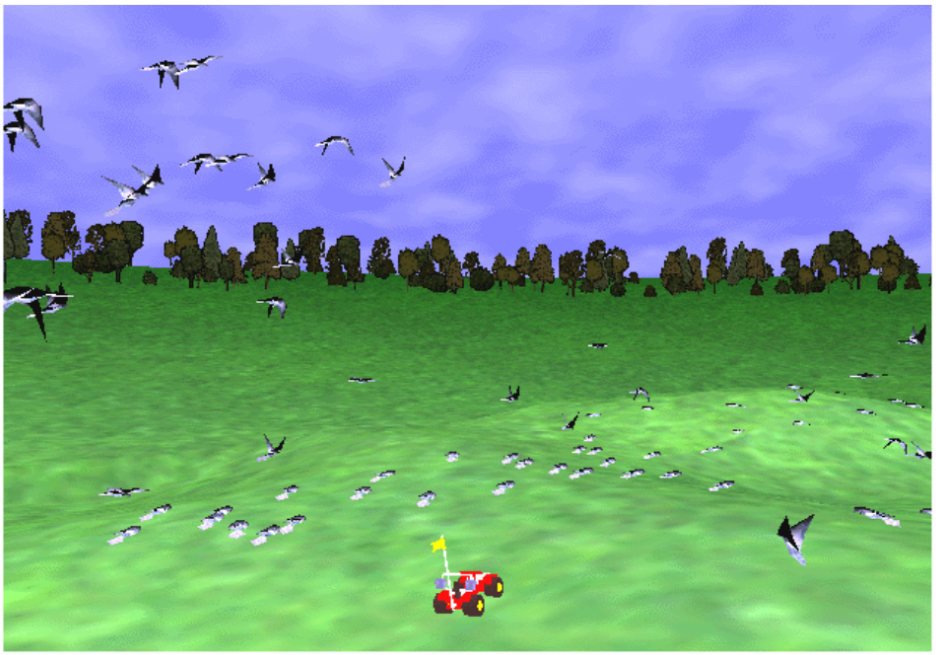
Mange av prinsippene er en videreføring fra et tidligere paper, "Interactions with Groups of Autonomous Characters" [30]. Dette paperet beskriver en simulering "Pigeons in the Park" laget til Playstation 2 og fokuserer ikke like mye på paralellisering. Simuleringen bestod av 280 agenter og ble kjørt ved 60 fps.

3.6.2 Lokasjonsoppslag

Den potensielt mest regnekrevende oppgaven i en slik simulering er å finne lokasjon til andre medlemmer i flokken [30]. En flokk av denne typen kan sees på som et partikkelsystem som har innbyrdes interaksjon (*interacting particle system*). Et slikt system er nødt til å se på *alle* partikler i gruppen, selv om det velger å ignorere de fleste. Som et resultat av dette vil et slikt system ha asymptotisk kompleksitet gitt av $O(N^2)$. Dobler man antall boider i en flokksimulering, vil problemet bli fire ganger så stort. Det er viktig å merke seg at uansett hvor raskt et slikt oppslag på lokasjon til andre med-



Figur 3.10: PSCrowd Chamelon Fish demo av 10 000 i stim som kjører i 60 fps [31]



Figur 3.11: Fra simuleringen *Pigeons in the Park* [30]

lemmer av flokken er, så vil det dominere problemstørrelsen når flokken blir stor.

I en simulering av denne typen foregår alle viktige handlinger og reaksjoner mellom agenter i nærheten av hverandre. $O(n^2)$-delen av algoritmen består kun av å finne naboer til hver enkelt agent. Ved å legge agentenes posisjon i en romlig database kan man gjøre oppslag i databasen på nærliggende boid i konstant tid.

Det er utviklet en rekke forskjellige algoritmer for å aksellerere lokasjonssøk. Noen intressante er f.eks kd-trær, BSP-trær og quad-trær. Reynolds benytter i [30] og [31] en grid-strukter, (*bin-lattice*) til lokasjonssøk.

Størrelsen på rutenettet justeres slik at man kan foreta et noenlunde nøyaktig oppslag, men likevel holder antall forflytninger av agenter mellom bins nede. Reynolds har foreslått et rutenett på 10x10x10 som et greit utgangspunkt for simuleringen i [30].

I [35] foreslås det en formel som vist i (3.2) for antall celler i et slikt grid. Størrelsen på grid'et er gitt i (3.1).

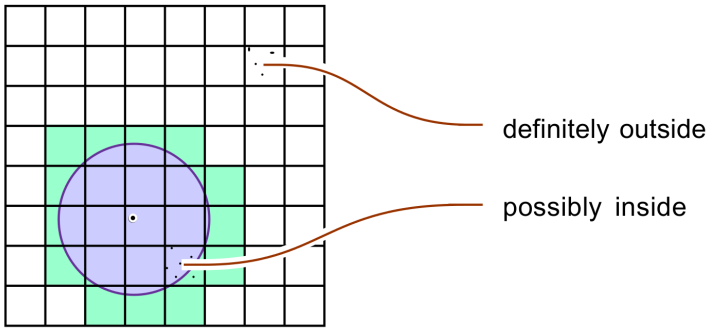
$$\text{GridSize} = \left\lceil \sqrt[3]{\frac{\text{TotalBoids}}{\text{DesiredDensity}}} \right\rceil + 1 \tag{3.1}$$

$$\text{CellCount} = \left\lceil \frac{\text{VisionDistance} \cdot (\text{GridSize} - 1)}{\text{WorldGridDimention}} \right\rceil + 1 \tag{3.2}$$

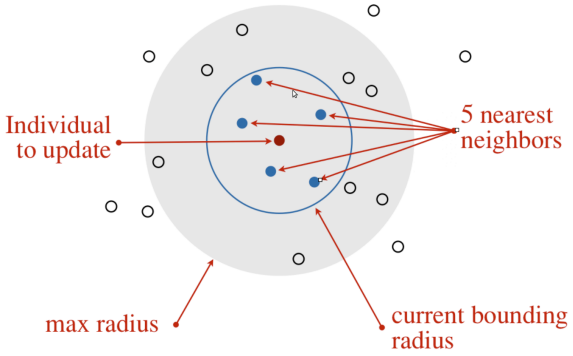
Et oppslag i en romlig database av grid-typen tar inn posisjon og søkeradius, og returnerer en liste over alle bins som befinner seg innenfor radien til den aktuelle posisjonen. Denne listen derefereres til en liste over agenter som igjen returneres til den agenten som foretar oppslaget. Deretter trenger agenten kun å gjøre tester på om agentene i listen befinner seg i dens nærområde. Denne prosessen illustreres i figur 3.12.

Hvis agentene opprettholder en minimumsavstand gjennom separationregelen vil antallet agenter i en bin ha en øvre grense. Hvis søkeradien i tillegg er kortere enn den lengste siden i grid'et (som den som oftest vil være), vil man være garantert et begrenset antall agenter fra et nabosøk. Med dette vil problemet rundt nabosøk bli redusert fra $O(N^2)$ til $O(N)$ i størrelsesorden.

Reynolds har i [31] lagt til nok en algoritme for å begrense tid brukt på nabosøk. I tillegg til bin-lattice oppdeling benyttes K *nearest neighbors*-søk. Denne algoritmen stopper søket etter den har funnet de k nærmeste agentene som vist på figur 3.13. De nære agentene har større innflytelse enn de lenger unna og i følge [12] vil fugler i en fuglestim mest sannsynlig kun bry seg om de 5 - 10 nærmeste naboer i stimen.



Figur 3.12: Lokasjonsoppslag med *bin-lattice* algoritme [30]



Figur 3.13: Oppslag på k nærmeste naboer [31]

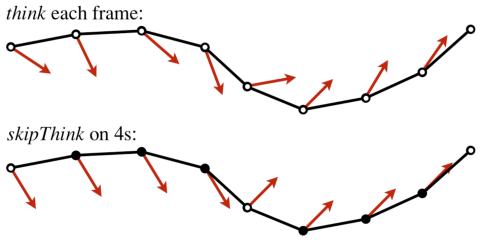
Denne algoritmen tregner ikke krav til søkeradius eller minimumsavstand mellom agenter for å ha lineær kjøretid på sortering og oppslag. Siden sortering kjører på $O(n)$ og oppslag på $O(1)$, er man nå garantert lineær kjøretid [31].

3.6.3 skipThink

For at simuleringen skal vises helt jevnt ønsker man å etterstrebe tegning av minimum 60 bilder i sekundet. Dette gjøres for å unngå screen tear.

I [30] skriver Reynolds at det er tilstrekkelig å la agentene "tenke" ved 10 Hz, en sjettedel av animasjonshastigheten. Mellom hver "tenking" følger agentene sin forrige handling. Det vil si at den samme `steer`-vektoren vil benyttes i seks etterfølgende rammer av simuleringen. Ved å la tidspunktet når hver agent tenker være tilfeldig, unngår man at hele flokken tenker

samtidig. Man ender da opp med at ca. en sjettedel av flokken tenker for hver ramme av simuleringen. Reynolds har i [31] valgt og kalle denne teknikken *skipThink*



Figur 3.14: *skipThink* gir samme steer-vektor til flere rammer av simuleringen [31]

3.6.4 Parallellisering

Mye forskning er viet til å parallellisere flocking-algoritmen til Reynolds.

Allerede i 1993 hadde Helmut Lorek laget en implementasjon av flocking-algoritmen som kjørte på et Meiko Transputer-system med opptil 50 transputere [21]. Fullt bestykket med 50 transputere gjorde denne implementasjonen en simulering av 100 boider med en hastighet på opptil ca 4,5 rammer i sekundet.

I 2004 beskrev Bo Zhou en algoritme som kjørte på et Linux PC-cluster bestående av 27 prosessorer koblet sammen med Myrinet[2][48]. Av 27 tilgjengelige prosessorer ble 16 benyttet i simuleringen. Gjennomsnittshastigheten for hver prosessor var på 450 MHz. Implementasjonen nådde en hastighet på rundt 67 rammer i sekundet for simulering av 500 boider med 16 prosessorer.

Etterhvert begynte man å fokusere mer på å benytte skjermkortet i en datamaskin til å kjøre simuleringen. Skjermkort hadde begynt å få mer regnekraft enn de raskeste prosessorene som var å oppdrive.

I november 2004 beskrev Ugo Erra en flocking-algoritme som kjørte simuleringen på en GPU [10]. Selve nabosøket ble gjort på CPU da GPUene hadde begrenset mulighet til å foreta branching i kode. Implementasjonen nådde en hastighet på 30 rammer i sekundet ved simulering av 4096 boider.

I 2006 ga Reynolds en beskrivelse på hvordan flocking-algoritmen kunne kjøres på Playstation 3-systemet [31]. Systemet består av én standard PowerPC CPU og syv Synergistic Processor Units (SPU) som tilsammen utgjør

[2]Høyhastighets fiberoptisk nettverk brukt til sammenkobling av flere maskiner for å danne et cluster

Cell-prosessoren. Implementasjonen kjørte en simulering av 15 000 boids med en hastighet på 60 rammer i sekundet.

I 2009 beskrev Ugo Erra en oppdatert flock-algoritme som benyttet CUDA til å kjøre på GPU [11]. Implementasjonen kjørte på et GeForce 8800GTS-kort og klarte å simulere 100 000 boider med en hastighet på 60 rammer i sekundet.

I denne oppgaven er det valgt å se nærmere på parallellisering ved å benytte OpenCL til å interface skjermkortet. Mer om dette i neste avsnitt.

3.7 OpenCL

Vi kan erstatte hovedløkken som vist i listing 3.5 ved å flytte data som skal prosesseres over til skjermkortet og la det gjøre jobben parallellt.

Når en slik flokksimulering skal parallelliseres er det viktig å huske på implementere ett synkroniseringspunkt. Etter alle boidene har lest ut naboenes fart og posisjon, og før de oppdaterer sin egen fart og posisjon er det nødvendig med en global barrieresynkronisering for å sikre at alle boidene har et konsist bilde av verden.

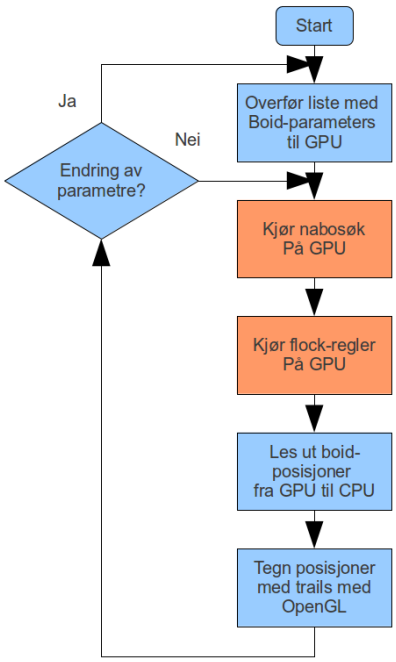
For at simuleringen skal kunne foregå på en GPU må all data ligge klar i global-memory på skjermkortet før kernels begynner å kjøre. Når kode kjøres på en GPU er det ikke mulighet for dynamisk minneallokering. Dette gjør at det blir mer komplisert å konstruere en romlig database på skjermkortet. For å gå rundt dette problemet blir det i første omgang forsøkt å administrere den romlige databasen på CPU'en for så å flytte nabolisten til hver boid over PCI-Express-bussen til skjermkortet og la GPU'en kjøre flock-algoritmen i parallell.

Fordi nabosøket som allerede tar mesteparten av tiden i en simulering foregår sekvensielt på CPU og overføringshastigheten på PCI-Express-bussen er relativt lav[3], vises kun en marginal ytelsesøkning.

På bakgrunn av dette implementeres en "brute-force" nabosøk-algorimte i OpenCL som finner de k-nærmeste naboer innenfor nærområdet til en boid. Denne kjøres som en egen kernel før flock-kernelen blir kjørt. Det er nødvendig med nok en barriæresynkronisering mellom nabosøk-kernelen og flock-kernelen for å sikre at nabosøket er ferdig idet flock-kernelen starter. Den overordnede gangen i programmet er vist i figur 3.15, her er oransje bokser deler av koden som kjøres parallelt.

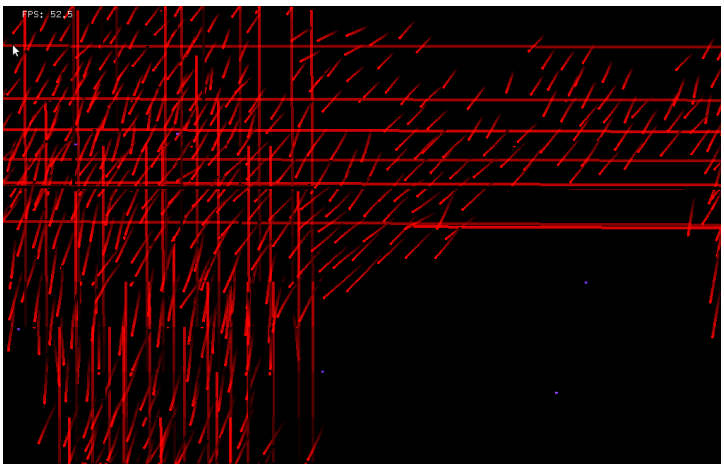
Ideelt sett hadde det ikke vært nødvendig å lese utregnede data tilbake fra skjermkortet kun for at disse igjen skal tegnes med OpenGL til skjerm-

[3] 4 GB/s teoretisk hastighet



Figur 3.15: Overordnet gang i flokksimulering som benytter OpenCL

kortet igjen. Det er mulig å assosiere data som allerede ligger på grafikkortet med data et API-kall til en OpenGL-funksjon skal tegne. Denne teknikken egner seg imidlertid ikke til applikasjonen fordi når trails skal lagres og tegnes må posisjonsdata filtreres for å unngå at det blir tegnet trails på kryss av skjermen når boider "wrapper" rundt skjermbildet. Et eksempel på dette er vist i figur 3.16



Figur 3.16: Feilvisning av trails

3.8 Loggdata

For å undersøke hvor mye aktivitet det er på et typisk brannalarmanlegg som Autronica har produsert blir det hentet ut loggdata fra anlegget som står i Autronicas fasiliteter på Lade. Anlegget består bl.a. av 870 røyk- og varmedetektorer koblet til et antall brannsentraler som alle videresender data til et sentralt presentasjonssystem som loggfører de fleste hendelser som forekommer på de forskjellige detektorsløyfene.

Loggene er lagret i binærfiler som følger denne strukturen:

- 4 byte little endian integer, secondsSinceUnixEpoch
- 3 byte ASCII diskretisert måleverdi
- ASCII linefeed

Ved en ny måling repeteres strukturen.

Det blir laget et program for å parse dataene til filer som enkelt kan analyseres i program som for eksempel MATLAB eller GnuPlot.

3.9 Sanntidsdata

Det er mulig å hente data i sanntid fra en brannalarmanlegg ved å åpne en HTTP-socket mot en brannsentral som står i samme subnett som datamaskinen. Etter å ha kjørt GET-kommandoen returnerer sentralen et JSON-dokument[4] som inneholder data om de n siste hendelsene sentralen har registrert.

På grunn av at det er såpass gode data tilgjengelig fra logger, vil det ikke prioriteres å implementere en nettverksforbindelse mot en sentral for mottak av sanntidsdata.

[4]akronym for JavaScript Object Notation. Er i prinsippet en lettvekts XML-protokoll

Kapittel 4

Eksperimenter og resultater

4.1 OpenCL

Etter OpenCL ble implementert er det mulig å ha langt større flokksimuleringer. Skjermkortet simuleringen kjørerpå inneholder 384 compute units, dette resulterer i at maksimalt 384 work-items kjører i parallell. På grunn av den naive implementasjonen av nabosøk ($O(n^2)$) får man ikke lineær speed-up. I forhold til sekvensiell kjøring på CPU med romlig database oppnås det ca 6 ganger speed-up. Ved nærmere analyse av implementasjonen vises det at nabosøket står for over 99 % av prosessortiden på skjermkortet-. En ordentlig implementasjon av en romlig database hadde vært å foretrekke. AMD presenterer i [23] en teknikk som muliggjør "spatial binning" av 500 000 punkter på GPU'en iløpet av 10 ms, dvs. 500 000 punkter, 100 ganger i sekundet.

Implementeringen kjører nå rundt 6 000 boids ved en hastighet på 60 rammer i sekundet uten andre optimaliseringer enn parallellisering av algoritmen (uten skipThink). Hvis det blir lagt ned mer arbeid i et effektivt lokasjonssøk som har kompleksitet $O(n)$ ville simuleringen gått betydelig raskere.

I vedlegg `500kBoids.MOV` vises en kjøring av simuleringen uten lokasjonssøk og uten trails. Flocking-algoritmen kjører fremdeles på alle boidene, men de kan kun se boid nr. 1 som nabo.

Som man ser er det nå betydelig flere boids i simuleringen, det er mulig å simulere 500 000 boids med en hastighet på over 30 rammer i sekundet.

Hvis man i tillegg skrur av tegning av boidene på skjerm når simuleringen en ytelse på 1 000 000 boids ved 55 fps.

Om man ytterligere skrur av readback av posisjonsdata tilbake til CPU,

yter simuleringen 4 millioner boids ved 300 fps. Ved et høyere antall boids kræsjer skjermdriveren, antagelig fordi skjermkortet går tomt for minne pga. at hver boid krever 14 float variabler.

4.2 Testrigg

Det er implementert to versjoner av testriggen, en hvor prosessereing foregår på CPU og en hvor prosessering foregår på GPU via OpenCL. Den opprinnelige CPU-baserte riggen er mest fleksibel, men også den som er minst effektiv. Dermed støtter den færre boids enn OpenCL-versjonen. OpenCL-versjonen av riggen er den som er mest komplett og hvor mesteparten av testingen har foregått på.

Ved å trykke ¡space¿ åpnes og lukkes innstillingspanelet, 'f' velger fullskjerm-modus, 'r' nullstiller posisjons- og fartsvektorer til alle boider i simuleringen, 'v' skur av og på VSync. Når testriggen er i vindumodus er det mulig å interaktivt endre størrelsen på vinduet.

Side en i panelet inneholder innstillinger som omfatter "vehicle"-modellen selve boids-algoritmen kjører på. Her kan det justeres maksimal hastighet, maksimal kraft, radius, trail-lengde og global farge på alle boidene. Ingen av disse parameterene er direkte relatert til boids-algoritmen.

Neste side inneholder parametere til flocking-algoritmen. Det er mulighet for å justere tre parametere for hver oppførsel. Justering av disse parameterene omfatter alle boider i flokken. Nederst til venstre i skjermbildet vises en illustrasjon av neighborhoods til de forskjellige oppførselene.

Side tre inneholder slidere hvor parameterene bare blir påtrykt 10 % av den totale flokken. Hver parameter er identisk med det som er justert på side to helt fram til man justerer en av parameterene. Da er det kun den ene parameteren som ble endret i 10 % av flokken. I tillegg endrer de aktuelle 10 % av flokken farge til rød slik at de kan enkelt identifisere hvilke boider som beveger seg etter andre parametere.

På grunn av at det er forholdsvis krevende kodemessing og tillate endring av flokkstørrelsen mens simuleringen er i gang er denne finessen utelatt på OpenCL-utgaven av testriggen.

4.3 Resultater fra testriggen

For å finne ut hvordan de forskjellige flokkparameterene påvirker oppførselen til flokken er det nødvendig å foreta noen tester.

Testinger foregår ved å la alle parametere stå fast, og gradvis justere en enkelt parameter til man ser en endring i oppførselen.

Utgangspunktet for testingen er følgende parametersett:

	Factor	Radius	Angle
Separation	5.0	20.0	360
Alignment	2.0	50.0	310
Cohesion	5.0	100.0	200

Max Speed settes til 100.0 og Max Force til 2.0.

Det er viktig å merke seg at under testingene som er foretatt med den aktuelle testriggen vil en enkelt boid kun se på de 10 nærmeste boider som befinner seg innenfor synsfeltet. Det kan derfor være vanskelig å se endringer i oppførsel hvis synsfeltet justeres meget stort.

4.3.1 Separation angle

Eksperimentet er illustrert i `sepAngle`-filen

Flokken beveger seg i utgangspunktet noenlunde likt som en virkelig fugleflokk kunne ha gjort. Ved å justere ned synsfeltet for separation observeres det at flokken etterhvert slutter å bevege seg. Dette skjer når vinkelen går mot 240 grader. Etter hvert som synsfeltet fortsetter å snevres ned mot 90 grader stopper til slutt hele flokken opp. Når synsfeltet justeres under 90 grader og ned mot 0 løses flokken opp igjen og Cohesion-oppførsel tar overhånd.

4.3.2 Alignment angle

Eksperimentet er illustrert i `aliAngle`-filen.

Med det parametersettet flokken har nå viser en endring i synsfeltet til Alignment-oppførsel liten forskjell i oppførsel til den totale flokken. Det er ikke mulig å se en merkbar endring før alignment angle justeres til under 30 grader. Da ser man at den uniforme flyten til flokken løses opp, ingen boider tar hensyn til andre boiders retning lenger.

Hvis man nullstiller flokken slik at alle boidene er ukordinert kan man se at Alignmentangle har stor betydning på hvor lang tid boidene bruker til å finne sammen til én flokk. Men hvis de allerede er stilt opp med naboer har parameteren liten innvirkning.

4.3.3 Cohesion angle

Eksperimentet er illustrert i `cohAngle`-filen.

Her observeres det at med Cohesion angle 0 (ingen cohesion) vil flokken forbli statisk og spredt jevnt utover skjermen fordi Separation-oppførsel tar overhånd. Bare en liten økning i vinkel skal til for at boidene begynner å bevege seg.

Det er vanskelig å se en merkbar endring i flokkoppførsel når Cohesion angle økes før den når ca 300 grader. Når vinkelen kommer i dette området ser man at boidene har en tendens til å klumpe seg tettere sammen enn de vanligvis har gjort.

Verdier mellom 90 og 300 grader synes og resultere i de mest uforutsette og komplekse mønstrene.

4.3.4 Separation weight

Eksperimentet er illustrert i `sepWeight`-filen.

Når separation-vekten gradvis økes observeres det en brå endring i flokkens oppførsel når vekten passerer cohesion-vekten. Når denne verdien passeres vil flokken umiddelbart søke ut slik at separation-radius opprettholdes. Dette er et resultat av at separation-oppførselen har en diskontinuitet som vist i fig 3.8(b). Boider vil nå hele tiden ligge på grensen til å være innenfor separation-radiusen, og tvert det skjer, blir de påtrykt en separation-vektor som overgår cohesion-vektoren.

Når separation-vekten justeres under 2,5 observeres det at separation-oppførselen er så godt som fraværende. Flokken vil da bevege seg sammen i lange tynne striper takket være cohesion- og alignmen-oppførsel.

4.3.5 Alignment weight

Eksperimentet er illustrert i `aliWeight`-filen.

Denne parameteren påvirker flokken i størst grad når boider ikke allerede er stilt opp i forhold til hverandre. Man ser at ved økning av alignment-vekten resulterer i at flokken beveger seg mer som en hel organisme. Ved store verdier observeres det at hele flokken beveger seg i samme retning, uten noe som helst dynamikk eller liv i simuleringen.

Ved en minking av vekten ned mot 0 resulterer det at flokken beveger seg helt ukorrelert i forhold til hverandre og selve simuleringen ser meget kaotisk ut.

4.3.6 Cohesion weight

Eksperimentet er illustrert i `cohWeight`-filen.

Ved økning av cohesion-vekt observeres det at flokken oppfører seg mer og mer kaotisk og mindre rasjonelt. Når derimot vekten minkes er det separation- og-alignment-oppførselene som hovedsaklig styrer flokken. Dette resulterer i at flokken beveger seg i jevne linjer og med en fast avstand mellom hverandre.

Hvis derimot cohesion-vekten minkes ned mot 0 stopper flokken opp, fordi det ikke er noen drivkraft som gjør at boidene ønsker å bevege seg *mot* noe. Dette er samme observasjon som under justering av cohesion radius.

4.3.7 Separation radius

Eksperimentet er illustrert i `sepRadius`-filen.

Denne parameteren angir minimumsavstanden separation-oppførselen forsøker og opprettholde. Ved å øke denne parameteren observeres det at flokken beveger seg mer spredt. Hvis denne forblir høy vil flokken etter en liten stund fordele seg uniformt over hel skjermen.

Når radien minkes tillater man også at boidene beveger seg nærmere hverandre. Hvor fort dette skjer avhenger av forholdet mellom cohesion- og alignment-oppførsel. Høy cohesion gjør at de vil nærme seg hverandre fort, mens høy alignment vil motvirke dette.

Hvis separation-radius minkes ned mot 0 er separation-oppførsel fraværende og flokken vil bevege seg i tynne striper, samme observasjon som ble gjort med justering av separation weight.

4.3.8 Alignment radius

Eksperimentet er illustrert i `aliRadius`-filen.

Endring av denne parameteren er litt vanskelig å observere og peke ut direkte hvilke endringer den forårsaker på flokksimuleringen. Siden alginment-vekten er godt under separation- og cohesion-vekten vil de sannsynligvis dominere.

Når radien justeres ned mot 0 observeres samme oppførsel som når alignment-regelen blir oversett (ikke spesielt uventet).

4.3.9 Cohesion radius

Eksperimentet er illustrert i `cohRadius`-filen.

Endring av denne parameteren er også litt vanskelig å observere og peke ut endringer som skjer i flokkoppførselen.

Når cohesion-radius minkes ned mot 0 vil flokken vise tendenser til å stoppe opp, akkurat som i ekperimenter der cohesion-angle eller -weight blir minket betydelig.

4.3.10 Endring av deler av flokkparameterene

Eksperimentet er illustrert i `10percentExperiment`-filen.

Testriggen er laget med tanke på at det kan utføres justering av parametere på kun deler av flokken.

Ved å la flokken få forskjellige flokkparametere kan man teste hvor godt de forskjellige oppførselene kommer fram.

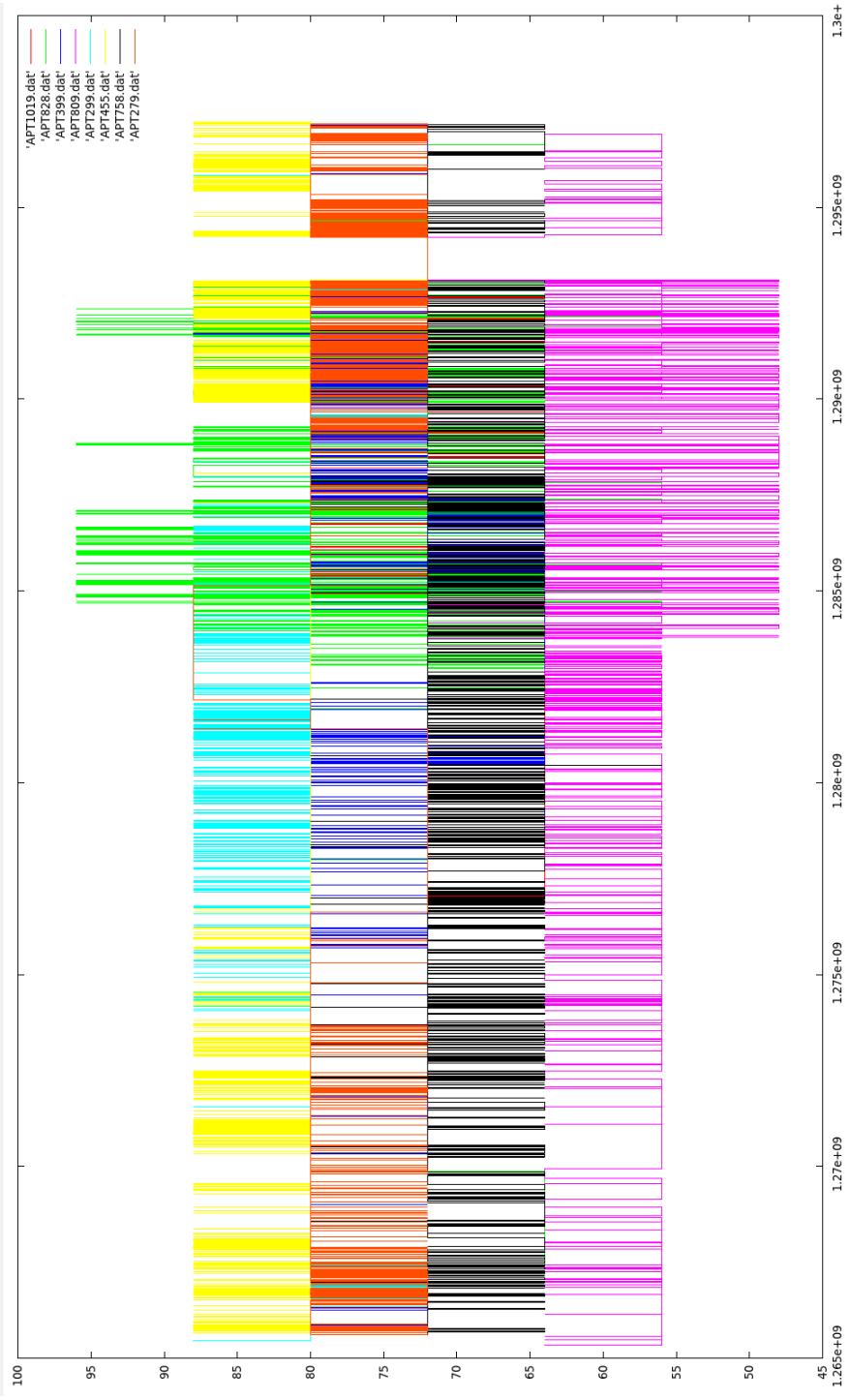
Når slike eksperimenter foregår må resten av flokken utøve noenlunde rasjonell oppførsel, hvis ikke blir det svært vanskelig å få øye på enkeltboidene som beveger seg etter andre parametere.

Filmen viser at det er mulig å skille ut boider som har større separationkraft, i forhold til resten av flokken. De aktuelle boidene blir for enkelhets skyld fargelagt. Andre parametere enn separation er vanskelig å si om resulterer i noen merkbar forskjell i oppførsel til de aktuelle boidene.

4.4 Loggdata

Loggdata for det siste året blir hentet ut. Diskrete hendelser slik som feil, utkoblinger, forvarsel og brannalarm er lagret i en separat fil. Målerverdier for røyk-/varmedetektorene består av ca 6,3 MB binærdata. Én måleverdi tar 7 byte. Det vil si at loggsettet inneholder 900 000 målinger. Dette betyr igjen at hvert punkt melder en endring i måleverdi i snitt ca 2,8 ganger om dagen. Ved å se litt nærmere på datasettet ser man at 20 % av detektorene står for 50 % av målingene. Faktisk så har 10 % av detektorene ikke registrert en eneste endring i måleverdi i løpet av et helt år.

Figur 4.1 viser data fra noen av detektorene som har hatt litt over middels aktivitet. Som man ser består så godt som alle hendelsene av endring mellom to faste nivå. Dette er et resultat av at detektorene er konfigurert med et *sliding window filter* som gjør at detektorene ikke rapporterer tilstandsendring før de har nådd en gitt forskjell fra forrige måling.



Figur 4.1: Data over ett år fra noen utvalgte detektorer

Kapittel 5

Diskusjon

Oppgaven har så langt tatt for seg implementasjon og testing av en flokksimulering basert på Reynolds paper fra 1987 [28]. Algoritmen er optimalisert tilstrekkelig slik at det er mulig å kjøre cirka 6 000 boider på en moderne PC uten at simuleringshastigheten blir for lav.

Ved å benytte et høyt antall boids er det lettere å se hvordan små justeringer av parametere påvirker hele flokken ettersom det visuelle intrykket forsterkes når mange agenter utøver lik oppførsel.

5.1 Endring av parametere

Under eksperimenter som er gjort for å prøve å kartlegge forskjellige typer flokkoppførsel kan det observeres et visst antall karakteristiske trekk ved flokken. Disse er bl.a.

- Flokken stopper opp, eller boider står bare og "dirrer" på samme sted
- Flokken klumper seg sammen til meget kompakte enheter hvor flere boider plasserer seg oppå hverandre
- Flokken beveger seg tilsynelatende uten kaotisk, det er ingen bevegelsene som kan forklares objektivt
- Flokken beveger seg i en jevn strøm over hele skjermbildet, uten noe dynamikk

Endringer som gjøres er lettest å observere når parameterene blir satt til det ekstreme. Her er det separation og cohesion-oppførsel som er mest fremtredende.

Det er fullt mulig å identifisere én enkel boid som har fått justert en parameter til det ekstreme blant en flokk som beveger seg "normalt". Men hvis det flere av boidene har denne egenskapen påvirker de også resten av flokken slik at de boider som er i normaltilstand gjerne vil oppføre seg unormalt.

5.2 Andre algoritmer

Det er mulig å benytte flocking-algoritmer til andre ting enn visualisering.

Boider kan benyttes i såkalte clustering-algoritmer. Dette er algorimer som forsøker å finne en ansamlig punkter i rommet. Hvis slike skal benyttes til branndeteksjon kan man for eksempel se for seg et todimensjonalt område hvor y-aksen tilsvarer detektorverdi og x-aksen tilsvarer fysisk lokasjon av detektoren. Hvis det blir en stor ansamling av punkter på en spesiell del av x-aksen vet man at det er aktivitet der.

5.3 Boider brukt for å fange interesse

En ting som kan være praktisk med slike flokksimuleringer er at de er fascinerende dynamiske og intressante å se på. Dette kan benyttes til å oppfordre en person til å kaste et blikk på simuleringen ofte. Ved en hendelse i systemet kan visualiseringen klart vise fra med for eksempel endret eller pulserende farge at det har skjedd noe. Andre ting som kan vekke oppmerksomhet er å endre lengde på trails, endre størrelse på boidene, endre bakgrunn osv.

Kapittel 6

Konklusjon

Å benytte flocking til en abstrakt visualisering for en systemtilstand er mulig. Men det er sannsynligvis ikke hensiktsmessig å mappe flocking-parameter til "analoge" data fra detektorer slik som røykverdi, temperatur, og/eller nedstøvingkompensering. Ihvertfall ikke i utstrakt grad.

En bedre løsning er heller å justere flocking-parameterene på forhånd slik at det utøves intressant og dynamisk flokkoppførsel, og kun la diskrete hendelser i brannalarmsystemet påvirke visualiseringen. Dette vil for eksemepel å la fargen til en boid pulsere rødt/gult hvis et punkt står i en feiltilstand eller liknende. Om boidene har lange trails vil man enkelt få med seg dette siden fargeendringene ligger igjen i trail-sporet. En annen idé kan være å for eksempel vekselvis endre en av flokkparameterene slik at flokken tilsynelatende "pulserer" hvis det en hendelse som er intressant.

Det er heller ikke ønskelig å la én boid i visualiseringen tilsvare ett punkt i systemet. I store anlegg med flere tusen punkt vil dette resultere i tilsvarende antall boids på skjermen. Som regel vil majoriteten av boidene være i "normaltilstand", og oppføre seg (til en viss grad) uventet og overraskende. Hvis én eller to boider skal visualisere en gitt diskret tilstand vil dette være svært vanskelig å observere da brukeren kan bli overveldet av den "normale" og tilsynelatende kaotiske aktiviteten på skjermen. I dette tilfellet vil boidene som skal represetere en normaltilstand fungere som kamuflasje for de få boidene man faktisk ønsker skal få oppmerksomhet

Ved å justere andre variabler som ikke har noen direkte sammenheng med flokksimuleringen slik som farge, sporlengde, tykkelse eller maksimalfart vil dette være en mer effektiv måte å gi en oversikt over et sett parametere som skal fremstilles. Når det er slike enkle parametere som justeres er det heller ikke behov for en underliggende flokksimulering selv om dette likevel kan

gjøre visualiseringen innbydene og intressant å se på.

6.1 Videre arbeid

Videre arbeid med visualiseringen kan være å jobbe mer med å det til å se pent og innbydene ut. Oppgaven har ikke prioritert visuell kvalitet først. Her er det bare fantasien som setter grenser. Man kan legge til HDR Bloom-effekt, motion blur, partikkelemittere i spissen av boider, levende bakgrunn osv.

En annen måte og forsterke eller øke den visuelle kvaliteten kan være å utvide til å benytte 3D i simuleringen For at dette skal komme fram noenlunde greit må det velges en bakgrunn som fremhever perspektivet. Det er også mulig og la kameraet bevege seg og holde fokus på en gruppe boids som en mener bør ha mest oppmerksomhet. Da vil perspektivfølelsen forsterkes. Ved å plassere små statiske referansepunkter små som prikker, stjerner eller liknende, vil det perspektivfølelsen forsterkes ytterligere se for eksempel `http://vimeo.com/2543780` og `http://vimeo.com/6088060` hvor det er laget boidsimuleringer i 3D med dynamisk kameraføring.

Hvis det skulle være intressant å visualisere mange boider på dårligere hardware bør det lages en skikkelig implementering av romlig grid-database som kjører på skjermkortet.

Bibliografi

[1] AMD. An Introduction to OpenCL. `http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx`, mai 2011.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] Aristotle. *Aristotle's metaphysics*. Oxford University Press, 1924. Translation from W.D. Ross.

[4] Nick Artim. An introduction to fire detection, alarm, and automatic fire sprinklers. `http://www.nedcc.org/resources/leaflets/3Emergency_Management/02IntroToFireDetection.php`, 2007.

[5] Pilgrim Beart. Emergent. `http://www.beart.org.uk/Emergent/`, 2003.

[6] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays*, volume 2. Academic Press, 1982.

[7] Dirk Vanden Boer. General-purpose computing on gpus. Master's thesis, Transnational University Limburg, 2005.

[8] Jens Breitbart. A framework for easy CUDA integration in C++ applications. *Diplome thesis, Universität Kassel, Kassel, Germany*, 2008.

[9] Rodney P. Carlisle. *Encyclopedia of play in today's society*. Sage Publications, Inc, 2009.

[10] Ugo Erra, Rosairo De Chiara, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In *Proceedings of vision, modeling and visualization*, novembeer 2004.

[11] Ugo Erra, Bernardio Frola, and Vittorio Scarano. A GPU-based method for massive simulation of distributed behavioral models with CUDA. In *Proceedings of the 22nd Annual Conference on Computer Animation and Social Agents*, juni 2009.

[12] Toni Feder. Statistical physics is for the birds. *Physics Today*, 60, 2007.

[13] David W. Gohara. Episode 2 - OpenCL Fundamentals. `http://www.macresearch.org/files/opencl/Episode_2.pdf`, august 2009.

[14] Experimental Media Group. About nodebox. `http://nodebox.net/code/index.php/About`, 2011.

[15] Khronos Group. OpenCL Overview. `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`, mars 2010.

[16] Apple Inc. OpenCL Programming Guide for Mac OS X. `http://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf`, juni 2009.

[17] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Preformance Comparison of CUDA and OpenCL, 2010.

[18] James Kennedy and Russel C. Eberhart. *Swarm Intelligence*, chapter 3, pages 81–132. Morgan Kaufmann, 2001.

[19] Pyarelal Knowles. GPGPU Based Particle System Simulation. Technical report, School of Computer Science and Information Technology, RMIT University, 2009.

[20] Lutz Latta. Building a million particle system. In *Game Developers Conference*, 2004.

[21] Helmut Lorek and Matthew White. Parallel bird flocking simulation, mai 1993.

[22] Pattie Maes. Artificial life meets entertainment: lifelike autonomous agents. *Communications of the ACM*, 38(11):108–114, 1995.

[23] Christopher Oat, Joshua Barczak, and Jeremy Shopf. Efficient Spatial Binning on the GPU. Technical report, AMD Inc., februar 2009.

[24] OpenEndedGroup. Welcome to field. `http://openendedgroup.com/field/wiki`, 2011.

[25] Arch Ronel K. Pabico. Water fountain tutorial. `http://www.3dallusions.com/forums/tutorials/2144-water-fountain-tutorial.html`, august 2007.

[26] Conrad Parker. Boids pseudocode. `http://www.vergenet.net/~conrad/boids/pseudocode.html`, 2007.

[27] William T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Transactions on graphics*, 2(2):91–108, 1983.

[28] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Annual Conference on Computer Graphics*, pages 25–34, 1987.

[29] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, 1999.

[30] Craig W. Reynolds. Interaction with groups of autonomous characters. In *Game Developers Conference*, 2000.

[31] Craig W. Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121. ACM, 2006.

[32] Craig W. Reynolds. Opensteer preliminary documentation. `http://opensteer.sourceforge.net/doc.html`, juni 2007.

[33] Hanan Samet. *Spatial data structures*. ACM Press and Addison-Wesley, New York, 1995.

[34] Adam Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, The University of Kent, september 2008.

[35] Alessandro Ribeiro Da Silva, Wallace Santos Lages, and Luiz Chaimowicz. Boids that see: Using Self-Occlusion for Simulating Large Groups on GPUs. *Computers in Entertainment (CIE)*, 7(4):51, 2009.

[36] Andrew Sorenson. What is impromptu? `http://impromptu.moso.com.au/`, 2011.

[37] Susan Stepney, Fiona A. C. Polack, and Heater R. Turner. Engineering emergence. In *Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on*, pages 9–pp. IEEE, 2006.

[38] the**barbarian**group. About — cinder. http://libcinder.org/about/, 2010.

[39] Zach Lieberman & Theodore Watson. About: history. http://www.openframeworks.cc/about, 2011.

[40] Processing wiki. Processing:about. http://wiki.processing.org/w/Processing:About, 2010.

[41] Wikipedia. Euler method. http://en.wikipedia.org/wiki/Euler_method, mai 2011.

[42] Wikipedia. Impromptu (programming environment). http://en.wikipedia.org/wiki/Impromptu_(programming_environment), mai 2011.

[43] Wikipedia. Nodebox. http://en.wikipedia.org/wiki/NodeBox, mai 2011.

[44] Wikipedia. OpenCL. http://en.wikipedia.org/wiki/OpenCL, mai 2011.

[45] Wikipedia. openframeworks. http://en.wikipedia.org/wiki/OpenFrameworks, mai 2011.

[46] Wikipedia. Processing (programming language). http://en.wikipedia.org/wiki/Processing_(programming_language), mai 2011.

[47] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601, 1983.

[48] Bo Zhou and Suiping Zhou. Parallel simulation of group behaviors. In *Proceedings of the 2004 Winter Simulation Conference*, pages 364–370, 2004.