Examensarbete

BRED KOLLISIONSDETEKTERING FÖR SPEL

BROAD PHASE COLLISIONS DETECTION FOR GAMES

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2015

Robin Iderström

Handledare: Mikael Thieme
Examinator: Sanny Syberfeldt

Sammanfattning

Detta arbete undersöker 4 olika algoritmer som används för bred kollisionsdetektering. De olika metoderna för bred kollisionsdetektering kan delas in i 4 olika kategorier. Algoritmerna som är valda att representera varje kategori är Bruteforce, Sweep and prune, Hierarchical grid och Bounding volume hierarchy. Fokus i arbetet ligger i att mäta deras lämplighet för spelmotorer där det är viktigt att algoritmerna kan köras i realtid.

Algoritmerna körs i simulationer av olika miljöer. Där antalet objekt, hur många som är rörliga och objektens distribution varieras mellan miljöerna. I simulationerna mäts tiden det tar för varje algoritm att exekvera per frame.

Resultaten av mätningarna visar Hierarchical grid är den bästa av algoritmerna för att hantera stora mängder objekt. Sap passar bäst för få objekt och Bounding volume hierarchy är ett stabilare alternativ.

Resultaten kan användas för att välja en lämplig bredfas algoritm vid implementation av en spelmotor.

Nyckelord: bredfas, kollisionsdetektering, kollision, fysik, spelmotor.

Innehållsförteckning

1 Introduktion

Kollisionsdetektering är ett viktigt problem inom många områden som robotik, fysikmodellering och datorspel. Målet med kollisionsdetektering är att upptäcka kontakt som ska eller har skett mellan ett eller flera objekt i miljön. Effektiva algoritmer spelar en stor roll för att kunna göra detta i realtid och mycket forskning har lagts åt att utveckla och ta fram nya algoritmer för att minska ner beräkningstiden. Den naiva lösningen för att hitta kollisioner mellan objekt i en miljö är att för varje objekt göra ett kollisionstest mot alla andra objekt i miljön. Denna lösning har en komplexitet på $O(n^2)$ och är olämplig för applikationer som har många objekt i miljön och behöver köras i realtid. Kollisionsdetektering brukar därför delas upp i 2 faser, bred kollisionsdetektering och smal kollisionsdetektering. Detta arbete undersöker och jämför olika algoritmer som används för bred kollisionsdetektering, fokus ligger på deras prestanda för användning i realtidsapplikationer.

Algoritmerna som jämförs är Sweep and prune, Hierarkiskt grid och en variant av Bounding volume hierarchy (AABB tree). Dessa algoritmer har använts inom flera olika spelmotorer/fysikmotorer som Bullet (Coumans, 2013), Box2D (Catto, 2007) och Havok (Havok, 2011). Jämförelsen sker i en 3d miljö där mängden objekt, distributioner mellan vart de är placerade i miljön och andelen statiska objekt i miljön varieras, för att få data över hur de presterar i olika situationer som kan uppstå i spel.

Experimenten utförs genom att sätta upp och köra simulationer i en existerande spelmotor. Från simulationerna exporteras positionsdata in till ett separat mätningsprogram som kör alla bredfas algoritmer med datan och mäter exekveringstiden för dem.

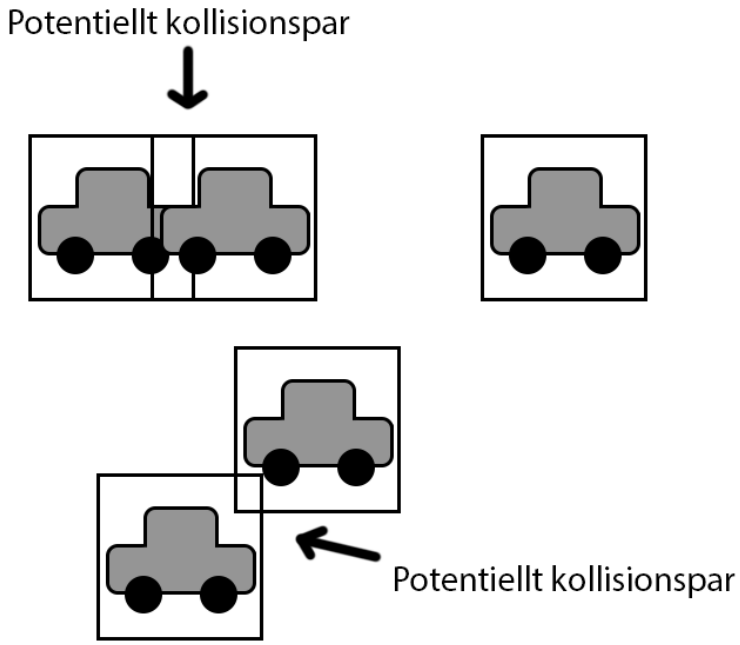
2 Bakgrund

Den naiva lösningen för att hitta kollisioner i en miljö med n stycken objekt är att för varje objekt göra ett kollisionstest mot de andra n-1 objekten. Detta tar n(n-1)/2 kollisionstest, en komplexitet på $O(n^2)$. Till exempel kräver en miljö som består av 100 objekt hela 4950 stycken kollisionstest med den här metoden. Beräkningstiden för varje kollisionstest beror på komplexiteten på modellen. Enklare modeller som kuber eller sfärer är inte särskilt beräkningstunga men för mer komplicerade modeller som en spelkaraktär kan det bli mycket dyrt att beräkna om en kollision har skett. Detta gör oftast kollisionsdetektering en flaskhals för prestandan om det inte är mycket få objekt i miljön.

Kollisionsdetektering brukar därför delas upp i 2 olika faser, bred och smal (Hubbard, 1995). Målet med det är att reducera beräkningsbelastningen genom att göra billiga test för att utesluta de objektpar som inte kan kollidera. Bred kollisionsdetektering tar en lista över alla objekt i miljön och skapar en lista med alla objektpar som potentiellt kan kollidera. I smal kollisionsdetektering görs exaktare beräkningar för att avgöra om en kollision mellan objektparen som togs ut i den breda fasen har skett. Valen av algoritmer för de båda faserna kan göras oberoende, eftersom den breda fasen agerar som ett filter för den smala kollisionsdetekteringen.

2.1 Bred kollisionsdetektering

Syftet med bred kollisionsdetektering är att hitta de objektpar som potentiellt kan kollidera och rensa bort resten för att undvika att göra dyra uträkningar mellan objekt som inte kan kollidera. Medan det är acceptabelt att kollisioner som inte skett blir undersökta vidare i den smala fasen är det mycket viktigt att kollisioner som har skett inte blir bortfiltrerade av den breda kollisionsdetekteringen (Culley & Kempf, 1986).



Figur 1 Visar en 2d miljö med 5 bilar i som alla omsluts av en gränsvolym var. Bilden har markerat 2 stycken potentiella kollisionspar som behöver undersökas i den smala kollisionsdetekteringen.

Figur 1 ovan visar en scen med 5 objekt i. Poängen med bred kollisionsdetektering är att undvika att göra dyra kollisionsuträkningar på objekt som inte kan kollidera. Om en bruteforce-lösning hade används på denna scen hade varje objekt behövt göra ett kollisionstest mot alla andra objekt i scenen. I det här fallet hade det lett till 10 tester, en bred kollisionsdetekteringsalgoritm hade kunnat reducera det till de 2 potentiella kollisionsparen som visas i bilden.

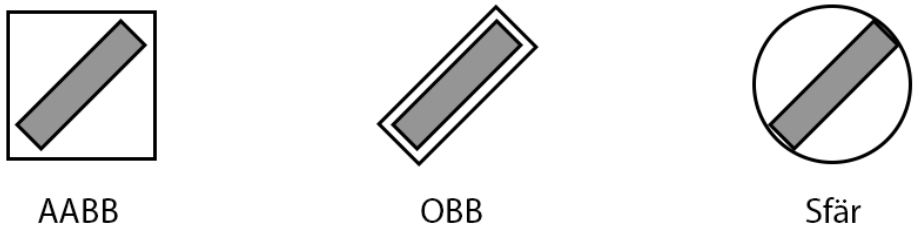
För att hålla beräkningarna i det här steget billiga används inte objektets exakta geometri utan modellen approximeras istället med en enklare geometrisk volym som omsluter hela objektet. Flera olika sorter av gränsvolymer existerar men Axis-Aligned Bounding Box (AABB), Oriented Bounding Box (OBB) och sfärer är bland de vanligaste.

Många olika algoritmer för bred kollisionsdetektering har blivit föreslagna men de går generellt att dela in i 3 olika kategorier: Topologi (Sweep and prune), Spatial (Grid, Octree osv) och Bounding volume hierarchy (Xiao-rong, et al., 2009).

2.2 Gränsvolymer

Gränsvolymer är enkla geometriska volymer som omsluter ett objekt helt. Dessa används för att undvika att göra beräkningarna i den breda kollisionsdetekteringen på objektens exakta geometri. Om 2 objekts gränsvolymer inte överlappar kan inte objekten i dem kollidera eftersom de omsluts helt av dem. Beräkningarna i bredfasen görs med volymernas enklare geometri för att hålla beräkningskostnaderna nere. Flera olika typer av gränsvolymer har använts för detta syfte.

För att minimera antalet potentiella kollisioner som sker i den breda kollisionsdetekteringen bör gränsvolymen vara så minimal som möjligt men fortfarande omsluta hela objektet. Detta är inte alltid fallet, (Hubbard, 1995) använder en implementation där volymen expanderas efter objektets hastighet för att se till att ingen kollision missas mellan fysikuppdateringar. Gränsvolymer kan även utökas för att minska ner på omstruktureringen som behöver ske när ett objekt förflyttas (Baraff, 1992).



Figur 2 Olika typer av gränsvolymer applicerad på en roterad rektangel. Bilden illustreras i 2d men samma principer gäller i 3d.

Axis-aligned bounding box (AABB) är en box som alltid är orienterad efter koordinataxlarna (se Figur 2 för illustrering). Fördelen med den här varianten är att den tar upp lite lagringsutrymme och det är billigt att göra intersektionstest mellan dem (Xing, et al., 2010). Dock måste boxen omformas när objektet roterar för att se till att objektet omsluts helt, volymen kan utökas till en storlek som täcker objektet vid alla dess rotationer för att undvika omformning, till bekostnad av dess täthet.

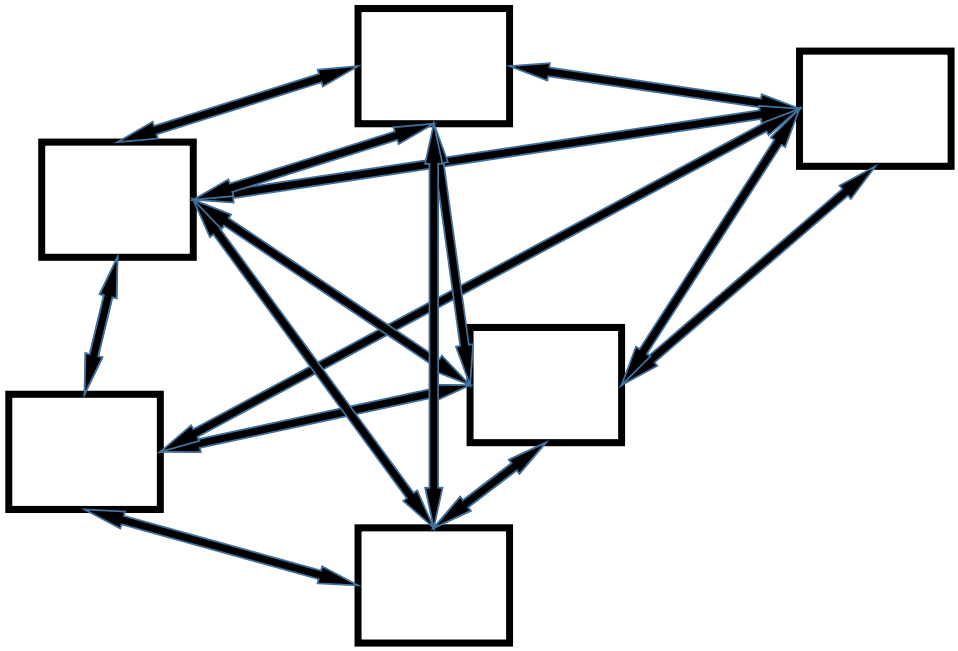
Orientated bounding box (OBB) är till skillnad från AABB, en box som följer objektets orientering (se Figur 2 för illustrering), vilket i många fall leder till en mycket kompakt gränsvolym (Gottschalk, et al., 1996). Detta kommer till bekostnad av att det är dyrare att göra intersektionstest med OBBs jämfört med andra gränsvolymer.

Sfärer används också ofta som gränsvolymer (se Figur 2 för illustrering). Dessa har fördelen att vara oberoende av objektets rotation och behöver därför inte beräknas om när objektet roterar. De tar även upp väldigt lite lagringsutrymme eftersom endast mittpunkt och radie behöver lagras, utöver detta är också intersektionstester mycket billiga. Dock är de oftast dåliga på att representera objektets geometri vilket kan leda till många fler potentiella kollisioner som behöver undersökas i den smala kollisionsdetekteringen.

Andra typer av gränsvolymer har också använts för bred kollisionsdetektering. Dessa inkluderar cylindrar, kapslar och olika trädstrukturer som Octree (Bandi & Thalmann, 1995).

2.3 Naiv lösning

Den naiva lösningen för bred kollisionsdetektering (Brute-force search eller Exhaustive search), går igenom alla objekt och för varje objekt görs kollisionstest mot alla andra gränsvolymer (illustreras i Figur 3). Den här metoden har en komplexitet på $O(n^2)$ vilket är acceptabelt om det är få objekt i miljön men blir snabbt ineffektiv om det är många objekt i miljön. Pseudokod för brute-forcealgoritmen finns i Figur 4.



Figur 3 En illustrering av de kollisionstester som behöver ske i brute-force lösningen. Varje pil representerar ett kollisionspar.

```
function bruteforce(Object[] objects){
      ObjectPairList p;
      for (i = 0; i < objects.length; i++){
            for (j = i+1; j < objects.length; j++){
                  if(objects[i] intersects with objects[j]){
                        add objects[i] and objects[j] as a pair to p
                  }
            }
      }
      return p;
}
```

Figur 4 Pseudokod för brute-force.

2.4 Spatiala metoder

Dessa algoritmer delar generellt in miljön i celler och varje gränsvolym tilldelas sen till sina motsvarande cell(er) beroende på deras placering i miljön (Overmars, 1992). Potentiella kollisioner hittas om flera objekt tillhör samma cell. Den enklaste varianten av det här kallas Grid.

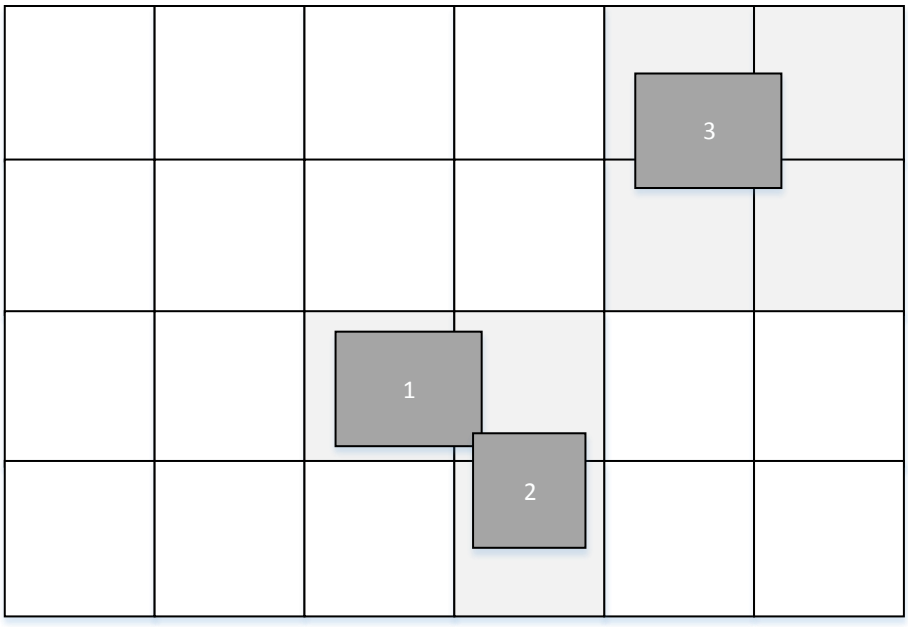
2.4.1 Uniform Grid

Grid, även känd som Uniform grid delar in världen i celler av en satt storlek. Varje objekt i miljön tilldelas sen till varje cell dess gränsvolym överlappar med. När det kommer till insättning i gridstrukturen finns det 2 olika strategier.

1. Om en gränsvolym överlappar ett flertal gridceller, sätts objektet in i alla celler den överlappar med.
2. Om cellstorleken är större än den största gränsvolymen, räcker det att endast sätta in objektet i en cell. Denna metod kräver dock att närliggande celler också undersöks när kollisionerna ska hittas.

När alla objekt är insatta i cellerna hittas de potentiella kollisionsparen genom att köra brute-forcealgoritmen mellan alla objekt som är tilldelade till samma celler. Uniform grid är mycket lätt att implementera men tar också upp mycket onödigt lagringsutrymme om miljön är glest populerad, eftersom celler skapas även om objekt inte existerar i det området (Ming & Gottschalk, 1998).

För att minska ner på uppslagningstiden kan cellerna istället lagras i 1D hashtable. Objektens koordinater i miljön körs igenom en hashfunktion för att få ut vilket index i tabellen de ska lagras på. En av de största fördelarna som kommer ifrån den här lagringsmetoden är att uppslagningar och insättningar kan göras på konstant tid. Denna metod kallas för ”Spatial hashing” (Overmars, 1992). (Hastings, et al., 2005) tillämpar spatial hashing för att inte bara optimera kollisionsdetektering men också samtidigt optimera rendering och Ai-rutiner.

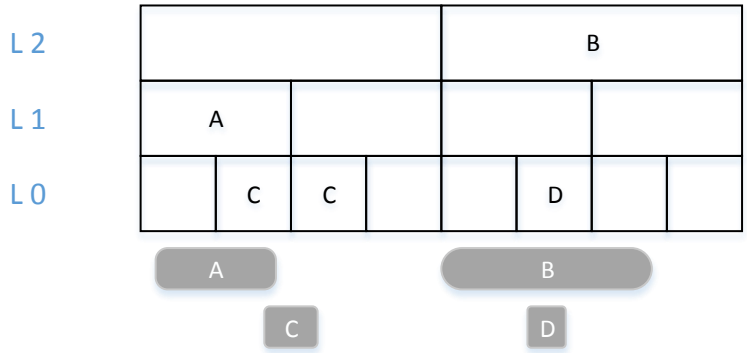


Figur 5 Uniform grid, i exemplet i bilden har en potentiell kollision hittats mellan objekt 1 och 2 eftersom de båda tillhör samma cell.

Det är viktigt att välja en storlek på cellerna som passar storleken på objekten som finns i miljön. Om cellerna är för stora leder det till fler potentiella kollisioner och om de är för små krävs det mycket mer lagringsutrymme och insättningen i celler blir mycket långsammare. En bra vald cellstorlek är mycket viktig för prestandan, men kan i en del miljöer vara mycket svår att välja om storleken på objekten varierar kraftigt.

2.4.2 Hierarkiskt grid

För att lösa problemet med en miljö som har kraftigt varierade storlekar på objekt föreslår (Eitz & Lixu, 2007) ett hierarkiskt grid. Miljön delas in av flera grids som alla delar in samma utrymme men har olika cellstorlekar, vilket skapar en hierarki av grids. Objekten sätts in i det grid som har minst cellstorlek men som fortfarande har en cellstorlek som är större än objektets gränsvolym (Se Figur 6). På detta sätt finns det ett grid med passande cellstorlek för varje objekt i miljön och bra prestanda går att få även om objektens storlek varierar mycket. Dock kommer detta till ett pris i form av prestanda. Det räcker inte längre att bara kolla objekt mot de andra objekten i samma cell, man måste också ta hänsyn till objekt i motsvarande celler från grids högre upp i hierarkin.

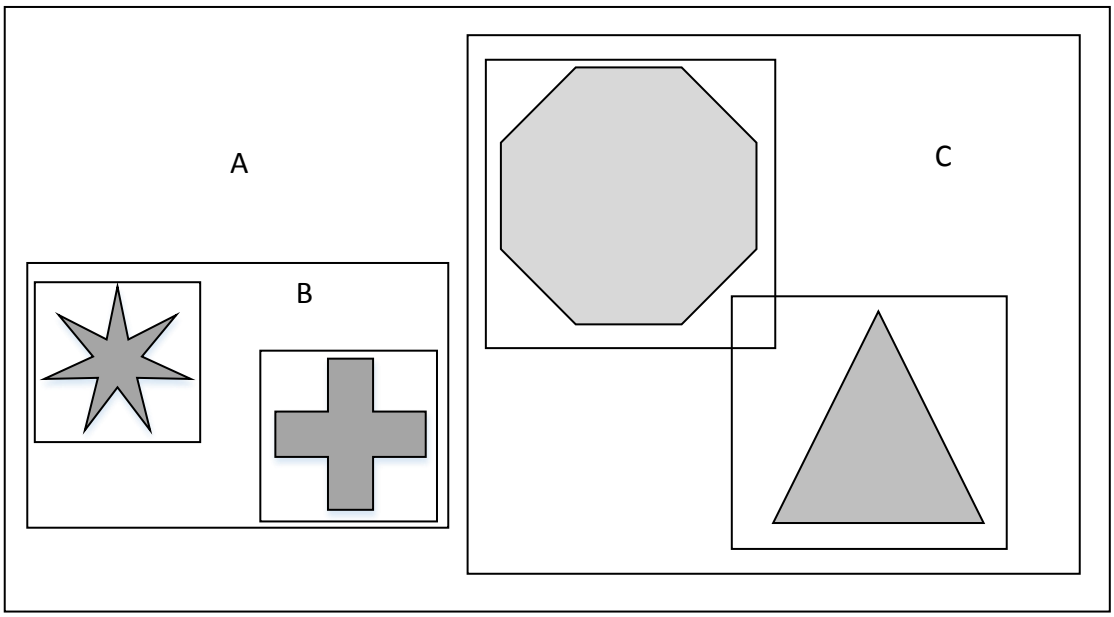


Figur 6 Illustration av hierarkiska grids. Till vänster indikeras vilken gridnivå som visas. Kollisionspar skapas mellan objekt som ligger i samma celler och cellerna som ligger i högre nivåer.

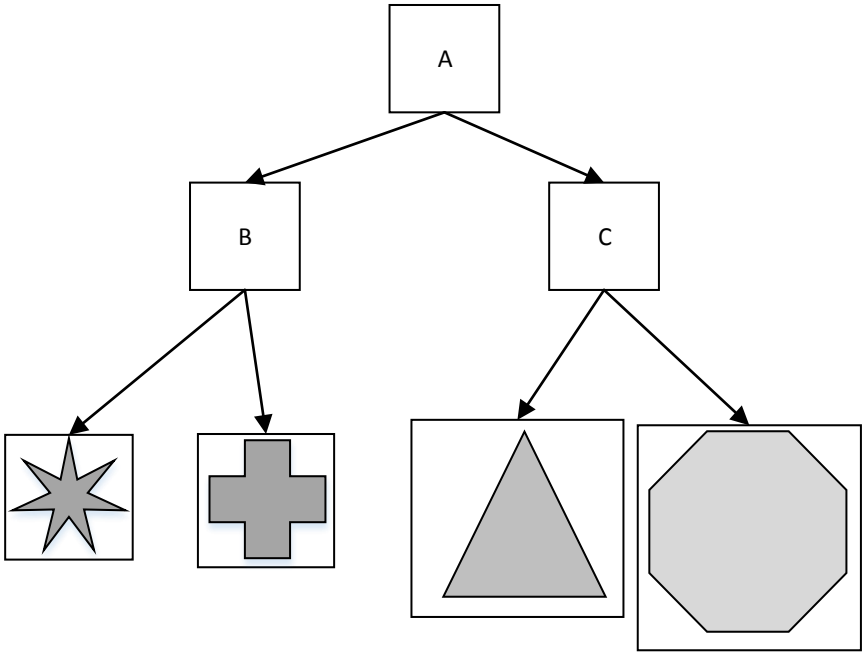
2.5 Bounding volume hierarchy

Bounding volume hierarchy är en metod som är relaterad till spatiala metoder. Användningsområden för bouding volume hierarchy är inte endast begränsade till bred kollisionsdetektering utan det kan även användas till smal kollisionsdetektering (Kockara, et al., 2007) eller ray-tracing (Zhao, et al., 2013).

Istället för att fokusera på att dela in utrymmet i miljön, delas objekten rekursivt in i grupper och omges av en gränsvolym som täcker alla objekt i gruppen. Indelningen sker tills varje grupp endast innehåller ett objekt (Figur 7). Gränsvolymerna som skapas ordnas i ett träd (Figur 8). Detta tillåter stora grupper av objekt att snabbt uteslutas vid kollisionsdetekteringen genom att göra ett intersektionstest mot en gren i trädet. Denna metod sätter inte heller begränsningar på hur stor miljön kan vara som en grid-baserad lösning hade gjort.



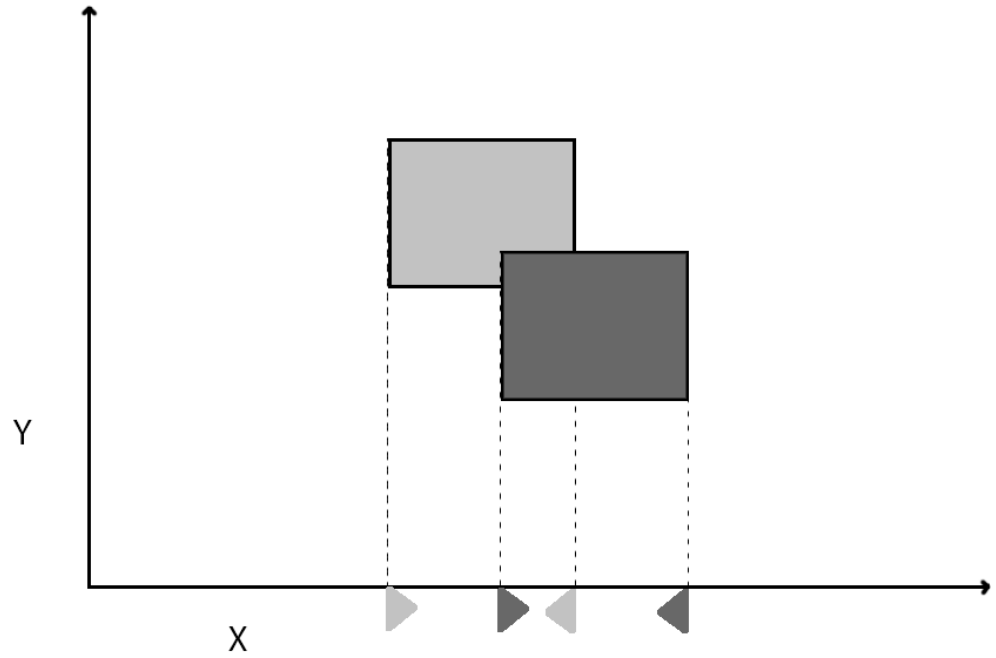
Figur 7 Illustrering av en bounding volume hierarchy. Hela miljön omsluts av en gränsvolym. Volymen delas in i två nya gränsvolymer och detta repeteras tills varje volym bara innehåller ett objekt.



Figur 8 Illustrering av hur träduppbyggnaden ser ut för hierarkin i Figur 7.

2.6 Sweep and prune

Sweep and prune även känd som Sort and prune (Baraff, 1992), använder en annan approach till problemet. Algoritmen bygger på att om objekt ska kollidera i en flerdimensionell miljö måste deras projektioner till lägre dimensioner överlappa (Cohen, et al., 1995).



Figur 9 Sweep and prune i en dimension, start och slutpunkten för gränsvolymerna projiceras på koordinataxeln. Detta görs för alla 3 axlar i en 3d miljö.

Sweep and prune projicerar volymernas start- och slut koordinater till koordinataxlarna (Se Figur 9) och lagrar dessa intervall i en sorterad lista för varje axel. Därefter görs ett pass över listorna för att hitta överlappningar av intervallen, om 2 objekt överlappar på alla axlar har en potentiell kollision hittats. Algoritmen utnyttjar faktumet att objekt inte rör sig långt mellan varje frame genom att behålla listorna från föregående uträkning och sortera om dem med de uppdaterade intervallerna. Om objekten inte förflyttat sig långt kommer listorna nästan vara sorterade vilket gör att sorteringsalgoritmer som insertion sort kan sortera listorna mycket snabbt (Tracy, et al., 2009).

(Cohen, et al., 1995) anmärker att en av de stora fördelarna med Sweep and prune över rumsbaserade metoder är att rumsbaserade metoder måste justera cellstorlekarna efter objekten i miljön för att få optimal prestanda medan Sweep and prune fungerar optimalt på alla miljöer.

2.7 Smal kollisionsdetektering

I den smala kollisionsdetekteringen undersöks de potentiella kollisionsparen som togs fram i den breda kollisionsdetekteringen för att ta reda på var kollisionen har skett, om det har skett en. Eftersom fokus i det här arbetet ligger på bred kollisionsdetektering kommer detta endast kort beskrivas. Mycket forskning har gjorts för att undersöka de olika algoritmerna som används i det här steget (Jiménez, et al., 2001) (Kockara, et al., 2007) (Ming & Gottschalk, 1998).

I spel används oftast approximationer av modellen i den smala fasen, dock mycket närmare approximationer än vad som används i den breda kollisionsdetekteringen. Approximationer

används eftersom helt exakta kollisioner inte krävs för att ge en bra upplevelse och ur spelarens perspektiv oftast inte märks om kollisionen är helt exakt. I simulationer används dock oftast modellens exakta geometri eftersom det kan vara vitalt för simuleringens exakthet.

Många olika algoritmer för att hantera den smala kollisionsdetekteringen har föreslagits, dessa delas in i kategorier av (Kockara, et al., 2007): Feature based, Simplex based, Image-Space based, Volume based och Bounding Volume Hierarchies.

Eftersom algoritmerna i det här steget kan vara väldigt beräkningstunga är det viktigt att den breda kollisionsdetekteringen eliminerar så många objektpar som möjligt, för att kunna köra applikationen i realtid.

3 Problemformulering

Syftet med detta arbete är att utvärdera 4 vanliga algoritmer som används i bred kollisionsdetektering. Algoritmerna som jämförs i studien är bruteforce, hierarkiskt grid, Sweep and prune och AABB tree (bounding volume hierarchy). Algoritmerna har valts ut för att representera de olika kategorierna av lösningsmetoder som togs upp i bakgrunden. Fokus ligger på deras användning i interaktiva virtuella miljöer med fysiksimulering, där prestandan är vital för applikationens interaktivitet, till exempel datorspel eller simuleringar.

I realtidsspelmotorer och fysikmotorer är än idag kollisionshantering en flaskhals för prestandan, speciellt då spelvärldarna växer och blir mer och mer fyllda av avancerade polygonmodeller. På grund av detta det viktigt att välja en bred kollisionsdetekteringsalgoritm som är effektiv och passar miljön som den kommer tillämpas på. I den här studien undersöks några vanliga algoritmer för bred kollisionsdetektering i olika miljöer för att få en bild över hur bra de presterar. Data som samlas in under experimenten kommer att kunna användas för att dra en slutsats om algoritmernas styrkor och svagheter, för att hjälpa utvecklare av spel eller fysikmotorer att välja en bredfasalgoritm som passar deras miljö.

AABB gränsvolymer kommer användas för alla algoritmer. Eftersom fokus ligger på algoritmerna kommer inte en jämförelse mellan olika gränsvolymer göras då detta ligger utanför arbetets område.

3.1 Metodbeskrivning

Algoritmerna i studien jämförs i en 3D-miljö som innehåller en mängd objekt. Dessa objekt är statiska och behåller sin plats genom hela simulationen eller rörliga och förflyttas ständigt. Olika distributioner av startpositionerna för objekten i miljön utvärderas, för att simulera vanliga scenarios som kan uppstå i spel eller simulationer.

Experiment har valts som metod för att utvärdera algoritmerna eftersom algoritmanalys inte ger en bild över hur algoritmerna presterar i olika miljöer. Det är mer värt att veta hur algoritmen presterar i en viss miljö istället för dess värsta eller bästa tidskomplexitet. Experiment har också använts som vald metod i ett antal andra studier som (de Sousa Rocha, et al., 2006), (Cohen, et al., 1995) och (Avril, et al., 2011) bland många mer.

Flera olika experiment utförs där miljön som algoritmerna körs i varieras, för att få en bild över hur de presterar i de olika scenarierna. Alla simuleringar körs i 400 frames för att ge en datamängd som är stor nog för att eliminera spikar i datan och få ett bra medelvärde för exekveringstiden. I miljön sker nedanstående variationer:

1. **Antalet objekt.** Experiment utförs med olika mängder av objekt i miljön för att se hur bra algoritmerna skalar när antalet objekt ökar. Dessa experiment utförs med 100, 250, 500, 1000, 2500 och 5000 objekt i miljön, varav 20 % är rörliga.
2. **Andelen statiska vs dynamiska objekt.** Det utförs experiment där distributionen mellan statiska och rörliga objekt i miljön varieras för att utvärdera hur väl algoritmerna hanterar att miljön förändras. Dessa experiment utförs i en miljö med 1000 objekt och andelen rörliga objekt som utvärderas är 0 %, 20 %, 50 % och 100 %. De resterande andelarna i dessa experiment är statiska och har samma position genom hela simulationen.

3. **Objektens distribution i miljön.** Experiment utförs där algoritmerna körs i olika typer av miljöer. I dessa experiment varieras främst objektens distribution genom miljön.
    - **a.** I det första experimentet slumpas objektens startpositioner ut jämnt över miljön och ingen gravitation appliceras. Denna miljö ska simulera ett scenario som representerar till exempel ett astroidfält.
    - **b.** I det andra experimentet slumpas objekten ut i de lägre 20 procenten av miljön. Denna miljö ska simulera en som vanligtvis förekommer i till exempel fps spel.
    - **c.** I de tredje experimentet placeras objekten ut i 5 kluster i miljön. Denna typ av miljö kan uppstå i tillexempel ett lagerhus där saker är staplade på varandra. Miljön visar hur bra algoritmerna hanterar en miljö där många objekt är placerade mycket nära varandra.
    - **d.** I det fjärde experimentet är objekten staplade som ett torn som sen faller isär och tillslut hamnar i viloläge. Denna typ av miljö kan förekomma i ett spel där byggnader kan förstöras. I denna miljö lagras exekveringstiden och antalet potentiella kollisioner för varje frame, för att sedan kunna skapa en graf över prestandan när miljön kraftigt förändras.

I de olika experimenten mäts exekveringstiden och antalet funna potentiella kollisioner för varje frame i simulationen. Exekveringstiden som mäts är tiden det tar från att algoritmen börjar köra tills den har returnerat en lista med potentiella kollisionspar. Den insamlade datan kan användas för att visa ett medelvärde för hela simulationen eller skapa en graf över hur exekveringstiden förändras över tid. De potentiella kollisionerna som mäts kan användas för att påvisa eventuella samband mellan antalet kollisioner för en frame och tiden det tar för algoritmen att köras.

En liknande studie görs av (de Sousa Rocha, et al., 2006) i denna mäts dock bara FPS-påverkan de olika algoritmerna gav och bara på en sorts miljö med samma objektdistribution genom miljön. Syftet med detta arbete är att göra en grundligare studie över styrkorna och svagheter med de olika algoritmerna.

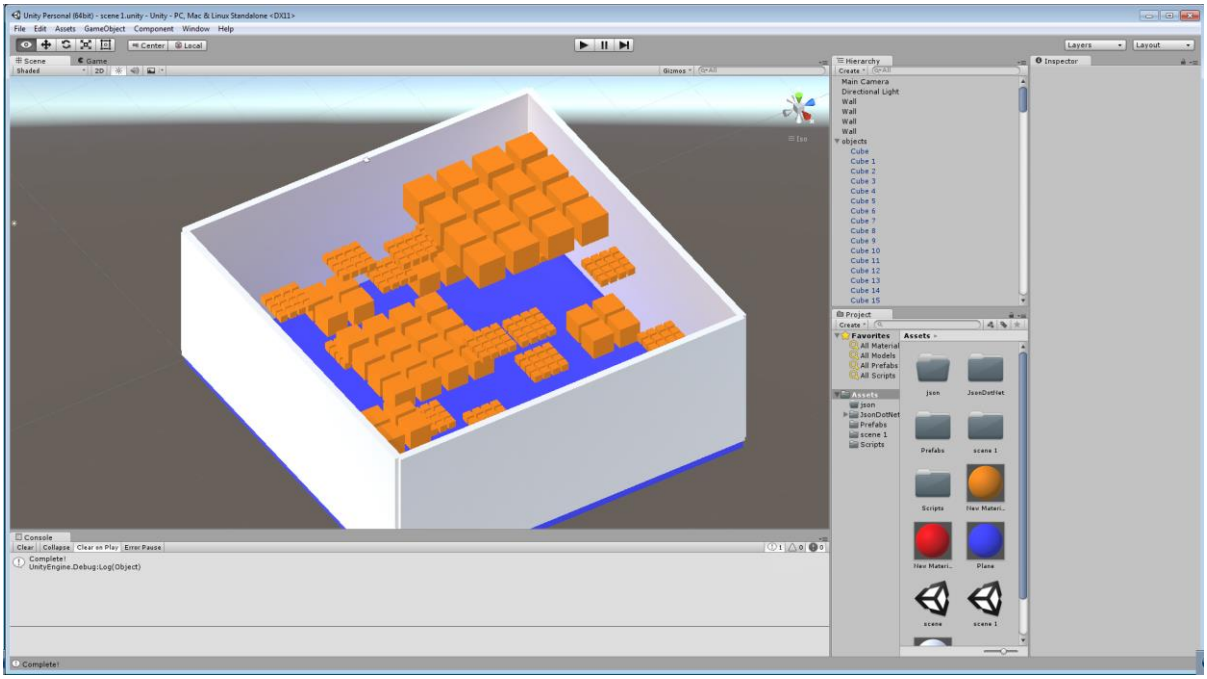
4 Implementation

Implementationerna av experimentmiljön görs i två separata delar, den första delen använder en existerande fysikmotor för att köra simulationen och sparar ner positionsdata för alla objekt vid varje tidssteg. Spelmotorn Unity valdes för detta steg eftersom det är mycket lätt att sätta upp de olika experimentmiljöerna. Den andra delen kör algoritmerna över datan som samlades in under simulationen och mäter tidseffektiviteten samt antalet funna potentiella kollisioner. Den här delen implementeras i c++ för språkets utmärkta prestanda och tillgängligheten till precisa tidmätningsverktyg.

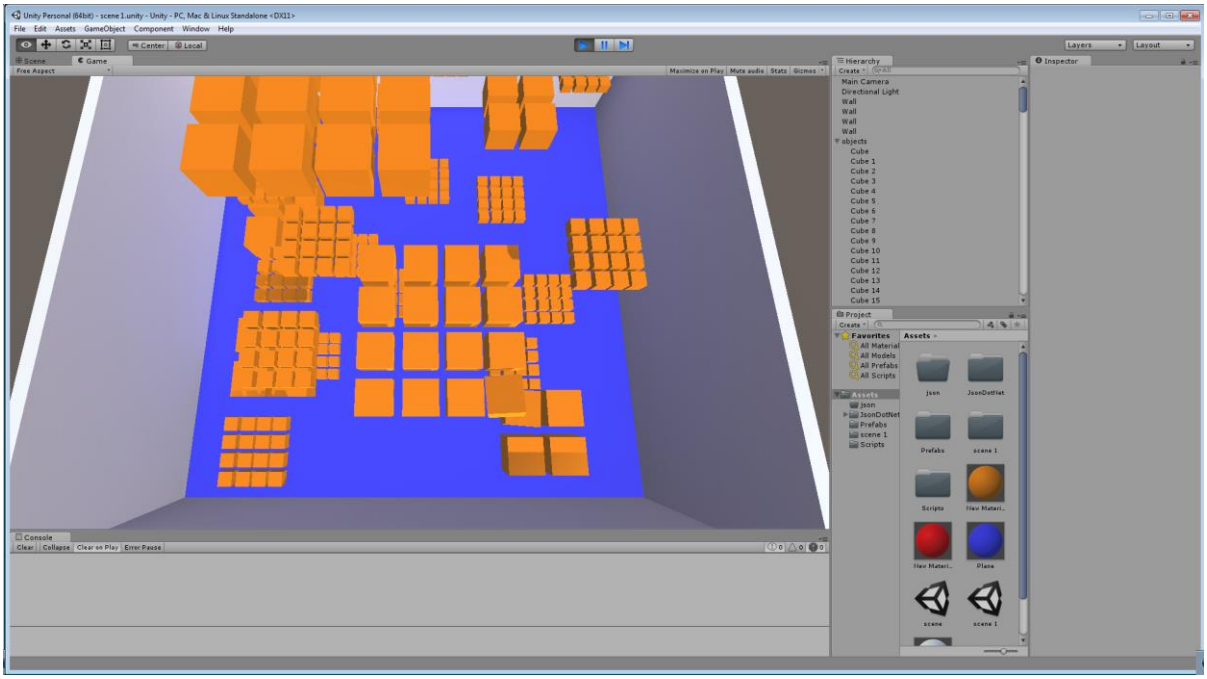
4.1 Unity

Miljöuppsättningen i Unity gjordes med tanken att det skulle vara så lätt som möjligt att skapa miljöer för simulationerna. Ett script skapades som för varje frame lagrar positionerna på alla objekt som ligger på en lägre nivå i objekthierarkin. När ett satt antal frames har passerat sparas den insamlade datan ner till en fil i json formatet. Json valdes för detta över xml eller ett binärt format eftersom det är hyfsat kompakt och lättanvänt, en binär lösning hade eventuellt gett ett kompaktare lagringsformat och snabbare inladdningstid till kostnad av längre utvecklingstid. Eftersom prestandan på den här biten är oviktigt för experimentet valdes json istället. För de tre första miljöerna som beskrivs i 3.1 skapades ett script som skapar alla kuber som ska ingå i simulationen och sedan förflyttar de kuber som är markerade som rörliga. Dessa förflyttas i riktningar som kontinuerligt slumpas om, kuberna kan även inte förflytta sig utanför miljön som är en 100 enheter stor kub.

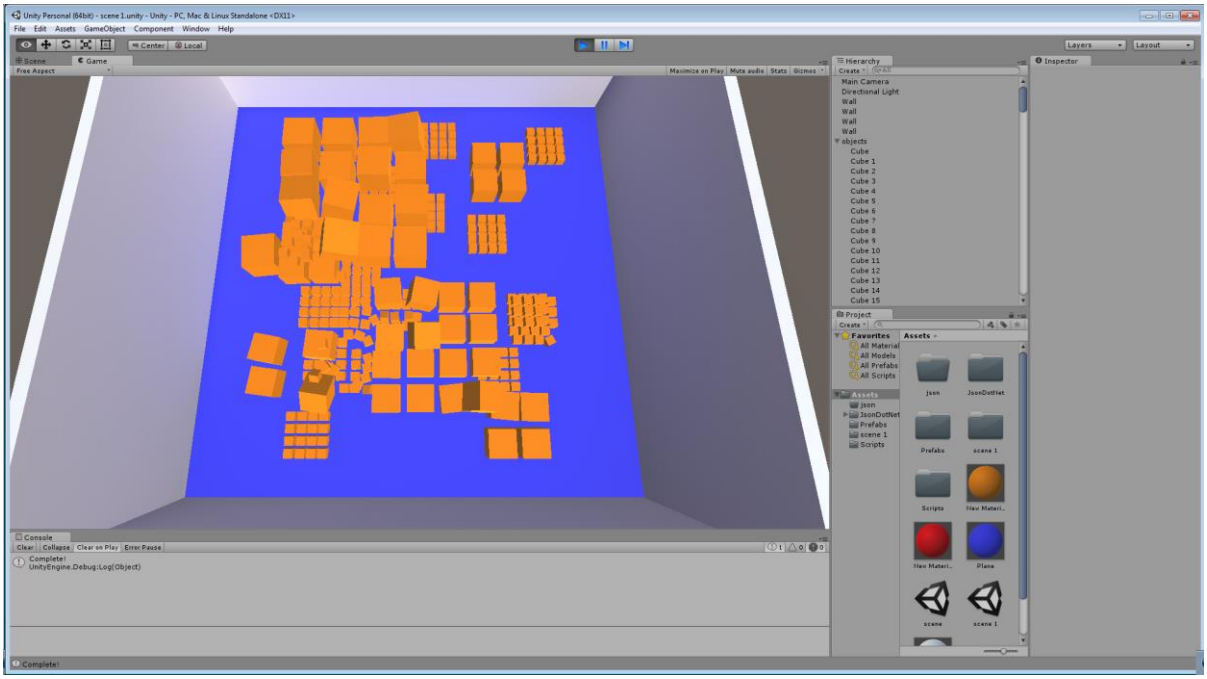
Figur 10 visar en experimentmiljö för pilotstudien. Miljön är fylld med rigida kroppar som påverkas av gravitation och kommer därför börja falla och kollidera med varandra när simulationen startas. För de 400 första frames som körs sparas varje objekts position. Figur 11 och Figur 12 visar en körning av simulationen. I den senare bilden har simulationen slutförts och den insamlade datan har skrivits till en fil. Datan är nu redo att importeras till C++-applikationen.



Figur 10 En experimentmiljö för en pilotstudie uppsatt i Unity, de orangea objekten är rigida kroppar som påverkas av fysik.



Figur 11 Simulationen startas och positionsdata spelas in för varje frame.



Figur 12 Simulationen är slutförd och den inspelade datan sparas till en textfil i jsonformat.

Nedan följer ett kort urklipp från filen som skapas under körningen. Startinfo-json-objektet innehåller info som används för att initiera objekten. Denna data består av 2 3DVektorer för varje objekt som håller dess startposition och storlek.

```
{
  "StartInfo": [
    {
      "position": {
        "x": 12.720000267028809,
        "y": 5.3299999237060547,
        "z": 15.859999656677246
      },
      "size": {
        "x": 1.0,
        "y": 1.0,
        "z": 1.0
      }
    },
```

…

I samma fil hålls också datan för varje frame. I denna del lagras endast positionen objekt har för den framen. Detta lagras i en 2darray.

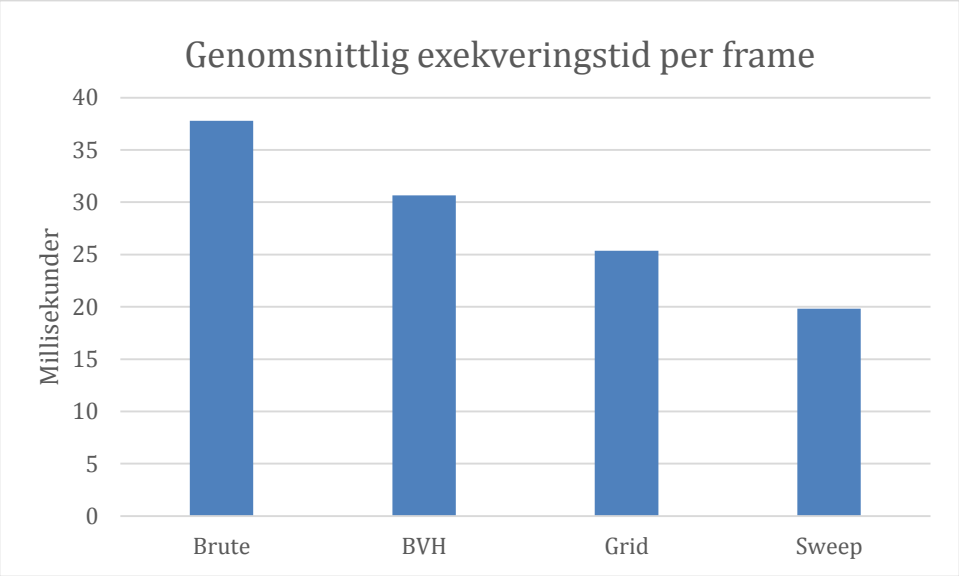
```
…
"FrameData": [
    [
      {
        "x": 12.720000267028809,
        "y": 5.3299999237060547,
        "z": 15.859999656677246
      },
      {
```

```
        "x": 13.970000267028809,
        "y": 5.3299999237060547,
        "z": 15.859999656677246
      },
      {
        "x": 16.430000305175781,
        "y": 5.3299999237060547,
        "z": 15.859999656677246
      },
...
```

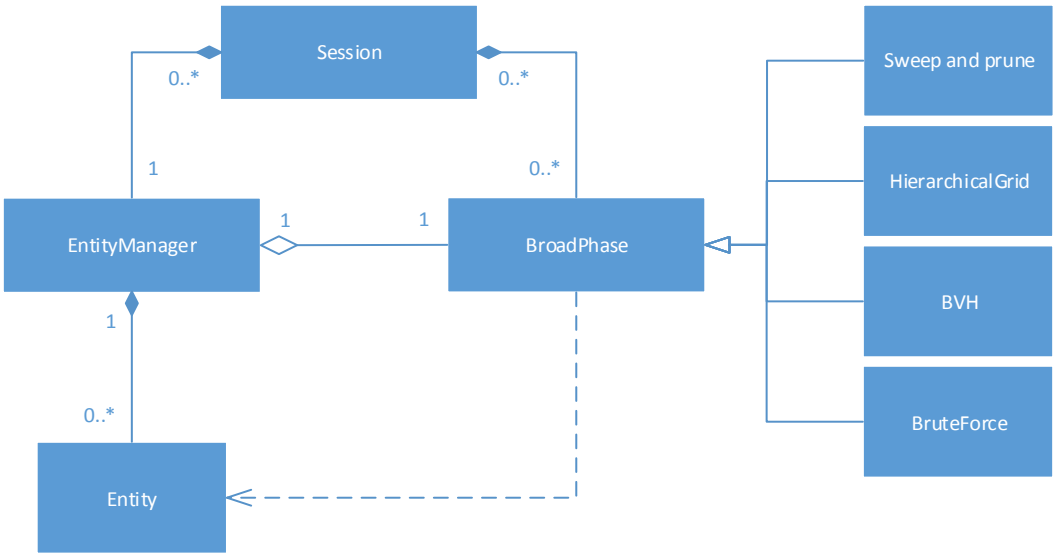
4.2 Mätningsprogrammet

Simulationsdatan som genereras av Unity-applikationen importeras sen till mätningsprogrammet som skapats. Denna data är uppdelad i två olika delar, den första delen innehåller all data som behövs för att initiera objekten. Den andra delen av datan innehåller positionerna för alla objekt för varje frame. Efter alla objekt är initierade, körs de olika bredfasalgoritmerna över datan från simulationen. Tiden som det tar för algoritmen att behandla datan mäts och sparas tillsammans med antalet potentiella kollisioner som hittades. Denna tid mäts med QueryPerformanceCounter som ger en högprecisionsklocka som kan göra exakta mätningar av exekveringstiden. När alla algoritmer har kört klart skrivs den insamlade datan för varje bredfasalgoritm till en xml-fil. Xml valdes som format till detta för att det enkelt skulle kunna gå att importera outputdatan till Excel för att kunna skapa grafer över den. Nedan följer ett kort exempel på hur outputdatan som genereras ser ut. Figur 13 visar en graf över outputdatan från en pilotstudie med 300 rörliga objekt.

```
...
<frame duration="38" collisions="88"/>
<frame duration="40" collisions="95"/>
<frame duration="44" collisions="116"/>
<frame duration="40" collisions="118"/>
...
```



Figur 13 Graf över data från en pilotstudie som visar medelexekveringstiden per frame.



Figur 14 Simplifierat klassdiagram över de mest intressanta delarna av mätningsprogrammet.

Figur 14 ovan visar en förenklad överblick över implementationen. Session hanterar inladdningen av datan och är den klassen som har hand om att köra alla bredfasalgoritmer och mäta exekveringstiden. Broadphase ger alla bredfasalgoritmerna ett gemensamt interface vilket tillåter Session att köra alla algoritmerna utan överflödig kod.

4.2.1 Sweep and Prune

Vid implementationen av Sweep and prune-algoritmen var planen från början att göra separata svep över varje enskild axel och för varje axel lagra de par som överlappar. När alla axlar blivit svepta skulle sen snittet av parmängderna ge de par som överlappar på alla axlar. Under utvecklingen av detta uppenbarade det sig att denna approach troligen skulle få väldigt dålig prestanda på grund av mängden minnesallokeringar som behöver ske varje frame och de dyra beräkningarna som skulle krävas för att få ut snittet av mängderna. En enklare lösning som var inspirerad av (Linneweber, 2011) implementerades istället. I detta blogginlägg föreslår han att istället för att lagra par för alla axlar, endast göra sökningen över en axel. Om en överlappning hittas görs ett snabbt test för de andra axlarna för att hitta överlappningarna. Algoritmen kör nu istället bara sina operationer över en axel. När en överlappning har hittas i 1D görs ett fullt 3d-intersektionstest. Detta skär drastiskt ner på antalet intersektionstest som behöver göras jämfört med en bruteforce-algoritm när objekten inte är samlade på samma axel.

4.2.2 Hierarchical grid

Hierarkiska grids kan implementeras på flera olika sätt för användning till bredfaskollision, dessa tas upp i bakgrundskapitlet. För den här implementationen lagras varje enskilt objekt endast i en cell istället för alla celler objektet täcker. Detta beslut togs eftersom det skulle kräva mycket mer insättningar och uttagningar när ett objekt flyttades om objektet lagrades i alla celler det täcker. Det här ledde till att implementationen av uppdateringssteget för algoritmen blev simpel men implementationen av kollisionsdetekteringen blev mycket mer komplicerad. Eftersom närliggande celler också behöver sökas igenom för att hitta alla kollisioner, istället för att bara ta de objekt som ligger i samma cell. Detta skapade en hel del problem med

dubbletter och missade kollisioner som tog lång tid att lösa. Ett annat beslut som togs här var att lagra gridet i en tredimensionell array istället för ett endimensionellt hashtable. Detta gjordes för att underlätta genomsökningen av närliggande celler. Den implementationen som först skapades baserades på psuedokod från (Schornbaum, 2009). Lösningen använde en itereringsstrategi som gick igenom varje cell i gridet och hittade kollisioner mellan de objekt som var i de närliggande cellerna. Denna implementation visades dock senare i utvecklingen att vara felaktig då den missade många av kollisionerna som skett och rapporterade flera dubbletter, vilket gjorde det svårt att märka av att algoritmen betedde sig felaktigt. Algoritmen skrevs om från grunden med en annan itereringsstrategi som istället itererade över varje objekt istället för cellerna i gridet.

4.2.3 Bounding volume hierarchy

Flera olika strategier för att bygga upp ett BVH existerar. I den här implementationen används top-down-approachen som beskrivs i (Anon., 2010). Strategier för att skapa kompaktare träd finns men denna valdes för dess snabba uppbyggnadstid och simpla implementation. Implementationen av algoritmen bygger upp trädet på nytt varje frame. En optimering som kan göras är att inte bygga om trädet varje frame utan istället bara uppdatera delar av trädet där objekt förflyttat sig. Detta skulle ge bättre prestanda när få objekt rör sig. Dock är den nuvarande uppbyggnadstiden kort och ger fortfarande en stor förbättring över bruteforce-algoritmen. Trädet används sen genom att iterera igenom alla objekt i scenen och för varje objekt göra ett rekursivt intersektionstest mot rotnoden som sedan gör intersektionstest mot dess barnnoder tills en lövnod är nådd.

5 Utvärdering

Detta kapitel presenterar resultatet av alla mätningar följt av en analys av dessa och tillslut en presentation av vilka slutsatser man kan dra av den insamlade datan.

Eftersom namnet på algoritmerna används flitigt i kapitlet har namnet Bruteforce förkortats till Brute, Sweep and Prune har förkortats till SaP, Hierarchical grid till HGrid och Bounding Volume Hierarchy till BVH.

5.1 Presentation av undersökning

I detta kapitel presenteras resultaten av undersökningen. Först presenteras en mätning av ett utvalt basfall som de andra mätningarna kan jämföras mot. Följande mätningar visar i stort sett samma miljö men med någon ändring mellan experimenten som till exempel ett annat antal objekt eller en annan placering av objekten i miljön. Alla mätningar utförs över 400 frames av simulationen. Resultaten av varje mätning visas med ett antal grafer som visar relevant data för den mätningen. Graferna visar exekveringstiden i millisekunder.

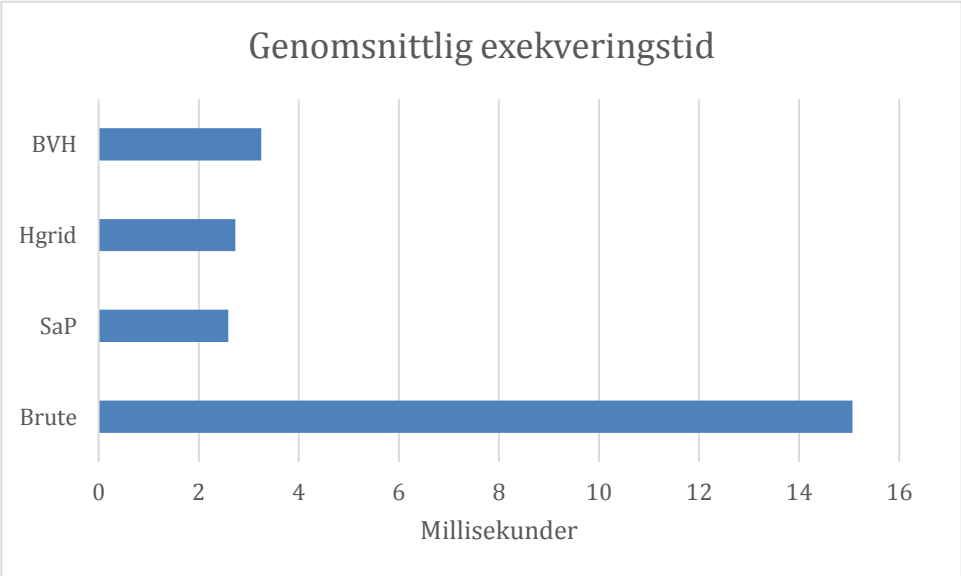
I bilderna för detta delkapitel representerar blåa kuber statiska objekt som behåller sin startposition under hela simulationen, orangea kuber är rörliga och förflyttas i en slumpad riktning hela simulationen.

Objekten i miljön har en slumpad storlek mellan 1 till 5 enheter. Miljön som simulationen körs i är begränsad till ett kubiskt område vars sidor är 100 enheter långa. Objekten kan inte förflytta sig utanför denna miljö.

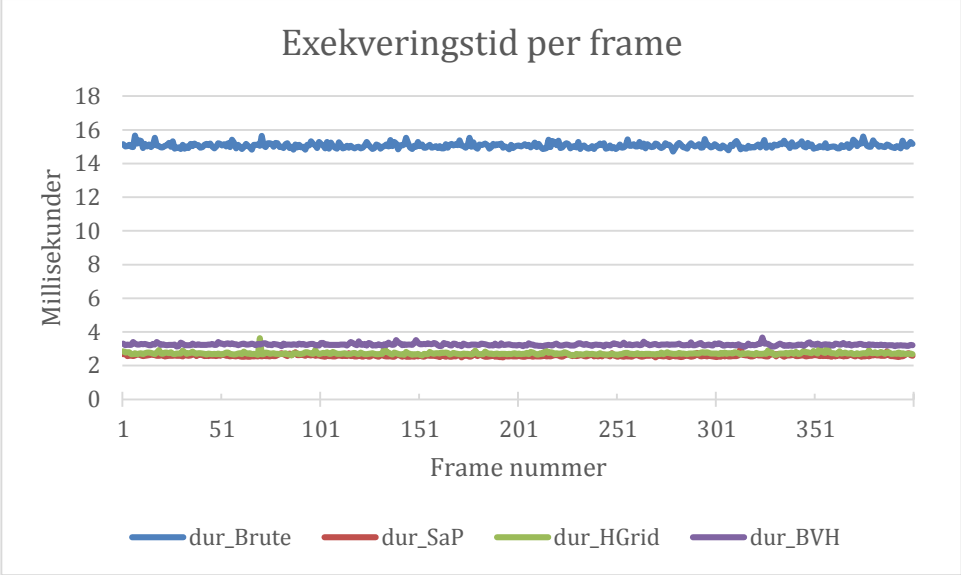
Alla mätningar har utförts på en dator utrustad med en Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz.

5.1.1 Standard mätning

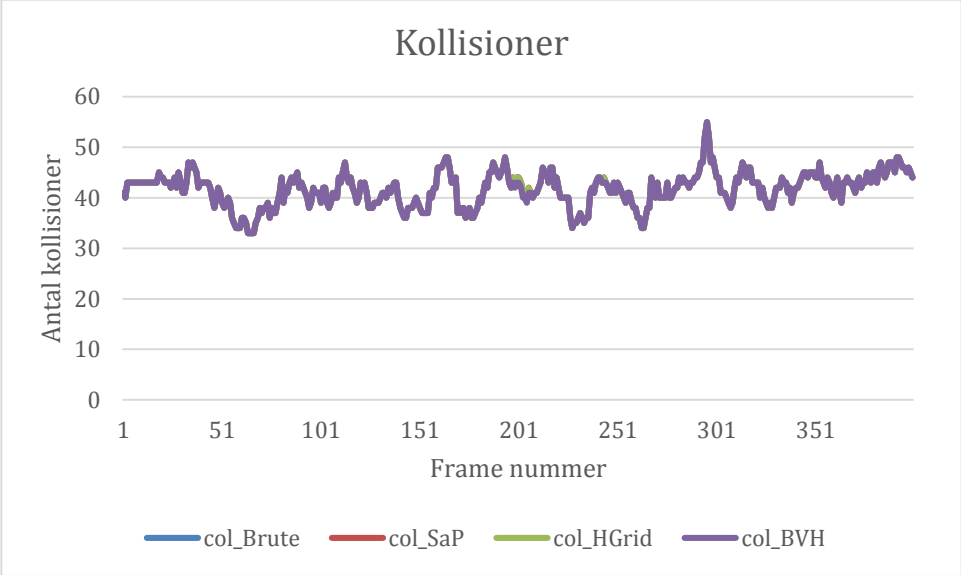
Som en baslinje för experimenten att jämföras mot utfördes en mätning med 1000 objekt varav 20 % var rörliga. Objektens startpositioner är jämnt distribuerade över miljön.



Figur 15 1000 stycken objekt.



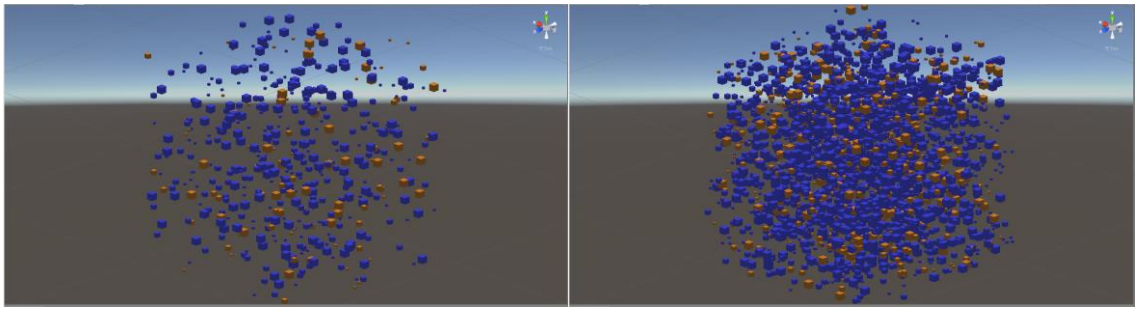
Figur 16 Exekveringstid per frame.



Figur 17 Kollisioner per frame.

5.1.2 Experiment 1: Varierande mängd objekt

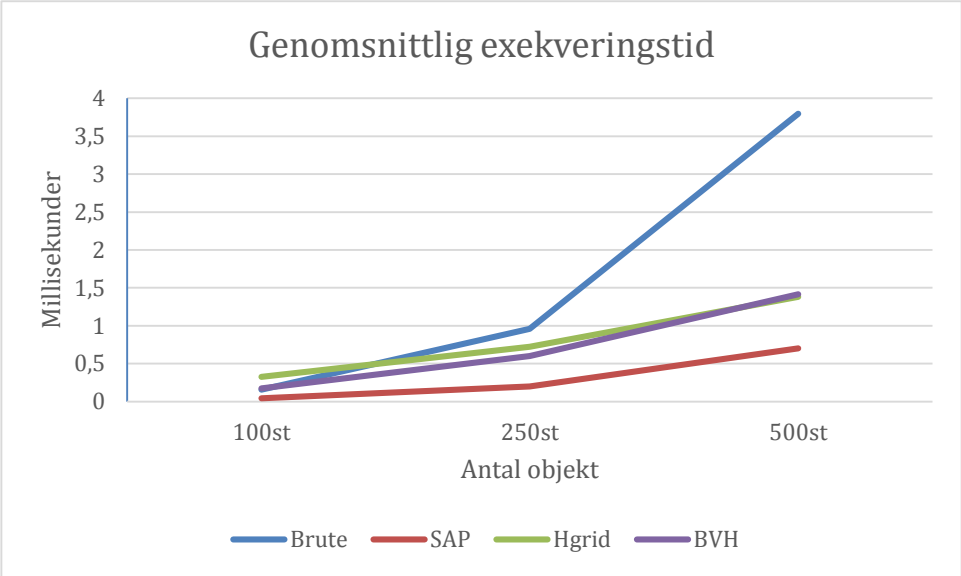
I detta experiment utfördes flera mätningar med en ökande mängd objekt. Mätningar utfördes med 100, 250, 500, 1000, 2500 och 5000 stycken objekt varav 20 % av dessa var rörliga. I Figur 18 visas två bilder över hur miljön som mätningen utfördes på ser ut. Objektens startpositioner är jämnt distribuerade i miljön.



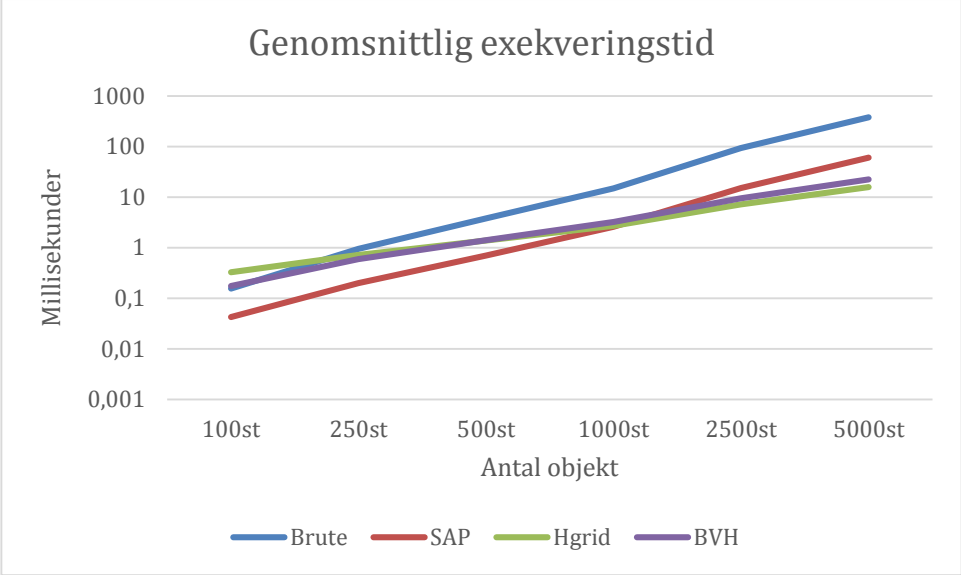
Figur 18 500 och 2500 objekt.

Grafen i Figur 19 visar mätningar av medelexekveringstiden som varje algoritm mätte i miljöer med 100, 250 och 500 stycken objekt. Den här grafen visar att exekveringstiden för Bruteforce algoritmen mäter ungefär samma exekveringstid som de andra vid 100 objekt men stiger kraftigt över de andra vid 250 och 500 objekt. HGrid och BVH har ungefär samma mättider. SaP ligger lång under de andra algoritmerna vid varje mätpunkt i denna graf.

I Figur 20 visas en graf med alla mätpunkter som gjordes. Grafen visar att SaP har lägst exekveringstid fram till 1000 objekt, efter den punkten stiger exekveringstiden till att vara över 3 gånger högre än HGrid och BVH vid 5000 objekt. BVH håller sig vid ungefär samma exekveringstid som HGrid fram till 5000 objekt där HGrid är ca 30 % snabbare.



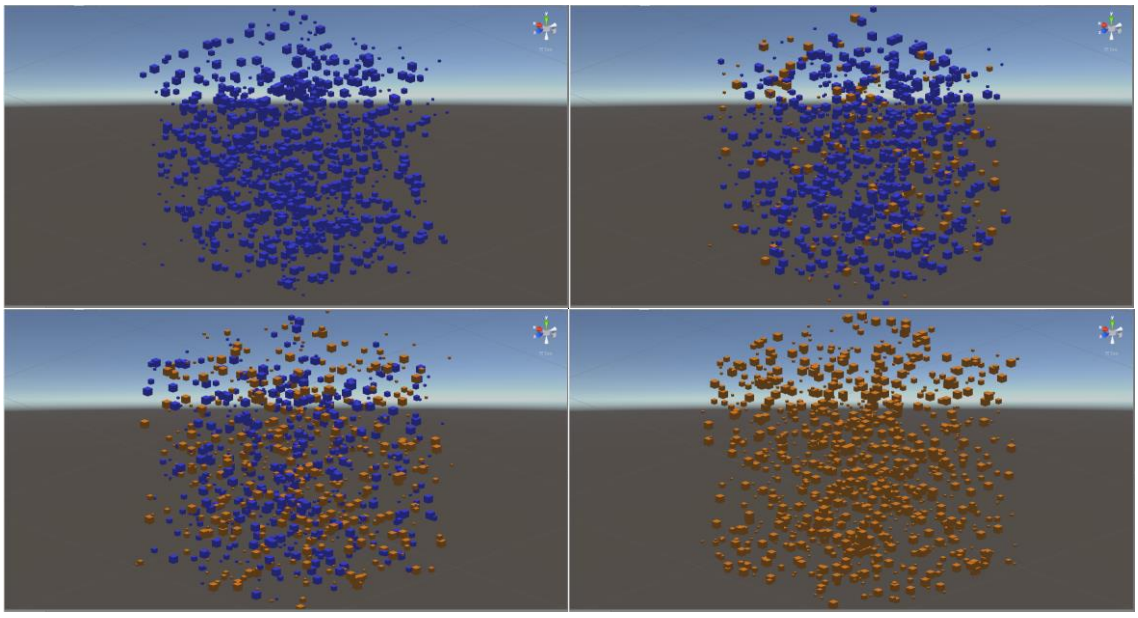
Figur 19 100 till 500 stycken objekt.



Figur 20 Mätningar mellan 100 till 5000 stycken objekt visat på en logaritmisk skala.

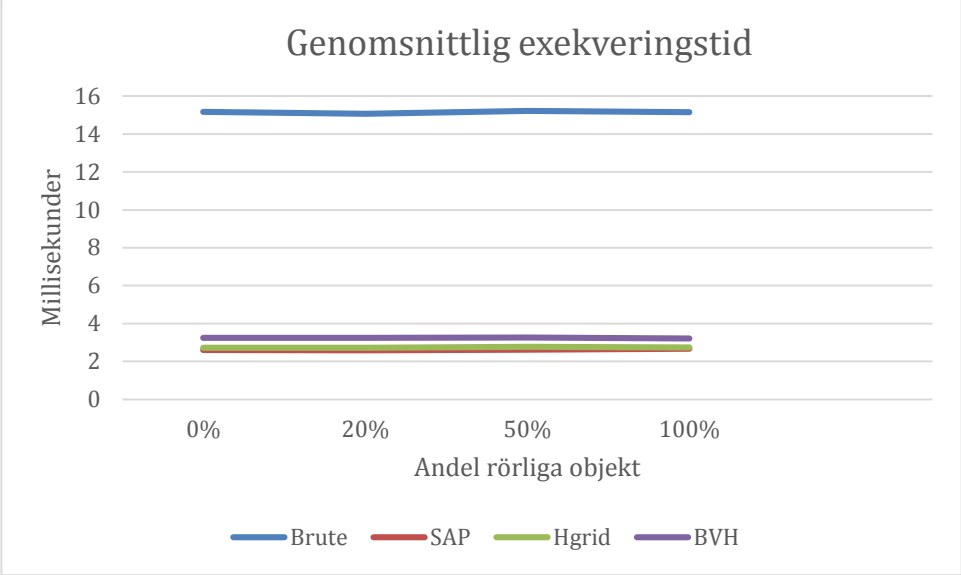
5.1.3 Experiment 2: Varierande mängd rörliga objekt

Detta experiment utfördes med 1000 stycken objekt med slumpade startpositioner i miljön. Fyra mätningar gjordes där andelen rörliga objekt sattes till 0 %, 20 %, 50 % samt 100 %. De fyra bilderna i Figur 21 visar hur miljöerna för mätningarna ser ut. Blåa objekt är statiska och förflyttas inte alls under hela simulationen. De orange objekten är ständigt i rörelse men kan inte förflyttas utanför miljöns begränsningar.



Figur 21 0%, 20%, 50% och 100% rörliga objekt.

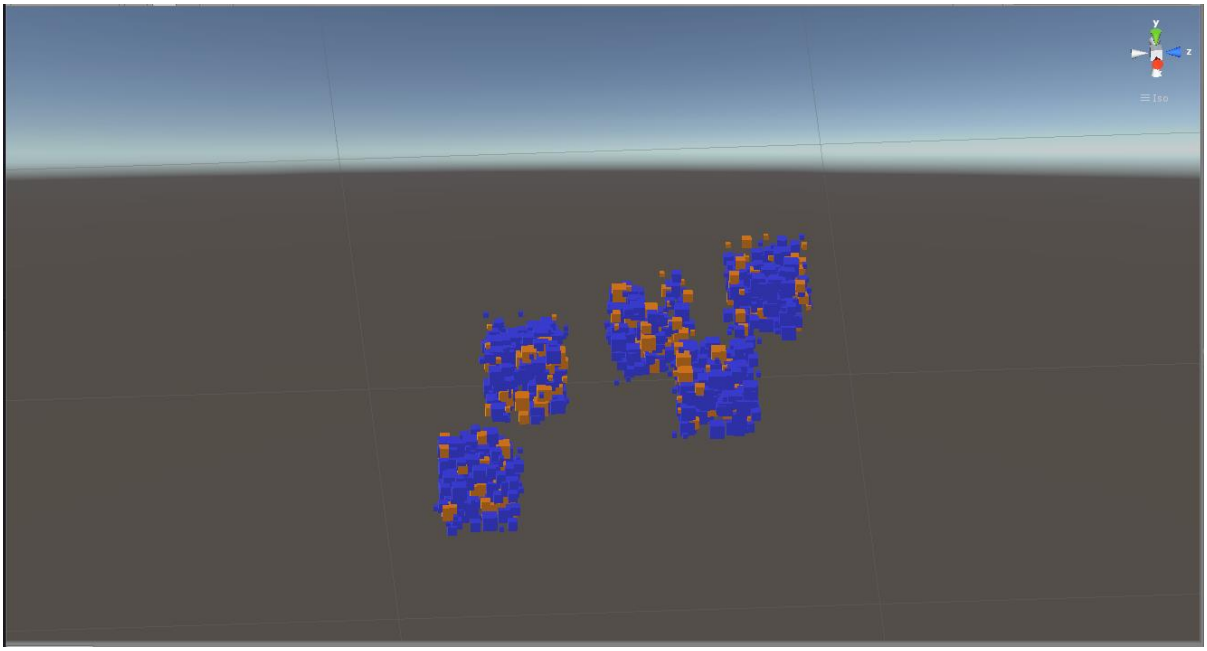
Grafen i Figur 22 visar att alla algoritmer i stort sett har samma mätvärden vid alla mätpunkter.



Figur 22 Varierad andel rörliga objekt.

5.1.4 Experiment 3: Kluster av objekt

I detta experiment slumpades 1000 objekt ut koncentrerat runt fem punkter i miljön. 20 % av dessa var rörliga och var inte bundna runt de punkter de skapades vid och spreds kort efter simulationsstarten ut i miljön. I Figur 25 utelämnas Bruteforce för att kunna visa de andra resultaten i högre upplösning.

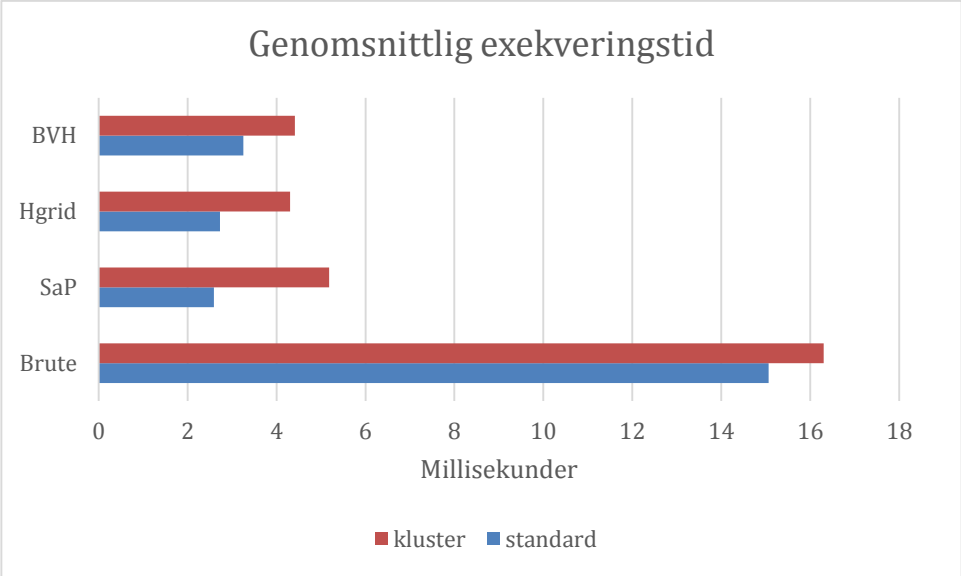


Figur 23 5 kluster med totalt 1000 objekt.

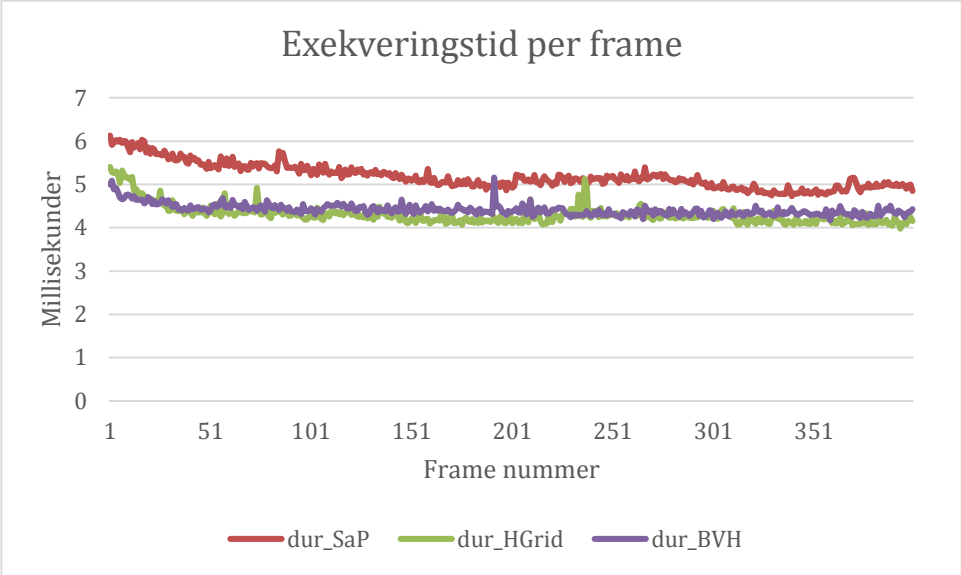
Grafen i Figur 24 visar medelexekveringstiden för denna mätning och standardmätningen som gjordes i 5.1. Alla algoritmer visar en markant ökning i tidsåtgången, värst drabbad är SaP som nästan har dubblerat sin exekveringstid jämfört med standardmätningen.

I Figur 25 visas exekveringstiden för varje frame. Alla algoritmer börjar på ett lite högre värde som sjunker en liten bit in i simulationen. Figur 26 visar antalet potentiella kollisioner som

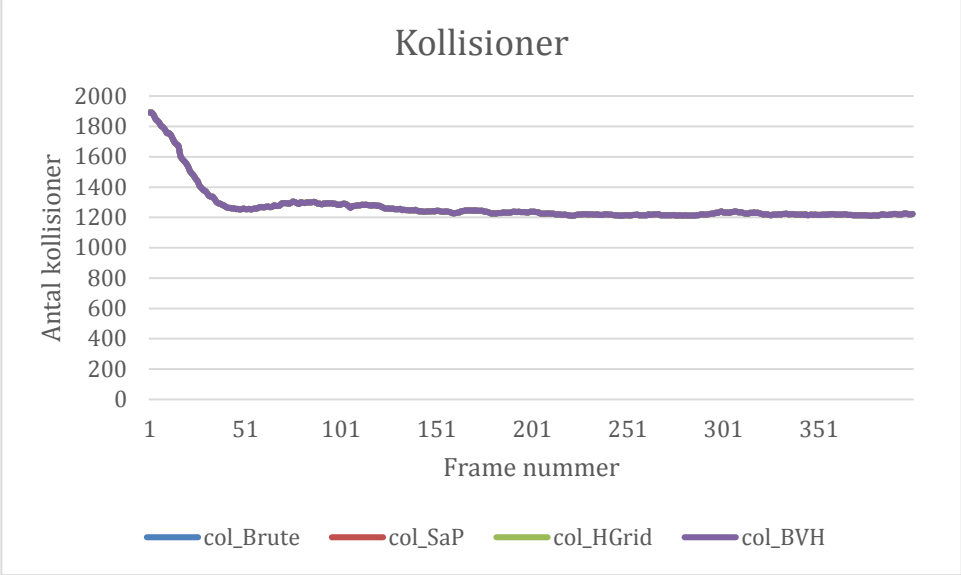
har skett under simulationen, alla algoritmer visar samma antal. Grafen visar att antalet kollisioner avtar snabbt i början av simulationen och jämnar där efter ut sig.



Figur 24 Medelexekveringstid med värden från standardmätningen att jämföra med.



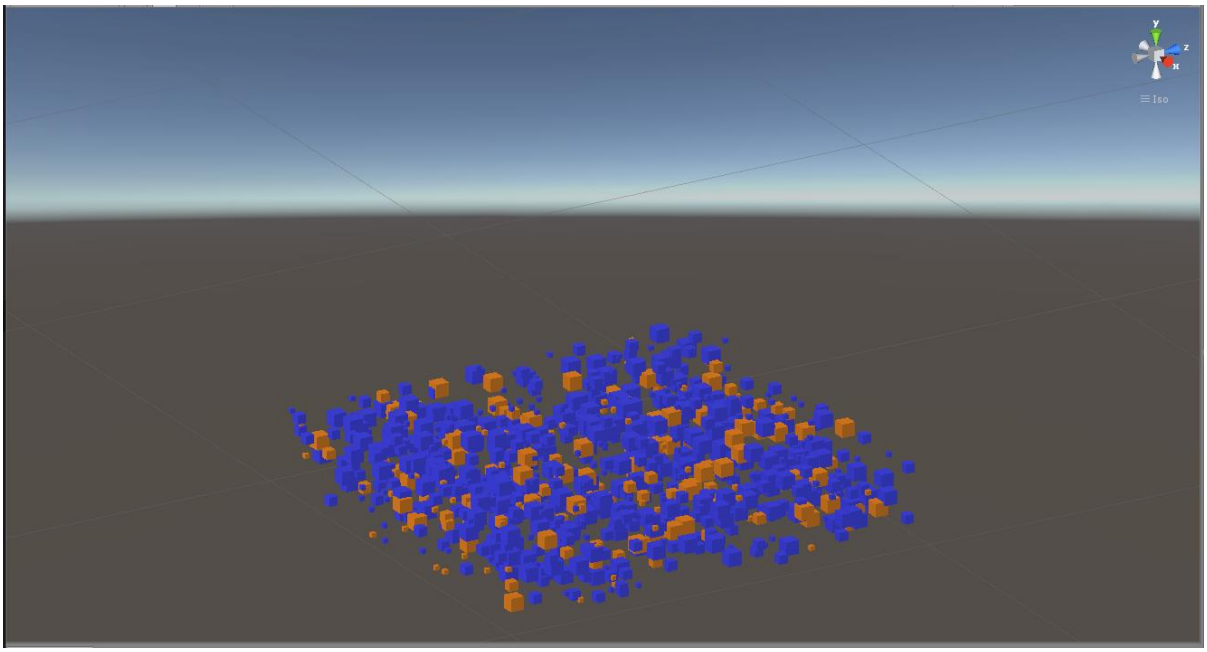
Figur 25 Exekveringstid för varje frame.



Figur 26 Antalet kollisioner för varje frame under simulationen.

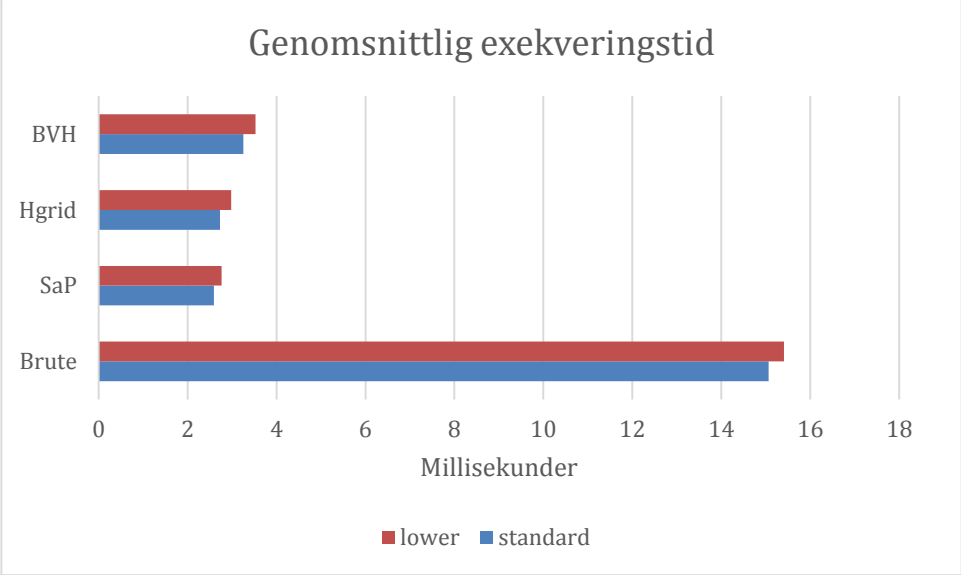
5.1.5 Experiment 4: Objekt koncentrerade runt botten av miljön.

I detta experiment distribuerades 1000 stycken objekt ut i de lägre 20 % av miljön. 20 % av dessa var rörliga och var inte bundna till startpositionerna och spreds kort efter simulationsstarten ut i miljön.

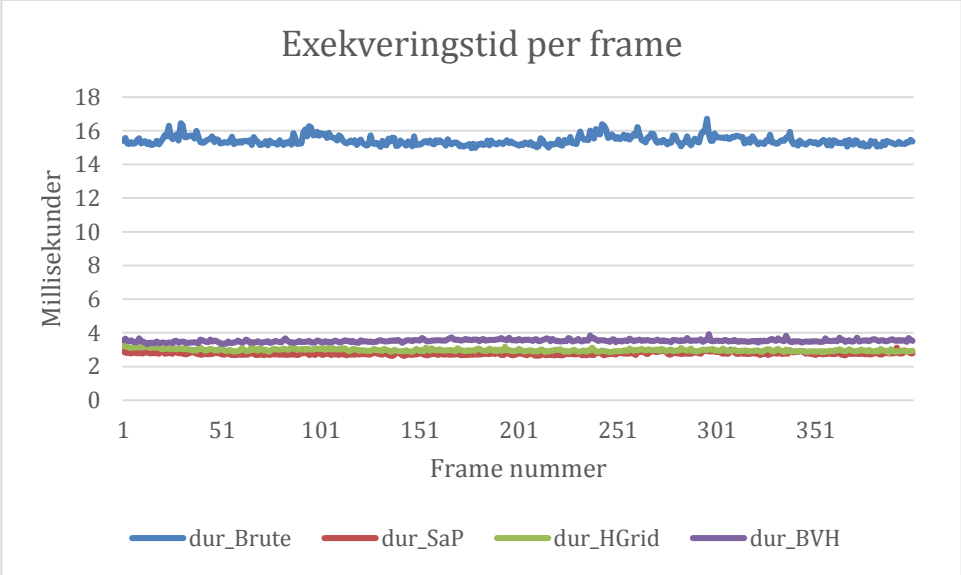


Figur 27 Objekten spridda över de lägre delarna I miljön.

Grafen i Figur 28 visar medelexekveringstiden jämfört med standardmätningen. Alla algoritmer visar en minimal ökning i tidsåtgången. Figur 29 visar att exekveringstiden för varje frame i stort sett är konstant under hela simulationen.



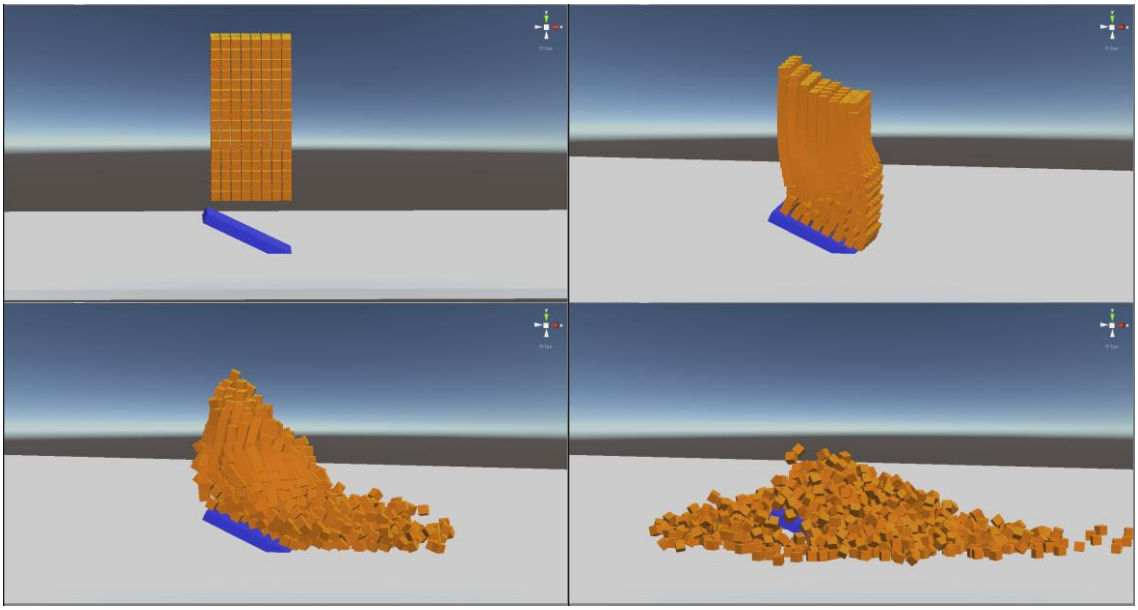
Figur 28 Medelexekveringstid med värden från standardmätningen att jämföra med.



Figur 29 Exekveringstid per frame

5.1.6 Experiment 5: Fallande torn

I detta experiment konstruerades ett torn av 1024 stycken objekt med rigida kroppar som påverkas av gravitation. Tornet är uppbyggt av kuber som sitter tätt placerade men är inte i kontakt med varandra i början av simulationen. Tornet är i början av simulationen en bit över ett vinklat bräde som det faller ner på när simulationen körs, som visas i Figur 30.

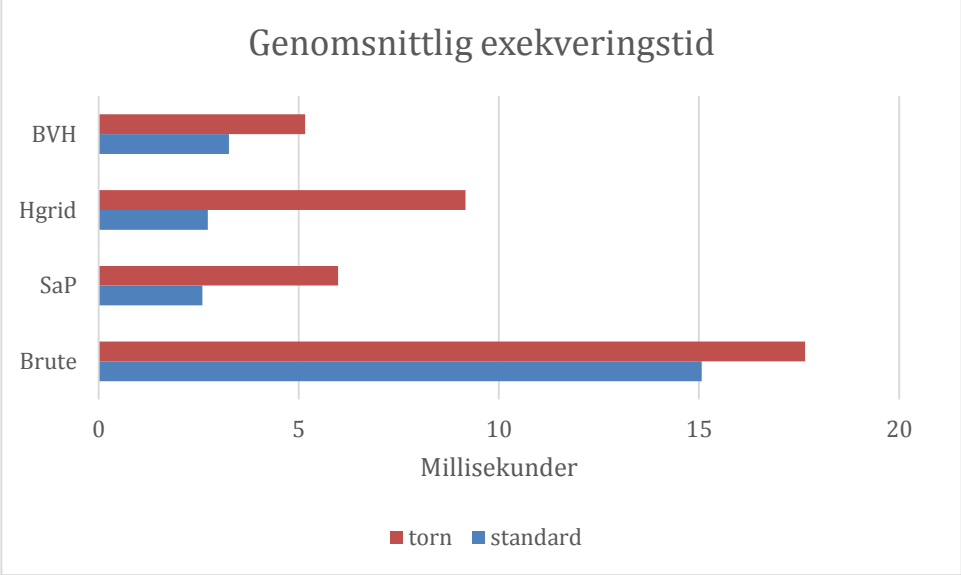


Figur 30 Tornet under olika tidpunkter I simulationen.

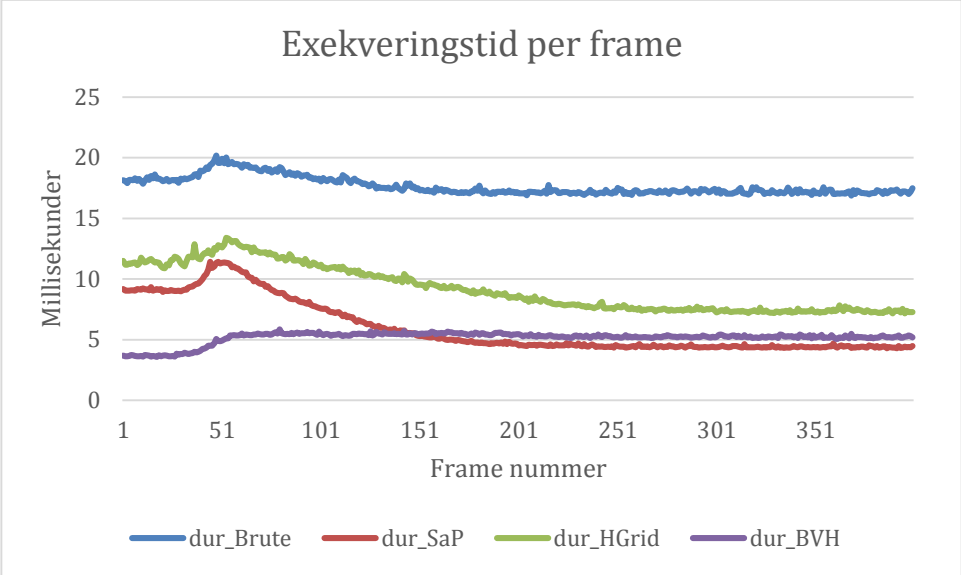
Grafen i Figur 31 visar medelexekveringstiden för algoritmerna i den här mätningen. Exekveringstiden för HGrid är över 3 gånger så hög som tiden för standardmätningen i det här experimentet. SaP har ungefär dubbelt så hög genomsnitts exekveringstid som standardmätningen och BVH visar minst ökning av alla algoritmer.

I Figur 32 visas exekveringstiden per frame för simulationen. HGrid och SaP börjar högt och går sedan upp lite till när alla kollisioner börjar ske, efter detta sjunker exekveringstiden allteftersom tills de tillslut planar ut efter halva simulationen. BVH håller sig ganska stabil igenom hela simulationen förutom att den går upp lite när objekten börjar kollidera.

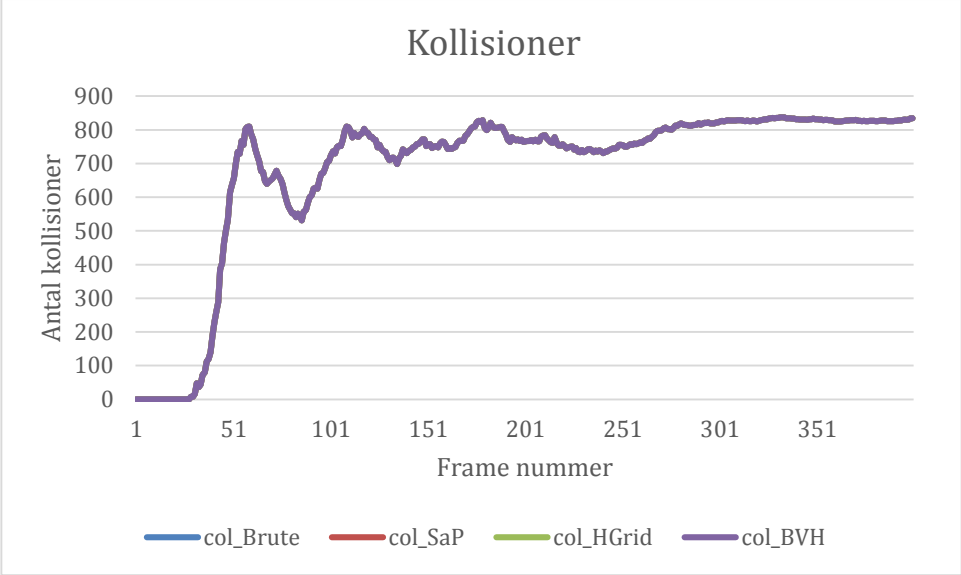
Figur 33 visar antalet kollisioner som sker under simulationen. I denna kan vi se att inga kollisioner sker i början förrän en stor spik i grafen sker och det går att se att objekten studsar och tar sig tillslut till viloläge.



Figur 31 Medelexekveringstid med värden från standardmätningen att jämföra med.



Figur 32 Exekveringstid för varje frame under simulationen.



Figur 33 Antalet funna potentiella kollisioner under simulationen.

5.2 Analys

I detta delkapitel analyseras resultaten från varje experiment i 5.1.

5.2.1 Experiment 1: Varierande mängd objekt

Som grafen i Figur 19 visar är bruteforce ett bra alternativ när det gäller färre än 250 stycken objekt, efter denna punkt vänder detta och får avsevärt mycket sämre prestanda än de andra lösningarna som grafen i Figur 20 visar. Detta beror på algoritmens O(n^2) skalning.

Sweep and prune hanterar låga mängder objekt mycket väl som syns i Figur 19 och är den enda av algoritmerna som tydligt presterar bättre än bruteforce i testet med 100 objekt. SaP skalar dock inte lika bra som BVH eller Hgrid vid högre mängder objekt, detta beror troligen på att SaP är dålig på att hantera när många objekt överlappar på en axel. Den ökade mängden objekt får många att överlappa på x-axeln till punkten att prestandan för algoritmen drabbas.

Hgrid och BVH skalar näst intill identiskt men vid mätningen på 5000 stycken objekt är Hgrid 30 % snabbare.

5.2.2 Experiment 2: Varierande mängd rörliga objekt

Resultaten av detta experiment var förvånande då hypotesen var att endast BVH och Brute skulle vara opåverkade av andelen rörliga objekt, på grund av att de inte lagrar någon data mellan varje frame. Dock visar grafen i Figur 21 även att HGrid och SaP inte påverkas märkbart av hur många objekt som rör sig i miljön. Detta är troligen att antalet cellbyten för HGrid eller omsorteringarna som sker för SaP sker sällan och snabbt nog för att det inte ska ge en märkbar effekt på prestandan. En annan version av SaP som beskrivs av (Baraff, 1992) som inte implementeras i denna studie påverkas starkt av hur många objekt som rör sig i miljön, denna version hade haft en mycket låg exekveringstid när inga objekt i miljön förflyttas.

5.2.3 Experiment 3: Kluster av objekt

Diagrammet i Figur 24 visar tydligt hur det här scenariot är mycket värre för samtliga algoritmer jämfört med standardmätningen. Detta beror på att objekten ligger mycket nära varandra, vilket gör det svårt för algoritmerna att eliminera kollisioner. BVH påverkas minst av detta scenario men håller sig rätt lika med HGrid i tidsåtgången. Exekveringstiden för SaP har ökat markant jämfört med standardmätningen, detta beror återigen på att fler objekt överlappar på x-axeln som algoritmen körs över.

Bruteforce algoritmen borde ge samma resultat som standardmätningen i alla fall där det är lika många objekt inblandade. Anledningen varför detta inte är fallet här beror troligen på hur intersektionstestet är implementerat. Intersektionstestet returnerar tidigare om objekten inte överlappar på en axel, vilket ger en stor prestandavinst eftersom den koden körs mycket av bruteforce algoritmen. Hade intersektionstestet inte varit implementerat på detta sätt hade bruteforce troligen haft mindre skillnad mellan de olika mätningarna där antalet objekt är detsamma.

Grafen i Figur 25 visar att prestandan för algoritmerna förbättras lite när simulationen körs och de rörliga objekten sprider ut sig i miljön.

5.2.4 Experiment 4: Objekt koncentrerade runt botten av miljön.

Detta experiment var tänkt att bättre representera miljöer som vanligtvis förekommer i spel eftersom de flesta spelvärldar har gravitation. Objekten brukar därför koncentreras nära marknivån i miljön. Resultaten i Figur 28 visar att alla algoritmer ger en i stort sett samma ökning i exekveringstiden. Ingen av algoritmerna hanterar denna typ av miljö bättre eller sämre än de andra.

5.2.5 Experiment 5: Fallande torn

Grafen över medelvärdena i Figur 31 visar att medelexekveringstiden för HGrid ökade kraftigast, följt av SaP och till sist BVH som påverkades minst. Anledningen varför HGrid visar en sådan stor ökning i exekveringstid är troligen att tornet endast täcker några få celler, vilket placerar en stor mängd av objekten i samma celler. SaP visar också en stor ökning i exekveringstiden, ett liknande problem som för HGrid uppstår eftersom en mycket stor del av objekten överlappar på x-axeln. Som Figur 32 visar förbättras exekveringstiden för både SaP och HGrid tydligt när objekten börjar spridas ut i miljön.

Exekveringstiden för BVH håller sig ganska stabil förutom de första 40 frames innan kollisionerna som syns i Figur 33 börjar ske. Alla algoritmer har en lägre exekveringstid upp till denna punkt. Anledningen varför BVH håller sig stabil är att den bryr sig mindre om hur utspridda objekten är i miljön än de andra två algoritmerna som får sämre prestanda om många objekt är på samma axel (SaP) eller i samma celler (HGrid).

5.3 Slutsatser

Bruteforce är ett acceptabelt alternativ om miljön kommer att innehålla under 250 stycken objekt. Efter den gränsen är det bättre att välja någon av de andra algoritmerna istället, eftersom det kan bli svårt att köra applikationen i realtid annars. Dock är det en algoritm som är mycket lätt att implementera och kan användas som placeholder i ett projekt till dess prestanda blir ett problem.

Denna variant av Sweep and prune presterar bra när det är en låg mängd objekt i miljön och är även lätt att implementera. Dock får den sämre prestanda än HGrid och BVH vid stora mängder objekt, eller om många överlappar på x-axeln. Andra variationer av algoritmen existerar som inte har samma svagheter som denna implementation.

Varianten av Sweep and prune som beskrivs av (Baraff, 1992) håller en lista med kollisioner som sker varje frame som uppdateras när saker förflyttas. Denna variant har en väldigt låg exekveringstid när objekten i miljön inte förflyttas, vilket är bra för spel i synnerhet eftersom den största delen av objekten som bygger upp en spelmiljö är statiska och det är få objekt som ständigt är i rörelse.

En annan av dessa varianter är Multi Box Pruning som bland annat används av Nvidia PhysX som kombinerar Sweep and Prune med ett grid genom att använda SaP inom cellerna. Med denna metod minskas antalet objekt som varje instans av SaP behöver hantera och förbättrar därför prestandan vid stora mängder objekt. En variant av denna presenteras även i (Tracy, et al., 2009) deras resultat visar en stor förbättring vid stora mängder objekt.

Hierarchical Grid var den bästa av algoritmerna för att hantera stora mängder objekt av de som undersöktes i den här studien. Dock sätter algoritmen en begräsning på hur stort området den används på är och kan kräva att parametrar justeras beroende på miljön för optimal prestanda. Algoritmen presterar även bäst om objekten är lite mer utspridda i miljön och får sämre prestanda om majoriteten av objekten ligger väldigt nära varandra som experimentet i 5.1.6 visar. Prestandan på HGrid skulle troligen kunna förbättras om statiska och rörliga objekt låg i olika HGrids för att förhindras att kollisioner genereras mellan statiska objekt.

Den största fördelen med Bounding Volume Hierarchy algoritmen är att det är den som påverkas minst av miljön. Hur objekten är placerade i miljön påverkar knappt prestandan till skillnad från HGrid och SaP. Denna egenskap ger algoritmen en väldigt jämn exekveringstid vilket är speciellt bra för spel, eftersom en stabil framerate är vitalt för att spelet ska kännas flytande. Prestandan på algoritmen är också mycket god, lite långsammare än HGrid vid 5000 objekt men ligger annars ganska jämnt med HGrid. Precis som HGrid skulle troligen prestandan för algoritmen förbättras om statiska och rörliga objekt lagrades i olika träd.

Ingen av algoritmerna som undersöktes påverkades märkbart av hur många objekt som förflyttas i miljön och alla algoritmer hittar samma mängd potentiella kollisioner.

För att besvara frågan om vilken av dessa algoritmer man ska implementera till spelmotorn man utvecklar beror valet mycket på miljön den ska användas till. Bounding Volume Hierarchy är ett solitt val för de flesta miljöer. Om mer prestanda behövs då det är mycket objekt i miljön kan Hierarchical grid vara ett bättre val förutsatt att objekten inte är grupperade nära varandra. Sweep and prune existerar som en lättimplementerad lösning för miljöer som innehåller färre objekt.

6 Avslutande diskussion

6.1 Sammanfattning

Detta arbete undersöker fyra olika algoritmer som används i bred kollisionsdetektering. De valda algoritmerna är Bruteforce, Sweep and Prune, Hierarchical grid och Bounding volume hierarchy, alla dessa löser problemet på olika sätt. De undersöktes i olika miljöer för att bedöma vilken algoritm som passar bäst för de olika miljötyperna.

Undersökningen gjordes genom att köra simulationer i en existerande spelmotor som sparade positionsdatan från varje objekt under simulationens gång. Denna positionsdata lästes sen in i den skapade artefakten som mätte tidsåtgången som det tog för varje algoritm att köra simulationen.

Resultaten visar att ingen av algoritmerna påverkas märkbart om objekten i miljön förflyttas under simulationen eller inte. Dock spelar mängden objekt, samt hur de är distribuerade över miljön en stor roll för algoritmernas prestanda. Resultaten visar också att Sweep and prune fick bäst prestanda vid låga mängder objekt men hade sämre prestanda än Hirachical grid vid stora mängder objekt. Båda dessa hade dock låg prestanda i de miljöer där många objekt var grupperade nära varandra. Bounding volume hierarchy påverkades minst av distributionen av objekten i miljöerna men hade annars medel resultat. På grund av sin stabilitet rekommenderas Bounding volume hierarchy algoritmen om man ska implementera en spelmotor/fysikmotor som ska klara av de flesta miljöer med en stabil framerate.

6.2 Diskussion

För att se till att alla resultat var pålitliga utfördes alla mätningar på samma dator under samma förutsättningar. Algoritmerna implementerades i sina simplaste former, det kan argumenteras för att dessa versioner inte representerar de valda algoritmerna fullt ut. Speciellt Sweep and prune algoritmen som implementeras skiljer sig från den vanligaste versionen av algoritmen. Den vanliga versionen av Sweep and prune har betydligt mycket bättre prestanda när få objekt förflyttas till bekostnad av sämre när det är en majoritet som är rörliga.

Annan forskning inom bred kollisionsdetektering visar spridda resultat över prestandan för algoritmerna. Resultaten från (de Sousa Rocha, et al., 2006) visar att deras version av Sweep and prune får 50fps vid 1000 objekt. Detta motsvarar ungefär 20 millisekunder vilket är betydligt sämre än resultaten i denna studie. Dock inkluderar den mätningen tiden det tar för allt att rendera. Mätningarna från (Tracy, et al., 2009) visar en mycket mer optimerad algoritm som klarar runt 80000 objekt på 20 millisekunder. Deras variant av Sweep and prune är många gånger mer effektiv än den som implementeras i denna undersökning.

Det går även att argumentera för att miljöerna som mätningarna utförs på inte representerar miljöer som algoritmerna i slutändan är tänka att användas på. Mer generella miljöer valdes för att lättare identifiera vad det är som påverkar prestandan men också för att det skulle ta för mycket tid att bygga upp nog många spelmiljöer för att få ut bra resultat. Liknande generella miljöer används också i flera andra studier som (Tracy, et al., 2009) och (de Sousa Rocha, et al., 2006).

Undersökningen som har bedrivits i detta arbete och dess resultat kan datorspelsbranschen dra nytta av. Denna bransch skapar många jobb i Sverige och andra länder och spel är även en stor exportvara som bidrar till ekonomin. Dock kan spel vara väldigt beroendeframkallande och precis som andra beroenden kan detta ha en negativ effekt på personens hälsa och välmående.

Kollisionsdetektering har även andra användningsområden utanför spel. Det kan användas för krocktest simulationer för att skapa säkrare bilar. Det kan användas till utbildning av bland annat piloter, brandmän eller kirurger i träningssimulationer för att hjälpa till att rädda liv.

6.3 Framtida arbete

Denna undersökning implementerar endast grundläggande versioner av algoritmerna. En logisk fortsättning på detta arbete vore att undersöka andra variationer av algoritmerna för användning i spelmiljöer. Några exempel för detta har tagits upp tidigare i rapporten. Bland dessa är Sweep and prune varianten som tas upp i (Baraff, 1992), eller hybriden mellan Sweep and prune och grid från (Tracy, et al., 2009). Det kan också vara intressant att testa olika implementationer av Octree-algoritmerna som inte undersöks i denna rapport.

Eftersom undersökningen endast utfördes på mer generella miljöer skulle det vara intressant att göra mätningarna på riktiga spelmiljöer. Detta för att se om resultaten skiljer sig åt från denna undersökning och eventuellt visa nya nackdelar eller fördelar med algoritmerna, samt få mer specifik data över hur algoritmerna presterar i mer typiska miljöer.

En separat studie skulle även kunna undersöka effekten som valet av gränsvolym har på algoritmernas prestanda.

En annan fortsättning på detta arbete vore att jämföra olika algoritmer för den smala kollisionsfasen för användning i spel, som komplement till denna undersökning.

Resultaten från detta arbete kan användas av företag eller enskilda utvecklare vid implementation av en spelmotor för att välja en algoritm som är bäst passad för den miljön spelet kommer att innehålla.

7 Referenser

Anon., 2010. *Will Code For Coconuts*. [Online] Available at: https://coocoscode.wordpress.com/2010/12/15/building-bounding-volume-hiearchies/
[Använd 09 04 2015].

Avril, Q., Gouranton, V. & Arnaldi, B., 2011. *Dynamic adaptation of broad phase collision detection algorithms*. Singapore, IEEE, pp. 41-47.

Bandi, S. & Thalmann, D., 1995. An Adaptive Spatial Subdivision of the Object Space for Fast Collision Detection of Animated Rigid Bodies. *Computer Graphics Forum,* 14(3), pp. 259-270.

Baraff, D., 1992. *Dynamic Simulation of Non-Penetrating Rigid Bodies. PhD thesis, Cornell University*. s.l.:s.n.

Catto, E., 2007. *Box2D*. u.o.:www.box2d.org.

Cohen, J. D., Lin, M. C., Manocha, D. & Ponamgi, M., 1995. *I-COLLIDE: An interactive and exact collision detection system for large-scale environments*.. Monterey, ACM, pp. 189-ff.

Coumans, E., 2013. *Bullet Physics Library*. u.o.:www.bulletphysics.org.

Culley, R. & Kempf, K., 1986. *A collision detection algorithm based on velocity and distance bounds*. Santa Clara, IEEE, pp. 1064-1069.

de Sousa Rocha, R., Rodrigues, M. & da Silva Taddeo, L., 2006. *Performance Evaluation of a Hybrid Algorithm for Collision Detection in Crowded Interactive Environments*. Manaus, IEEE, pp. 86 - 93.

Eitz, M. & Lixu, G., 2007. *Hierarchical Spatial Hashing for Real-time Collision Detection*. Lyon, IEEE, pp. 61-70.

Gottschalk, S., Lin, M. C. & Manocha, D., 1996. *OBBTree: A hierarchical structure for rapid interference detection*.. New York, ACM, pp. 171-180.

Hastings, E. J., Mesit, J. & Guha, R. K., 2005. *Optimization of large-scale, real-time simulations by spatial hashing*.. Hilton, SCS, pp. 9-17.

Havok, 2011. *Havok Technology Suite*. u.o.:Intel.

Hubbard, P., 1995. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics,* 1(3), pp. 218-230.

Jiménez, P., Thomas, F. & Torras, C., 2001. 3D collision detection: a survey. *Computers & Graphics,* 25(2), pp. 269-285.

Kockara, S. o.a., 2007. *Collision detection: A survey*. Montreal, IEEE, pp. 4046-4051.

Linneweber, T., 2011. *jitter-physics.com*. [Online] Available at: http://jitter-physics.com/wordpress/?tag=sweep-and-prune
[Använd 08 05 2015].

Ming, L. & Gottschalk, S., 1998. *Collision detection between geometric models: A survey*.. u.o., IMA, pp. 602-608.

Overmars, M., 1992. Point location in fat subdivisions. *Information Processing Letters,* 44(5), pp. 261-265.

Schornbaum, F., 2009. *Hierarchical Hash Grids for Coarse Collision Detection,* Erlangen: University of Erlangen-Nuremberg.

Terdiman, P., 2007. *Sweep-and-prune*. [Online] Available at: http://www.codercorner.com/SAP.pdf [Använd 09 04 2015].

Tracy, D., Buss, S. & Woods, B., 2009. *Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal*. Lafayette, IEEE, pp. 191 - 198.

Xiao-rong, W., Meng, W. & Chun-gui, L., 2009. *Research on Collision Detection Algorithm Based on AABB*. Tianjin, IEEE, pp. 422-424.

Xing, Y.-S., Liu, X. P. & Xu, S.-P., 2010. *Efficient collision detection based on AABB trees and sort algorithm*. Xiamen, IEEE, pp. 328-332.

Zhao, S. o.a., 2013. *A fast spatial partition method in bounding volume hierarchy*. Beijing, IEEE, pp. 15-18.