

Bred kollisionsdetektering för spel

Brodad phase collisions detection for games

Examensarbete inom huvudområdet Informationsteknologi

Grundnivå 30 högskolepoäng

Vårtermin 2020

Franz Jonzon

Handledare: Mikael Johannesson

Examinator: Sanny Syberfeldt

Sammanfattning

[Direkt efter titelsidan ska det finnas en sammanfattning, som omfattar högst 150 ord. Denna sida ingår inte i ordinarie sidnumrering. Sammanfattningen ska i kort form sammanfatta hela arbetet inklusive bakgrund, problemformulering, genomförande/implementation, resultat och framtida arbete. Den ska därmed vara det sista som skrivs i rapporten. Den behöver därför endast vara med i den rapport som lämnas in för opponering (och den slutgiltiga rapporten). I tidigare inlämningar kan denna text lämnas oförändrad.]

**Nyckelord:** [minst tre nyckelord men inte allt för många, helst inte fler än sex stycken]

Innehållsförteckning

[1 Introduktion 3](#_Toc35422840)

[2 Bakgrund 4](#_Toc35422841)

[2.1 Typer av kollisioner 4](#_Toc35422842)

[2.2 Typer av kollisionsdetektion 5](#_Toc35422843)

[2.3 Gränsvolymer 6](#_Toc35422844)

[2.4 Bredkollisionsdetektion 8](#_Toc35422845)

[2.4.1 Naiva lösningen (råstyrka) 9](#_Toc35422846)

[2.4.2 SAP 10](#_Toc35422847)

[2.4.3 Rumsuppdelning 11](#_Toc35422848)

[2.4.4 BVH 13](#_Toc35422849)

[2.4.5 Anpassningsbara algoritmer 15](#_Toc35422850)

[2.5 small kollisionsdetektion 16](#_Toc35422851)

[2.6 Broadmark 16](#_Toc35422852)

[3 Problemformulering 21](#_Toc35422853)

[3.1 Metodbeskrivning 22](#_Toc35422854)

[3.1.1 Metoddiskussion 24](#_Toc35422855)

[3.1.2 Vad som ska implementeras 25](#_Toc35422856)

[4 Genomförande/Implementation/ Projektbeskrivning 26](#_Toc35422857)

[4.1 Research / Förstudie 26](#_Toc35422858)

[4.2 Progressionsexempel: modellering 26](#_Toc35422859)

[5 Utvärdering 28](#_Toc35422860)

[5.1 Presentation av undersökning 28](#_Toc35422861)

[5.2 Analys 28](#_Toc35422862)

[5.3 Slutsatser 28](#_Toc35422863)

[6 Avslutande diskussion 29](#_Toc35422864)

[6.1 Sammanfattning 29](#_Toc35422865)

[6.2 Diskussion 29](#_Toc35422866)

[6.3 Framtida arbete 29](#_Toc35422867)

[Referenser 30](#_Toc35422868)

# Introduktion

[Introduktionen ska innehålla en sammanfattning av bakgrund, problem, metod och genomförande/implementation. Inför varje inlämning behöver bara de relevanta delarna av introduktionen lämnas in. Tänk på att lämna plats i introduktionen för stycket om genomförande vid första inlämningen. Introduktionen bör revideras vid varje inlämning eftersom arbetet förändras över tid.

För att separera innehållet i introduktionen ska den indelas i onumrerade stycken med hjälp av radbrytningar (i likhet med dokumentmallen). Styckena ska inte vara kortare än 4-5 rader text och innehålla ett sammanhållet ämne - till exempel bakgrund om datorspel.

Introduktionen ska *inte* ha numrerade underkapitel eller figurer och vara ungefär en sida lång (inte under en halv sida och inte mycket mer än en sida).

Introduktionen ska ha sidnummer 1.]

# Bakgrund

Kollisionsdetektion är i grunden ett enkelt problem: detektera när två eller flera objekt överlappar varandra. Detta problem preciserar Ericson (2004, s. 1), det handlar om att för två objekt avgöra om de överlappar, var de överlappar och när de började överlappa. Trots problemets enkla frågeställning är det ett område som ådragit sig markant med intresse från flertalet forskningsområden: fysiksimuleringar, datorspel, robotik, digitaltestning, motorsimuleringar, klädsimuleringar, operationssimuleringar och datorgrafik (Ericson 2004, s. 1; Zou, Liu, Yang, Li & Cheng 2017, s. 1765; Teschner m.fl. 2005, s. 61), bara för att nämna några. Det har även visat sig vara ett svårlöst problem och det är fortfarande en av de största flaskhalsarna för systemen de används i (Wang, Tang, Manocha & Tong 2018a, s. 227; Serpa & Rodrigues 2019b, s. 1). Interaktiva applikationer förklarar Friston & Steed (2019, s. 2611) har särskilt problem med denna flaskhals. Tiden interaktiva applikationer kan allokera till hela kollisionshanteringen är nämligen mycket begränsad. Exempelvis, som Ericson (2004, s. 1) förklarar, datorspel (datorspel refererar till alla former av digitala spel) uppdaterar skärmen med mellan trettio till sextio bilder per sekund vilket markant begränsar tiden under vilken spelet kan uppdatera logik, grafik m.m. Kollisionsdetekteringen är en vital del av spel, vilket Ericson (2004, s. 1) utvecklar. Det kan, i kombination med den strama tidsbudgeten, göra att kollisionsdetekteringen kan uppta en majoritet av tiden det tar att uppdatera en bild. Dålig kollisionsdetektering poängterar Ericson (2004, s. 1) kan enkelt bli den värsta flaskhalsen i ett spel.

För att effektivisera kollisionsdetekteringen brukar den delas upp i två faser, bred kollisionsdetektering och smal kollisionsdetektering. Den breda kollisionsdetekteringen lokaliserar alla potentiella kollisioner med hjälp av enklare gränsvolymer. Den smala kollisionsdetekteringen undersöker de potentiella kollisionerna som lokaliserats i den breda fasen och sorterar ut kollisionerna som inträffat, genom att jämföra objektens modeller alternativt mer exakta gränsvolymer (Ericson 2004, ss. 14-15; Luque, Comba & Freitas 2005, s. 179; Serpa & Rodrigues 2019b, s.1). Vissa, som Schmidtke & Erleben (2018, s. 3049), väljer även att implementera en tredje mittenfas.

## Typer av kollisioner

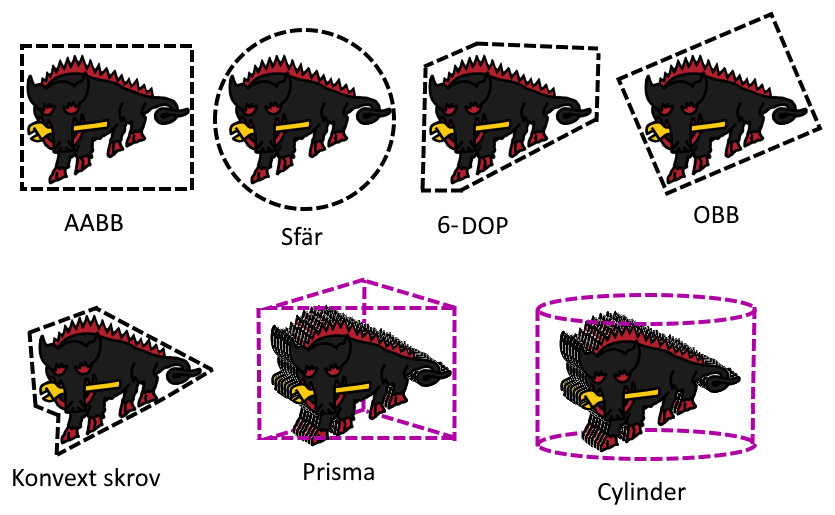
Som Zou m.fl. (2017, s.1767) beskriver brukar kollisionsdetekteringstekniker grovt delas upp i två typer, rigida kroppar och mjuka kroppar (deformerbara kroppar). De vidareutvecklar med att förklara att mjuka kroppar är tyngre och långsammare att beräkna. Detta till följd av att strukturer för de mjuka objekten måste uppdateras under kollisionens gång. Teschner m.fl. (2005, s. 62) förklarar även att en skillnad mellan typerna är att för mjuka kroppar måste självkollision hanteras. Detta är kollision som exempelvis hindrar en tygbit att gå igenom sig själv. Självkollision är något som Teschner m.fl. (2005, s. 62) förklarar brukar utelämnas för rigida kroppar. Det kan noteras att i datorspel brukar majoriteten av objekten i spelmiljön bestå av rigida kroppar (Serpa & Rodrigues 2019a, s.261; Teschner m.fl. 2005, s. 61).

## Typer av kollisionsdetektion

Li, Ding, Hong, Pan & Liu (2018, s.75574) förklarar att kontinuerlig kollisionsdetektering (KKD)[eng: CCD] används när mer exakt kollisionsdetektering önskas som i operationssimuleringar. De redogör att digitala applikationer uppdateras i diskreta intervaller. Därmed poängterar de att det alltid finns en risk att kollisioner missas. KKD förklarar Li m.fl. (2018, s.75574) undviker detta genom att beräkna vägen mellan de diskreta kontrollpunkterna, och därigenom lokalisera den exakta punkten där överlappningen inträffade. KKD:s lösningars förmåga att förhindra att objekt klipper in i varandra gör dem som Tian, Hu & Shen (2019, s.2) noterar, väl lämpade att hantera självkollision. Det gör dem dock beräkningstunga. Att KKD är beräkningstunga är något som Fukuhara, Tsujita, Sase, Konno, Jiang, Abiko & Uchiyama (2014, s. 4) instämmer med, De går vidare med att förklara att till följd av dess beräkningstyngd är inte KKD-lösningar lämpade för storskaliga interaktiva scenarion med många kollisioner. Li m.fl. (2018, s.75574) utvecklar att för mindre till medelstora scenarion är det genom att utnyttja accelerationsstrukturer möjligt att uppnå interaktiv hastighet. Det går att bl.a. utnyttja den parallella beräkningsförmågan i moderna beräkningsprocessorer och grafikprocessorer. Exempelvis utnyttjar Du, Zhao, Pan, & Wang (2015) grafikprocessorer för en monteringssimulering och Tian m.fl. (2019) använder en parallell hybridlösning som utnyttjar både beräkningsprocessorn och grafikprocessorn för en hjärndeformationssimulator. Båda lösningarna uppnår interaktiv prestanda samtidigt som de använder KKD.

KKD är dock inte lämpad för applikationer som prioriterar interaktivt högre än kollisionernas korrekthet, exempelvis: datorstödd design [eng: CAD], virtuell verklighet[eng: VR] och vägplanering. För applikationer som dessa redogör Weller, Debowski, & Zachmann (2017, s. 131) att diskret kollisionsdetektering (DKD) används vanligtvis. Fukuhara m.fl. (2014, ss. 2-4) förklarar att DKD-algoritmer endast söker efter kollisioner i diskreta intervaller, utan att ta hänsyn till rörelsen mellan uppdateringarna. Dessa egenskaper utvecklar Fukuhara m.fl. (2014, ss. 2-4) gör att beräkningskostnaden för DKD är avsevärt längre än den för KKD, dock finns risken att kollisioner missas. Om objekt rör sig tillräckligt fort kan det rent av passera rakt igenom andra objekt till följd av att de passerade varandra mellan intervallerna. Detta är ett problem som Ericson (2004, s. 17) förklarar kallas tunnelgrävning[eng: tunneling].

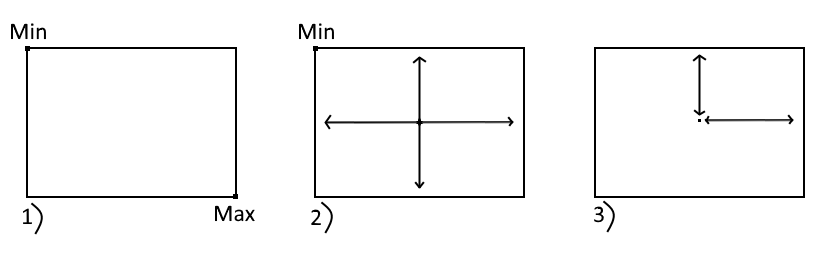
## Gränsvolymer



1. Flertalet olika gränsvolymer har utvecklats, här visas ett axplock av dem.

Den naiva metoden för att kontrollera om två objekt kolliderar är genom att jämföra objektens geometri. Detta är dock beräkningsmässigt dyrt att göra Ericson (2004, s.18). En vanlig metod för att snabba på kollisionsdetekteringen är gränsvolymer, vilket är approximationer av objekten med enklare geometriska figurer (Luque m.fl. 2005, s.180; Friston & Steed 2019, s.2612). Gränsvolymerna omsluter objekt och används i den breda kollisionsdetekteringen istället för objektens geometri för att sålla ut ickekollisioner och lokalisera kollisionspar. Logiken är att om två objekts gränsvolymer inte överlappar finns det ingen möjlighet att objekten kolliderar (Serpa & Rodrigues 2019a, s.261). Genom åren har flertalet olika gränsvolymer presenterats, **figur 1**, och valet av vilken av dessa som ska användas är ett av de viktigaste besluten i relation till kollisionsdetekteringen. Det har nämligen en stor inverkan på hur den kommer att implementeras (Serpa & Rodrigues 2019a, s.260).

Vilken gränsvolym som används förklarar Ericson (2004, ss. 76-77), beror på vilka egenskap som önskas av den och kollisionsdetekteringen. Han redogör att egenskaper som kan önskas är: tät passform, billigt att kontrollera överlappning, billig att beräkna, enkel att transformera och rotera samt litet minnesavtryck. Ericson (2004, ss. 76-77) konstaterar dock att dessa egenskaper ofta motverkar varandra. En gränsvolym som ger tätaste möjliga passform kommer resultera i att andelen falska kollisioner (se **figur 3**) minskar. Den kommer dock vara dyrare att beräkna, transformera och rotera samt ha ett större minnesavtryck.



1. De tre vanligaste representationerna av AABB:er, (1) min-max värden, (2) min-bredd värden och (3) center-radie värden.

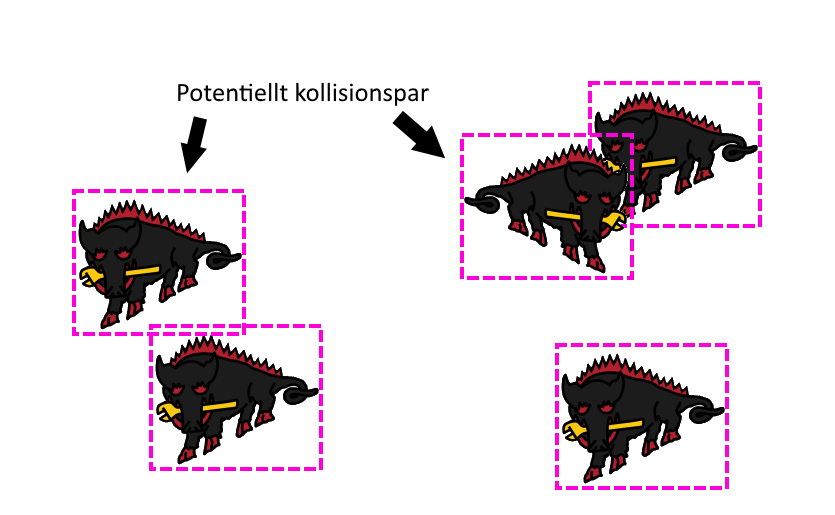
AABB (axelställd gränsvolym[eng: axis-aligned bounding box]) är en av de vanligaste gränsvolymerna. Den består av en rektangel vilken är konstruerad så dess normaler är parallella med givna koordinatsystemets världsaxlar. Fördelar med denna avgränsningsvolym är att den har ett relativt litet minnesavtryck. Detta kan dock variera något då det finns olika sätt en AABB kan representeras på (se **figur 2**). Den mest attraktiva egenskapen för AABB:er är dess triviala och snabba överlappningskontroll vilket görs genom att direkt jämföra koordinatvärdena. Nackdelen med AABB:er är att om objektet roteras så måste AABB:en beräknas om för att försäkra att objektet innesluts (Ericson 2004, ss. 77-86).

OBB (objektorienterad gränsvolym [eng: oriented bound box]) liknar AABB i att båda är rektanglar men medan en AABB är roterad efter koordinatsystemets axlar så är enOBB roterad efter objektet. Det är möjligt att få mycket tätare passform med en OBB än med AABB. Dock är det krångligare att beräkna och överlappningstesta samt är dess minnesavtryck mycket större (Ericson 2004, ss. 101-112).

En närapå lika populär gränsvolym som AABB är sfären, vilken är den mest minneseffektiva gränsvolymen då den endast behöver lagra sfärens centrum och radie. En annan fördel är att den är oberoende av objektens rotation, vilket gör den trivial att transformera. Överlappningskontrollen för sfärer är likvärdig med den för AABB, Troligen är den till och med något snabbare. Det är dock mycket svårare att beräkna minsta möjliga sfärvolym jämfört med att beräkna minsta AABB   
(Ericson 2004, s. 88).

Utöver de tre gränsvolymer som redogjorts för här så finns det även sfäriskt skal, konvext skrov samt trädstrukturer som låd träd (Teschner m.fl. 2005, s. 63).

## Bredkollisionsdetektion



1. Exempel på en scen med fem grisar varav två kolliderar med varandra. Två kolliderar potentiellt och en är inte fristående. De rosa rektanglarna är grisarnas gränsvolymer (i detta fall AABB). Alla grisar vars gränsvolymer överlappar noteras som kollisionspar och kommer att vidarebefordras till den smala kollisionsdetekteringen. Notera att de två till vänster är vad som kallas en falsk kollision. Detta då den inte kolliderar men deras gränsvolymer överlappar.

Den breda kollisionsdetekteringen är den första detekteringen som körs och den har till uppgift att lokalisera alla potentiella kollisionspar och att sålla ut övriga objekt. Detta görs för att slippa använda de tyngre mer noggranna kollisionsdetekteringsalgoritmerna i onödan (Wang m.fl. 2018, s. 227). Luque m.fl (2005, s.179) understryker att det är av största vikt att algoritmen som används för bredkollisionsdetektering kan snabbt och effektivt utföra utsållningen. De förklarar att därför används gränsvolymer, se **figur 1**, för att utföra kollisionstesterna snarare än objektens exakta geometri. Som Serpa & Rodrigues (2019a, s. 261) tar upp resulterar inte användandet av enkla geometriska approximationer endast i kortare beräkningstider. De gör även att den breda kollisionsdetekteringen i stort blir obunden av vad det är som kolliderar. De exemplifierar detta med att implementationen av den breda kollisionsdetekteringen för rigida kroppar och mjuka kroppar inte skiljer sig åt nämnvärt.

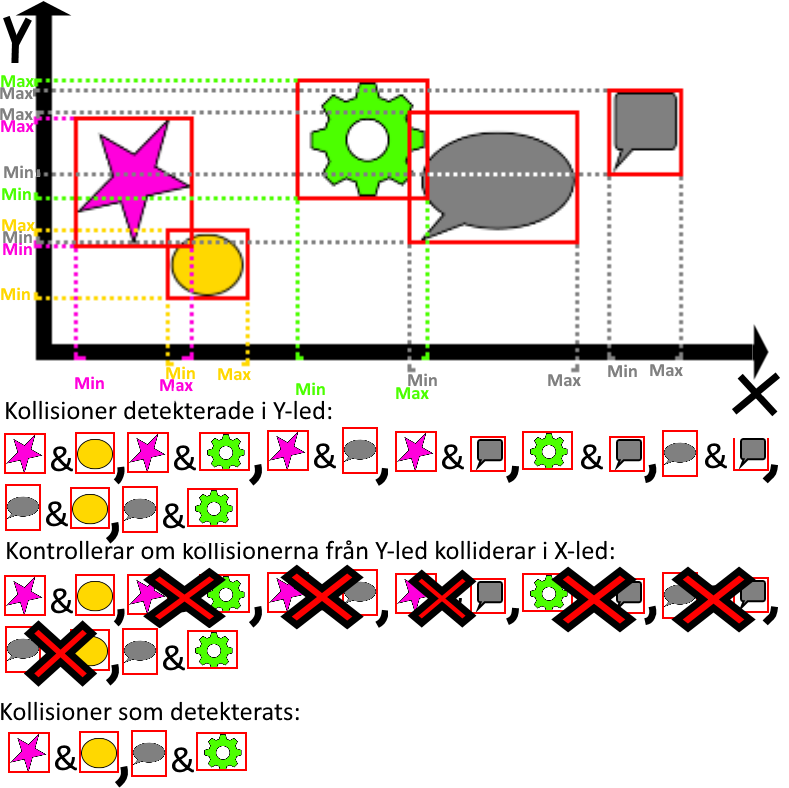
Exempel på nyttan med den breda kollisionsdetektering kan ses i **figur 3** i vilken den begränsar antalet tester som behöver göras med de tyngre detektionsalgoritmerna till två stycken. Med en naiv lösning hade alla objekt behövt testas mot alla övriga objekt med de tyngre algoritmerna.

Under åren har flertalet olika algoritmer presenterats för bredkollisionsdetekteringen och det finns en stor variation på hur dessa är utformade. Wang m.fl. (2018, s. 227) konstaterar att det grovt går att dela upp alla lösningar i två kategorier, de som använder rumsuppdelning och de som använder objektgruppering. Wang m.fl. (2018, s. 227) förklarar att rumsuppdelning innefattar lösningar som delar upp världen i delområden, och de nämner: KD-träd, rutnätsrumshierarkilistor[eng: hierarchical spatial hashing tables] och dylikt som exempel på algoritmer som faller inom rumsuppdelningskategorin. Den andra kategorin objektgruppering innefattar lösningar som grupperar objekten i scenen utifrån deras relation till varandra. De förklarar att den mest framträdande lösningen inom denna grupp är BVH (gränsvolymshierarki [eng: Bounding volume hierarchy]) vilket är en trädstruktur där noderna består av gränsvolymer som innesluter en delmängd av scenens objekt (se **figur 6**).

### Naiva lösningen (råstyrka)

Den absolut enklaste algoritmen för bredkollisionsdetektering är *råstyrka*[eng: Brute force] vilken är att testa alla möjliga kollisionspar mot varandra. Detta förklarar Serpa & Rodrigues (2019a, s.227) ger denna lösning den trevliga egenskapen att dess utföringstid endast är beroende av antalet objekt som ska testas. De utvecklar dock att den i praktiken i stort sett är oanvändbar då dess komplexitet är kvadratisk, vilket innebär att för *n* objekt måste *n*2 tester utföras. Serpa & Rodrigues (2019b, s. 3) redogör dock att tack vare dess avsaknad av hjälpstrukturer och kringarbeten är denna naiva lösning den mest effektiva kollisionsdetekteringsalgoritm för mindre mängder objekt. Geleri, Tosun & Topcuoglu (2013) bevisar även att en kraftigt optimerad grafikprocessor-baserad-råstyrkelösning kan tack vare grafikprocessorns parallella beräkningskraft vara effektiv för upp till medelstora scenarion. De noterar dock att den jämfört med andra tekniker skalar sämre gentemot större arbetsbördor. Serpa & Rodrigues (2019b, s.3) gör gällande att det inte finns några algoritmer uteslutande baserad på råstyrka inom utvecklingsfronten för bredkollisionsdetektering till följd av dess dåliga komplexitet. De noterar dock att den nyttjas som operator i grafikprocessorns baserade lösningar. Exempelvis Avril, Gouranton & Arnaldi (2014) presenterar en råstyrka och SAP(svepa och beskär [eng: Sweep And Prune])-algoritm vilken använder grafikprocessorns parallella beräkningskapacitet. Dock konstaterar Capannini & Larsson (2018, s.2076) i sina testningar så är inte heller denna lämpad för större scenario.

### SAP



1. Exempel på användning av SAP. Först lokaliserats alla potentiella kollisioner i Y-led. Sedan kontrolleras de potentiella kollisioner i X-led. Om det är det så är det en potentiell kollision som kommer att vidarebefordras till den smala kollisionsdetektionen.

SAP-tekniker bygger på att bryta upp den breda kollisionsdetekteringen till flertalet endimensionella tester gentemot världskoordinatsystemets koordinataxlar. Detektionen görs genom att först kollapsa ner alla objekt på en av koordinataxlarna och lokalisera de par som överlappar varandra. Därefter kontrolleras det om dessa par även överlappar de övriga koordinataxlarna, se **figur 4** till exempel. För att effektivisera de endimensionella sökningarna görs dessa genom att sortera objekten i intervaller utefter deras extrempunkter på den nuvarande koordinataxeln. Efter det kontrolleras vilka objekt som ligger mellan varandras maximum- och minimumpunkter. SAP-lösningar är bäst lämpade för upp till medelstora scenarion (Serpa & Rodrigues 2019b, ss. 3, 11).

Serpa & Rodrigues (2019b, s.3) tar upp att det i utvecklingsfronten finns två olika kategorier av algoritmer som använder sig av SAP, parallella versioner av SAP eller algoritmer som inkorporerar SAP som en operator. Exempel på den första är den anpassningsbara, parallella dubbel-SAP som Capannini & Larsson (2018) presenterade. Deras algoritm använder en dubbelaxelstrategi, vilket Capannini & Larsson (2018, s.2066) förklarar reducerar antalet tester som måste göras genom separera axlarna i minde intervaller och testar två axlar mot varandra samtidig. Detta utvecklar Capannini & Larsson (2018, s.2066) gör att varje objekt endast måste testas mot de objekt som är inom samma intervall. Capannini & Larsson (2018, s.2066) trycker dock på att det som särskiljer deras lösning gentemot andras är att det är en parallell beräkningsprocessorbaserad lösning vilken anpassas efter arbetsbördan, vilket de framhäver kan hantera komplexa scenarion med miljontals objekt.

En lösning där SAP används som operator är den som Serpa & Rodrigues (2019a) presenterar, vilket är ett *KD-träd* och SAP-hybrid som beroende på arbetsbördan varierar mellan att utföra inkrementell och komplett kollisionsdetektering. De använder en SIMD (en instruktion med flertalet data [eng: Single instruction multiple data]) optimerad SAP-operation för att motverka KD-trädets svagheter, vilket de utvecklar är att uppdaterandet av KD-trädet bildar en flaskhals om noderna endast innehåller ett objekt. Därav sorterar de KD-trädet något grundare och använder SAP-operation för att noggrannare sålla ut ickekollisioner ur lövnoderna, se **figur 5**.

Ett alternativ till SAP är inkrementell SAP (ISAP) vilken istället för att kontrollera objekten mot en axlarna sekventiellt, kontrollerar den istället axlarna var för sig för att sedan göra par av de objekt som kolliderar på alla axlar. ISAP sorterar in objekt i en lista för varje axel och utnyttjar det faktum att objekt inte brukar röra sig speciellt långt mellan två på varandra följande uppdateringar. Därav kan listorna hanteras som närstan sorterade vilket gör att istället för att bygga listorna från grunden så räcker det med att uppdatera dem. Utnyttjandet av information från tidigare uppdateringar gör att ISAP är optimal för större scenarion i vilka en stor andel av objekten är statiska. Det gör dock även att ISAP är mycket olämpligt för dynamiska scenarion där det uppstår stora förändringar mellan uppdateringar. Faktumet är att för dynamiska scenarion så är ISAP generellt tre gånger sämre än vanliga SAP-lösningar (Serpa & Rodrigues 2019b, s.3).

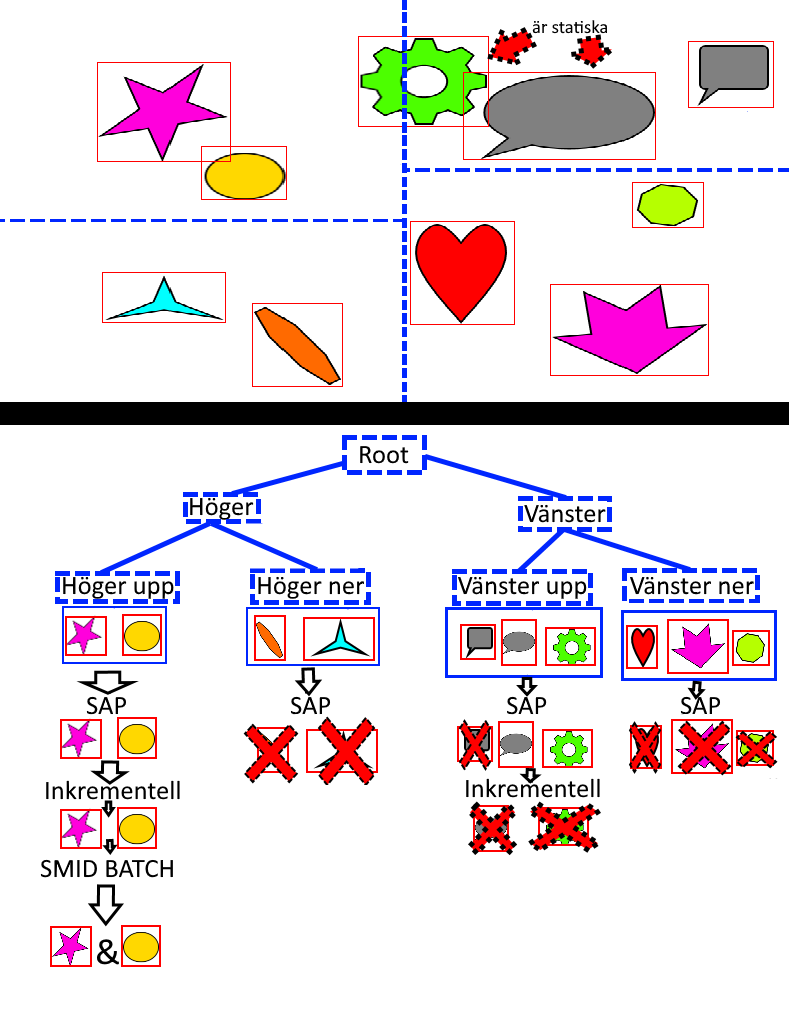
### Rumsuppdelning

Här följer en redogörelse av en handfull algoritmer vilka delar upp miljön i delområden (Friston & Steed 2019, s.2612). Objekten i miljön sorteras in i dessa delområden utefter deras position i kombination med deras gränsvolym (Serpa & Rodrigues 2019b, s. 4). Efter att objekt sorterats in i delområden testas varje objekt för kollision mot alla objekt inom samma delområde (Serpa & Rodrigues 2019b, s. 4). Detta är grunden för hur alla rumsuppdelningsalgoritmerna fungerar.

Den enklaste av de rumsuppdelningsalgoritmerna är *rutnätsalgoritmer* (på engelska känd som grid eller uniform grid) vilka delar upp miljön i ett rutnät Ericson (2004, s. 1). Serpa & Rodrigues (2019b, s.4) tar upp att denna algoritmtyp är både enkel att förstå sig på och att implementera. De tar även upp rutnätsalgoritmers exekvering. Dessa egenskaper, utvecklar de, gör att rutnätsalgoritmer är väl lämpade för att parallellisering och Serpa & Rodrigues (2019b, s.4) poängterar att de är mycket populära inom det området. Slutligen tar Serpa & Rodrigues (2019b, s.4) upp att det är mycket enkelt att utnyttja andra bredkollisionsdetekteringsalgoritmer som operatorer i rutnätsalgoritmer.

Ett annat tillvägagångssätt för uppdelningen är *rumsuppdelningsträd*[eng: space partitioning trees] vilka rekursivt delar upp miljön i mindre och mindre regioner, vilka används för att bygga en trädstruktur. Det finns en handfull olika algoritmer med varierande grader av flexibilitet för uppdelningen av miljön. Det ska noteras att ofta är mer flexibla algoritmer mer beräkningstunga än de mindre flexibla algoritmerna. Uppdelningen försöker ofta att dela upp miljön så att andelen objekt är lika många på båda sidorna av delningslinjen. För att åstadkomma detta är en vanlig metod att använda median eller medelvärdet av alla objektens positioner (Serpa & Rodrigues 2019b, s. 5).

Något som skiljer rumsuppdelningsträd från vanlig rumsuppdelning är att den vanliga rumsindelningen är objektoberoende, medan rumsuppdelningsträd oftast är beroende av objekten för sin uppdelning (Teschner m.fl. 2005, s. 72).

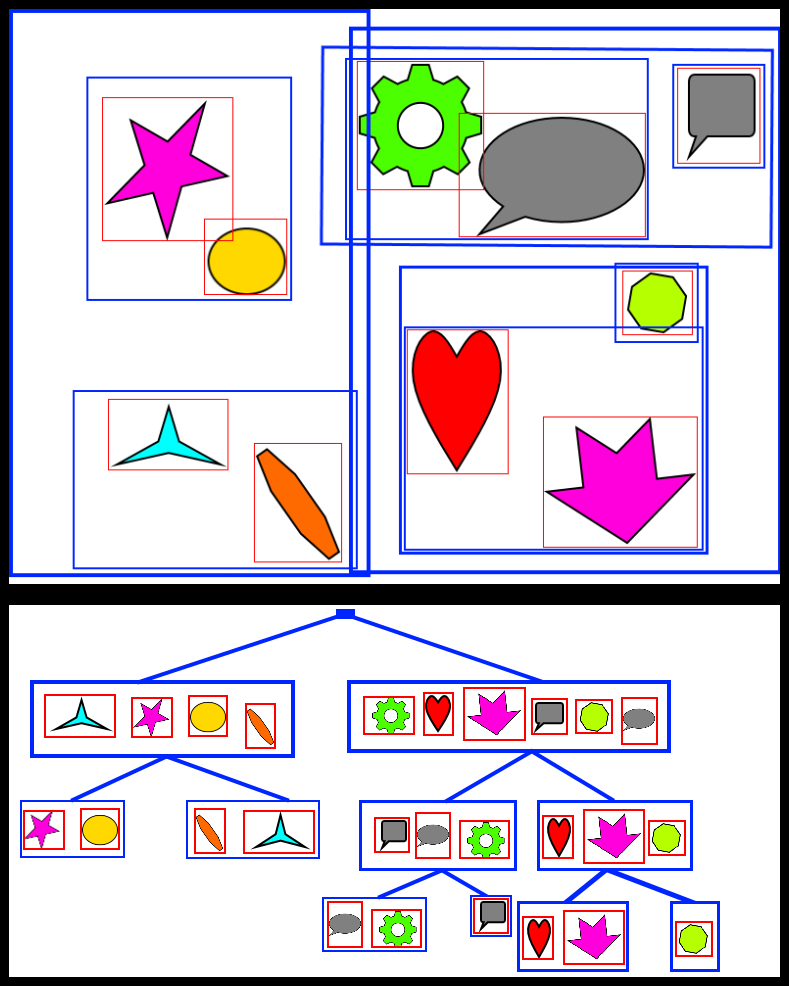


1. Exempel på en miljö som delats upp med Serpa & Rodrigues (2019a) med KD-träd och SAP-hybriden. Överst är själva miljön och därefter kommer trädet som byggts vilket följs av SAP-utrensningen och det inkrementella steget. Sist är SMID-optimerad gruppsökning vilken lokaliserar överlappningarna.

*K-dimensionella träd* (*KD-träd* eller *k-d träd*) är en trädstruktur vilken delar upp miljön en dimensionsaxel i taget. Uppdelningen görs antingen sekventiellt, exempelvis skulle en 3D-miljö delas upp i ordningen: x, y, z, x, y, z, x osv, men det är även vanligt att låta algoritmen välja fritt mellan de tillgängliga dimensionerna (Ericson 2004, s. 319). KD-träd är även en av de algoritmer som lämpar sig bäst för scenarion som har ett stort antal objekt i sig (Ericson 2004, s. 249).

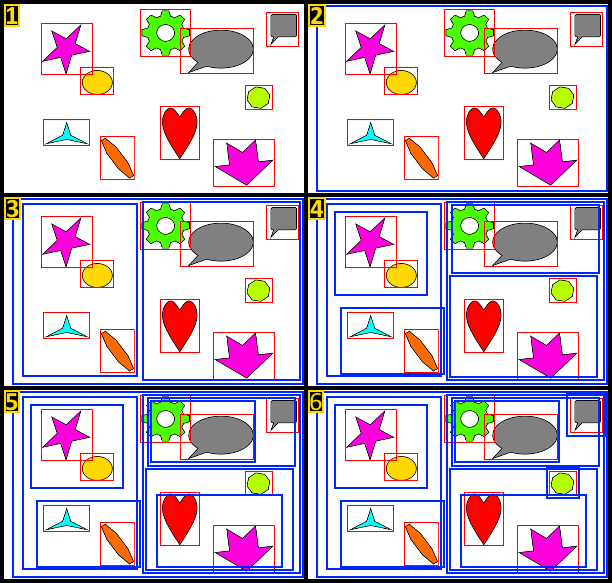
Exempel på användning av KD-träd är Serpa & Rodrigues (2019a) vilka använder dem i kombination med en SIMD-optimerad SAP-algoritm. Serpa & Rodrigues (2019a, ss. 261-266) menar att deras breda kollisionsdetektering utförs i tre steg, först används KD-trädet för att gruppera objekt in i mindre grupper. För slippa beräkna om hela trädet från grunden utnyttjar de samstämmighet mellan uppdateringarna. Istället för att bygga ett nytt träd uppdateras och anpassas trädet från föregående uppdatering. När KD-trädets uppdatering är klar används sedan SAP-algoritmen för att sortera ut ickekollisioner ur noderna. Där efter använder de en inkrementell metod för att sortera bort alla statiska överlappningar och slutligen använder Serpa & Rodrigues (2019a) en SMID-gruppfunktion för att lokalisera överlappningarna, se **figur 5**.

### BVH



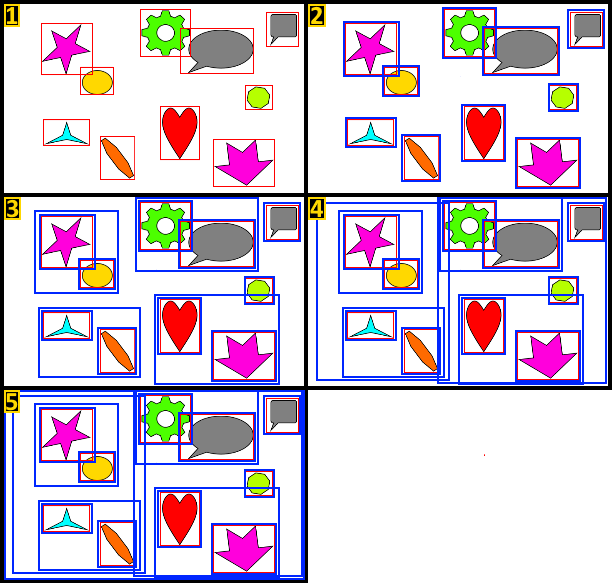
1. BVH som delar upp objekt tills max två objekt finns i varje gränsvolymsnod.  
   Överst: Exempel på en scen som delats upp med BVH   
   Nederst: trädstrukturen som BVH har byggt.

BVH är en trädstruktur vilken rekursivt delar upp objekten i gränsvolymer. Vanligtvis så definieras BVH enligt följande, varje nod associeras med en delmängd objektgränsvolymer. De är även associerade med en gränsvolym som är precis så stor att den innesluter alla objektgränsvolymer, se **figur 6**, (Teschner m.fl. 2005, s. 63). BVH är inte begränsade till men de brukar struktureras som ett binärträd, vilket betyder ett de försöker splittra delmängden av objekt i två (Serpa & Rodrigues 2019b, s.5). Uppdelning brukar göras rekursivt till någon form av lövnodsmål uppnåtts, som exempelvis att det ska finnas X antal eller mindre objekt i dem. Dock är det vanligast att låta uppdelningen pågå till dess varje lövnod innehåller endast ett objekt (Teschner m.fl. 2005, s. 63). BVH används sällan i hybridlösningar till följd av att dess sökning är för noggrann för att kunna användas som superstruktur (grovt första sökning, KD-trädet i Serpa & Rodrigues (2019a) lösning är en superstruktur) samt att den är för beräkningstung för att kunna användas som operator (Serpa & Rodrigues 2019b, s.5).



1. Exempel på bygget av en BVH med toppen-botten, uppdelningen av miljön i mindre och mindre gränsvolymer kan följas från bild 1, 2, 3, …, 6.

Konstruktion av BVH:er kan göras på tre olika sätt: *botten-toppen*, vilken sätter ihop noder två och två utifrån någon form av avståndskriterium (se **figur 8**), *stegvis*[eng: incrementally], i vilken objekten sätts in i BVH-trädet en i taget, *toppen-botten*, delar stegvis upp alla objekt i miljön i mindre grupper. Uppdelningen sker efter förvalda kriterier som exempelvis binärträd (se **figur 7**) (Serpa & Rodrigues 2019b, s.5; Teschner m.fl. 2005, s. 64). Den stegvisa metoden har utnyttjats med stor framgång i exempelvis btDBVT algoritmen från Bullet Library (Coumans 2018). Dock är det toppen-botten som är den vanligaste för beräkningsprocessorbaserad kollisionsdetektering medan botten-toppen är den mest populära för grafikprocessorbaserade kollisionsdetektering (Serpa & Rodrigues 2019b, s.5).



1. Exempel på bygget av en BVHmed botten-toppen, uppdelningen av miljön i större och större gränsvolymer kan följas från bild 1, 2, …, 5.

För att kunna dra nytta av samstämmigheten mellan uppdateringar kan BVH-lösningar använda sig av ett BVTT(gränsvolymstestningsträd[eng:Bounding Volume Testing Tree]) vilket är en struktur som hierarkiskt representerar potentiella kollisioner mellan två BVH:er. Noderna i en BVTT representerar enskilda överlappningstester mellan två gränsvolymer, alternativt kan de representera självkollisionstestet för en gränsvolym (Tian, Hu & Shen 2019, s. 5).

BVH är en av de populäraste algoritmerna för bredkollisionsdetektering som Serpa & Rodrigues (2019b, s.5) påpekar. Där av finns det otaliga exempel på lösningar som använder sig av BVHoch här efter följer ett axplock av dessa. Schmidtke & Erleben (2018) presenterar en grafikprocessorbaserad lösning vilken använder sig av flera storleksbegränsade BVH:er för varje objekt vilka de använder för virtuell prototypstestning. Tian, Hu & Shen (2019) fokuserar på en parallell beräkningsprocessor och grafikprocessorhybridbaserad BVH-lösning, vilken de använder för exakt kollisionsdetektion i hjärndeformationssimulation. Slutligen Wangs m.fl. (2018) lösning är även den grafikprocessorbaserad men särskiljer sig från övriga i och med att den bygger sin BVH-lösning på histogramsortering, vilken de kombinerar med en BVTT som anpassar sig efter arbetsbördan.

### Anpassningsbara algoritmer

Luque, Comba & Freitas (2005, s.180) hävdar att för att en bredkollisionsdetekteringsalgoritmer ska kunna vara riktigt bra så krävs det att den anpassar sig efter arbetsbördan. Detta är något som Serpa & Rodrigues (2019a, ss. 261-262) instämmer med. De utvecklar att deras KD-träd automatisk anpassar sig efter arbetsbördan för kunna vara så generell som möjligt. Tidigare algoritmer för bredkollisionsdetektion påpekar de har tenderat att vara specialiserade. Detta gör att de oftast endast är lämpade för den specifika situation den konstruerades för. Serpa & Rodrigues (2019a, s. 261) exemplifierar detta med bredkollisionsdetektion gjorda för datorspel ofta optimeras för samstämmighet[eng: coherency] då majoriteten av objekten i spelmiljön kan förväntas vara statiska. De menar att spelarnas agerande i spel dock kan leda till osamstämmighet. Om osamstämmighet skulle uppstå förklarar Serpa & Rodrigues (2019a, s.261) att det kommer att leda till markanta prestandaförluster. Wang m.fl. (2018, s. 236) noterade även en markant prestandaökning för sin algoritm jämfört med tidigare lösningar, vilket de bland annat krediterade det till algoritmens anpassningsbara BVTT. Capannini & Larsson (2018) profilerar även sin metod med att den är en anpassningsbar beräkningsprocessorbaserad lösning, vilket enligt deras testningar klarar av att hantera komplexa scenarion med miljontals objekt i rörelse. Alla håller dock inte med om att anpassning efter arbetsbördan är en önskvärd egenskap. Schmidtke & Erleben (2018, s. 3044) profilerar sig exempelvis med att deras lösning inte behöver använda någon form av arbetsbelastningsschema. Serpa & Rodrigues (2019b, s. 3) konstaterar även att för mindre mängder av objekt så kan en arbetsbalanseringsstruktur direkt vara stjälpande, till följd av att tiden som sparas av balanseringen går förlorad till kringarbetet runt den.

## small kollisionsdetektion

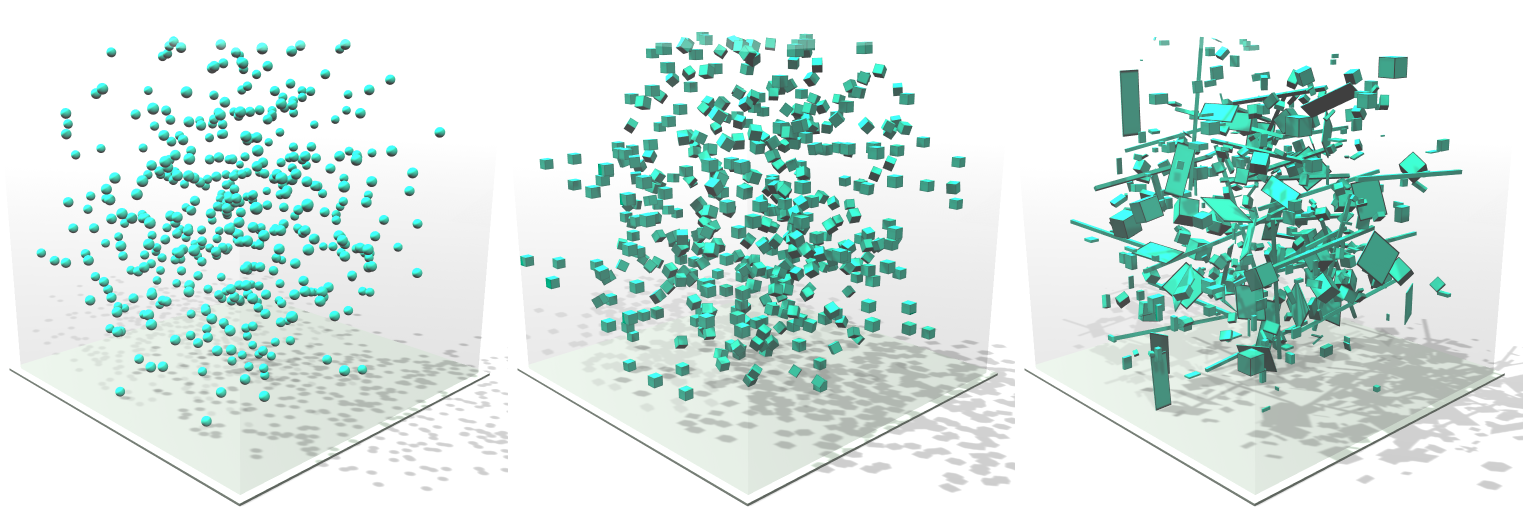
Den smala kollisionsdetektering undersöker kollisionsparen som den breda kollisionsdetekteringen lokaliserat och avgör om kollision har hänt (Serpa & Rodrigues 2019b, s. 1; Ericson 2004, s. 14). Serpa & Rodrigues (2019a, s. 260) förklarar att den smala kollisionsdetekteringen använder objektens geometri för att utföra testerna. Detta menar de gör att implementation av den smala kollisionsdetekteringen skiljer sig avsevärt beroende på vad det är som kolliderar (stelakroppar, mjuka kroppar, partiklar, kombination av dem m.m.). Dock förklarar Serpa & Rodrigues (2019b, s. 1) vidare att om precision inte är en nödvändighet så kan den smala kollisionsdetekteringen förenklas, alternativt utelämnas helt. Exempelvis i spel är det sällan nödvändigt med exakt kollision så simplare geometriska approximation kan användas istället för den exakta geometrin. Den smala kollisionsdetekteringen använder mycket tyngre beräkningar för utvärdera om kollisioner har uppstått. Därför är det, som Zhang & Liu (2015, s.31) noterar, viktigt att eliminera så många falska kollisioner som möjligt innan den smala kollisionsdetekteringen utförs.

## Broadmark

När det kommer till bredkollisionsdetekteringen finns det som Serpa & Rodrigues (2019b, s.1) noterar ingen standard för hur testning och utvärdering utförs. Serpa & Rodrigues (2019b, s.2) förklarar att ett fåtal försök har gjorts för etablera ett gemensamt testramverk. Dock konstaterar de att dessa ofta har lidit av att vara komplicerade att hantera och begränsade i vad de kan göra. Denna avsaknad av en standardiserad testnings, och utvärderingsramverk förklarar Serpa & Rodrigues (2019b, s.1) försvårar jämförelsen av olika algoritmer, speciellt om det är olika fakulteter som utvecklat dem (exempelvis jämföra en algoritm gjord inom operationssimulerings fakultet med en gjord inom robotikfakultet). De utvecklar även att det saktar ner utvecklingen av algoritmer då det leder till ogenomtänkt testning i vilka irrelevanta parametrar testas, samt påpekar Serpa & Rodrigues (2019b, s.1) att risken ökar för att arbete dupliceras. Utifrån detta så presenterar Serpa & Rodrigues (2019b) ett testramverk för bredkollisionsdetekteringsalgoritmer. Ramverket förklarar de består av två delar, *simuleringsgeneraren* och *Broadmark.* Simuleringsgeneraren har till uppgift att skapa och spara ner 3D-simuleringserna, medan Broadmarkutför testerna av bredkollisionsdetekteringsalgoritmer.

Simuleringsgeneraren utvecklade Serpa & Rodrigues (2019b, s.6) med C# i unity och den har som uppgift att skapa 3D-scener med tusentals rörliga objekt vilka används för att testa bredkollisionsdetekteringsalgoritmerna. De belyser simuleringsgeneratorns sex huvudaspekter.

Serpa & Rodrigues (2019b, s.6) redogör först att simuleringsgeneraren är oberoende av fysikmotorn. Simuleringsgeneratorn förklarar de har nämligen ett gränssnitt för fysikmotorer vilket gör det enkelt att ändra vilken som används. Systemet har inbyggt stöd för fysikmotorerna PhysX och Bullet, en beskådarmotor som kan användas för att spela om skapade scener samt en splittermotor för att kombinera flera mindre simuleringar till en större.



1. De tre objekt typerna som stöds i simuleringsgeneratorn. Till vänster sfärer, mitten kuber och till höger olikformade rektanglar.

För objekt redogör Serpa & Rodrigues (2019b, s.6) att simuleringsgeneratorn preliminärt har stöd för sfärer, kuber och olika former av rektanglar, se **figur 9**. I börja av simuleringen placeras dessa föremål ut på slumpvist valda positioner och startas med en slumpad utgångshastighet.

De tillhandahåller även tre inbyggda test scener i simuleringsgeneraren. Den först är *Fritt fall*, alla föremål i scenen får falla fritt tills de kolliderar med marken och i slutet av simuleringen har alla objekt blivit statiska. Denna scen representerar ett statiskt eller förutsägbart scenario. Den andra är *Slumpvandring*[eng: Brownian], vilket periodvis slumpar nya hastigheter till objekt så att de aldrig kommer till ett vilande stadium. Detta representerar ett scenario med jämt fördelat kaos. Slutligen *Gravitet*, vilken periodvis ändrar gravitationens riktning så att objekt förflyttar sig runt scenen i en stor hög. Detta representerar ett fullt dynamiskt scenario med massivt antal kollisioner.

Simuleringsgeneraren förklarar Serpa & Rodrigues (2019b, s.6) sparar ner scenernas exekvering i ett enkelt binärtfilformat. Information som sparas ner är scenens namn, vilken typ av objekt som hanteras, hur många objekt det är, antalet bilder simuleringen består av och slutligen objektens AABB:er för varje bilduppdatering. Serpa & Rodrigues (2019b, s.6) utvecklar att genom ned sparningen så frånkopplas exekveringen av algoritmerna från genereringen av simuleringarna. De förklarar att detta möjliggör för andra att implementera egna simuleringsgenereringsapplikationer.

Serpa & Rodrigues (2019b, s.6) redogör att simuleringsgeneraren kommer med uppsättnings applikation för att underlätta skapandet av testsimuleringar. De exemplifierar detta med att det går att automatiks generar sex olika simuleringar med varierande antal objekt för två olika test scener.

Den andra delen i ramverket är Broadmark vilket Serpa & Rodrigues (2019b, s.6) förklara används för att utföra testerna på algoritmerna och mäta resultatet. De förklarar att systemet använder en parameterfil, vilken definierar vilka algoritmer som ska användas, samt en testscen som indata. Efter att simuleringen har körts förklarar de att programmet tillhandahåller som utdata en summering av hela exekveringen i ett per-uppdateringsformat. Information som tillhandahålls inkluderar tiden det tog att uppdatera strukturerna och objektet, likaså hur lång tid kollisionsdetektionen tog.

Serpa & Rodrigues (2019b, ss.6-8) tar även upp att Broadmak har inbyggt i sig 13 algoritmfamiljer för bred kollisionsdetektering, råstyrka, SAP, rutnät-råstyrka, rutnät-SAP, Axelsvep, DBVTF, DBVTD, CGAL, Tracy, KD-träd, grafikprocessor-rutnät, grafikprocessor-LBVH och grafikprocessor-SAP, se **tabell 1** för en summering av algoritmerna.

1. Summering av algoritmer inkluderade i Broadmak (Serpa & Rodrigues 2019b, s.9)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Implementation | | | | | |
| Algoritmer | | Princip | Optimering\* | Inkrementell | Noteringar | källor |
| **Råstyrka** | | Råstyrka | SIMD + MT\*\* | **\_** | Naiv | Serpa & Rodrigues (2019a) |
| **SAP** | | SAP | SIMD + MT | **\_** | STL sortering | Serpa & Rodrigues (2019a) |
| **Rutnät-Råstyrka** | | Rutnät | MT | **\_** | *T* objekt/cell | Serpa & Rodrigues (2019a) |
| **Rutnät-SAP** | | Rutnät + SAP | MT | **\_** | *T* objekt/cell | Serpa & Rodrigues (2019a) |
| **Axelsvep** | | iSAP | **\_** | Ja | Insättningssortering | Bullet 2 |
| **DBVT F** | | BVH | **\_** | Ja | Konstant träd | Bullet 2 |
| **DBVT D** | | BVH | **\_** | **\_** | Konstant träd | Bullet 2 |
| **CGAL** | | Träd + SAP | **\_** | **\_** | Tillståndslös | CGAL |
| **Tracy** | | Rutnät + iSAP | MT | Ja | Insättnings sortering | Tracy, Buss & Woods (2009) |
| **KD träd** | | Träd + SAP | SMID | Ja | Anpassningsbar, Konstant träd | Serpa & Rodrigues (2019b) |
| **Grafikprocessor-rutnät** | | Rutnät | Grafikprocessor | **\_** | OpenCL | Bullet 3 |
| **Grafikprocessor-LBVH** | | BVH | Grafikprocessor | **\_** | OpenCL | Bullet 3 |
| **Grafikprocessor-SAP** | | SAP | Grafikprocessor | **\_** | OpenCL | Bullet 3 |

\*Notera att när det står något annat än \_ eller Grafikprocessor finns mer än en variant av algoritmtypen. Exempelvis för Råstyrka så finns det en grundimplementation, en SIMD-optimerad version, en MT-optimerad version och en SMID-MT-optimerad version.

\*\* MT (flertrådad[eng:Multi-threaded] d.v.s. parallell version)

Serpa & Rodrigues (2019b, s.9) tar upp att trots att ramverket täcker de flesta scenarion så har den begränsningar. De utvecklar det med att för tillfället är simuleringarna begränsade till konstanta antal objekt och därmed går det inte att addera nya objekt under simuleringens gång. AABB är även den enda gränsvolym som för tillfället går att använda med system. Serpa & Rodrigues (2019b, s.9) nämner att algoritmerna som kommer med systemet är alla kompletta, vilket innebär att de rapporterar alla kolliderande gränsvolymer. Dock poängterar de att några av algoritmerna rapporterar även ett par falska kollisioner, till följd av att de använder något större gränsvolymer än övriga algoritmerna. Serpa & Rodrigues (2019b, s.9) menar att Broadmark inte kan ta med den negativa effekten dessa falska kollisioner har på algoritmernas prestanda. De nämner att Broadmark däremot registrerar kvantiteten av falska kollisioner som testas. Slutligen nämner de att spelliknade applikationer oftast även utnyttjar bredkollisionsdetektionen för att accelerera uppgifter som strålsökning[eng:ray-cast] och lådspårning[eng: boxcast]. För tillfället har inte Broadmark förmågan att testa dessa sorters förfrågningar, förklarar de.

# Problemformulering

Målet med detta arbete är att utvärdera tre stycken samtida bredkollisionsdetektionsalgoritmer. De tre algoritmerna är, BVHlösningen som Wang, Tang, Manocha & Tong (2018a) presenterar (enoteras här som *BVH-SR*), parallella SAP algoritmnn som Capannini & Larsson (2018) redogör för (denoteras här som *PSM-SAP*), KD träd och SAP-hybriden som Serpa & Rodrigues (2019a) skapat (denoteras här som *KD-SAP*). Dessa tre algoritmer har valts eftersom de är sentida arbetsbelastningsanpassade lösningar, men även då BVH-SR, PSM-SAP och KD-SAP representerar de tre olika lösningsstrukturerna presenterade i bakgrunden.

Utöver KD-SAP, PSM-SAP och BVH-SR kommer även de tolv övriga algoritmfamiljerna som inkluderades med Broadmark, se **tabell 1**, att testas i detta arbete. Detta görs både för att ha basvärden att jämföra de tre samtida algoritmerna emot samt för att bredda testningen.

Datorspel blir bara större och större i skala, speciellt inom AAA sektorn. Samtidigt som spelvärdarna ökar i storlek har kraven på grafiska kvalitéerna och interaktiviteten ökat. I ett modernt tävlingsinriktat-online-baserade-flerspelar ställer spelarna idag höga krav på de grafiska kvalitéerna, vilka förväntas uppdatera med minst sextio bilder per sekund, samtidigt som responstid för interaktioner ska vara så nära noll som möjligt. Allt detta gör att det har blivit viktigare än någonsin att spelutvecklare väljer bredkollisionsdetektionsalgoritmer som är optimerade och effektiva för den miljön som algoritmen ska användas i. Spelarnas nyckfullhet i kombination med större och mer dynamiska spelvärldar gör det dock svårare att optimera algoritmerna för en specifik form av scenario.

Frågan som söker att besvaras utifrån detta är:

* *Hur väl presterar KD-SAP, PSM-SAP och BVH-SR för datorspel-liknande-scenario (DLS) med avseende på exekveringshastighet jämfört med de tolv övriga algoritmfamiljerna som inkluderades i Broadmark*

Det ska förtydligas att prestanda syftar här på att exekveringshastigheten för att lokalisera alla potentiella kollisioner under en bilduppdatering. Det ska även noteras att den enda gränsvolymen som kommer användas är AABB. Fokusen för detta arbete är nämligen att testa algoritmerna och inte gränsvolymerna. Mjuka objekt har även utelämnats från detta arbete då huvudfokus är riktad mot datorspel, i vilka majoriteten av objekten är icke-deformerbara.

Det kommer vara omöjligt att testa alla DLS:er som finns till följd av spels mångfasighet, så detta arbete kommer behöva avgränsa vilka DLS:er som används. Med det sagt så kommer scenariona som används vara de tre som inkluderas i Serpa & Rodrigues (2019b) scenariogenerator. Inget av dessa scenarion är huvudsakligen gjort för spel men de är så pass generella att det enkelt går att anpassa dem till att fungera för det. Det enda som behöver göras är att addera statiska objekt till dem. Här följer exempel på vilka DLS kan representera:

* F*ritt fall* fungerar som en analog för exempelvis att en byggnad förstörs i ett spel.
* *Slumpvandring* fungerar som en analog för exempelvis ett asteroidbälte eller approximation av ett fiskstim
* *Gravitet* kan sägas representera ett rent stresstestscenario som exempelvis att spelaren lyckas klumpa upp alla interaktiva dynamiska föremål i en spelmiljö på en liten yta.

Detta är bara ett par scenarion som de skulle kunna tänkas representera. Men tydligt är att deras generalitet gör att många fler kommer kunna nyttja information som tas fram i detta arbete, jämfört med om exempelvis ett exakt scenario från ett specifikt datorspel används. En annan stor fördel med att använda Fritt fall, Slumpvandring och Gravitet är att de redan finns implementerade och endast behöver mindre modifieringar, adderandet av statiska objekt, för att kunna användas. Detta är något som kommer spara in tid så att fler tester kan utföras och därmed kan mer data samlas in.

## Metodbeskrivning

Algoritmerna kommer testat med ramverket som Serpa & Rodrigues (2019b) presenterar. Beslutet togs att använda deras ramverk då det är ett gediget och flexibelt testramverk för bredkollisionsdetektion. Det erbjuder även tre testscenarion, tretton inbyggda algoritmer samt mäter ramverket, det som önskas mätas i denna undersökning. Det bör dock noteras att deras ramverk är relativt nytt, förutom i artikeln där Serpa & Rodrigues (2019b) presenterar ramverket så har ingen annan använt det än. Båda skribenterna är dock väl citerade och det finns ingen anledning att misstro ramverket de presenterar. Men för att testa validitet av ramverket kommer dock ett mindre test att göras. En egen utloggning av tiden de tar för algoritmerna att detektera alla kollisioner kommer att göras för att kunna jämföra med resultatet de presterar.

Testning av algoritmerna kommer göras med modifierade, se **3.1.2**, versioner av de tre testscenariona som är inbyggda i ramverket eftersom de fungerar som bra analoger för scenarion som kan uppstå i datorspelspel,

* F*ritt fall* fungerar som en analog för exempelvis att en tågbro sprängs
* *Slumpvandring* fungerar som en analog för online baserade flerspelarspel där dynamiska objekt kan ses som spelare.
* *Gravitet* kan sägas representera mer fysikbaserade spel, alternativt spel med ett stort antal interaktiva objekt som spelaren kan påverka. Det skulle även kunna tänkas representera ett taktikspel med två arméer som strider.

### Testningen av prestanda för olika scenarion

För att få en bild av hur bredkollisionsdetektionsalgoritmerna presterar under olika scenarion så kommer varje testscenario att köras flertalet gånger med varierande antal objekt. Optimalt borde alla möjliga kvantiteter av objekt testas men denna studie utförs under en begränsad tidsperiod så antalet tester måste begränsas. För att få en bra representation av bredkollisionsdetektionsalgoritmernas förmågor med hänsyn till tiden som finns så kommer terserna delas upp i tre intervaller *små* som testar tio till hundra objekt med fem intervaller, *mellan* som testar tusen till tiotusen med femhundra intervaller, *stora* som testar hundratusen till en miljon med femtiotusen intervaller. De tre intervallerna valdes för att undersökningen skulle bli så bred som möjligt utan att ta för lång tid. Att testa bredkollisionsdetektionsalgoritmer med varierande antal objekt är något för att utvärdera hur algoritmerna skalarar är något som är vanligt förekommande. Metoden används av Luque m.fl. (2005), Capannini & Larsson (2018) och Serpa & Rodrigues (2019a, 2019b) för att bara nämna några. Att köra alla tre intervaller på ett testscenario definieras som ett *set*, så att utföra intervallet små, mellan och stora på scenen Slumpvandring är att göra ett *set* på Slumpvandring.

### Testningen av prestanda beroende på förhållandet mellan statiska och dynamiska objekt

Hur bredkollisionsdetektionsalgoritmerna hanterar olika distributioner mellan dynamiska och statiska objekt kommer även de att testas, genom att andelen statiska objekt varieras. Hur fördelningen mellan dynamiska och statiska objekt påverkar algoritmerna är något som både Luque m.fl. (2005) och Serpa & Rodrigues (2019a, 2019b) testar. De testar dock endast med scenarion där alla objekt släps och får falla till marken där de till sist blir vilandes, d.v.s. statiska. Denna undersökning testar mot DLS:er vilka består av ett stort antal varierade scenarion, dock brukar majoritet av objekt i dem vara statiska. Att endast testa med det enkla scenario som Luque m.fl. (2005) och Serpa & Rodrigues (2019a, 2019b) använder sig av anses därav inte vara tillräckligt, istället kommer andel statiska objekt varieras i alla de scenarion som testat. Dock kommer samma argumentation som den för variation av antalet objekt, det vill säga variationer av andelen statiska objekt att begränsas. För få ut maximalt med andelen testdata utifrån tidsramen som tilldelats kommer därav varje *set* testas med sex olika andelar statiska objekt: 0 %, 20 %, 40 %, 60 %, 80 % och 100 %. Att testa ett *set* med alla sex fördelningar av statiska objekt definieras som en *omgång.*

### Hur slumpens inverkan ska minimeras

Simuleringsgeneraren spelar upp simuleringarna och sparar ner alla objektens AABB:er för varje bild i exekveringen, och Broadmark använder detta för att testa bredkollisionsdetektionsalgoritmerna. Testningen görs genom att Broadmark tar information som sparades ner för varje bild och kör bredkollisionsdetektionsalgoritmerna över dem och kontrollerar hur lång tid det tar för algoritmen att hitta alla överlappningar. Eftersom testscenariona kommer att utföras exakt lika dant varje gång Brodmark använder dem, anses det därför inte vara nödvändigt att testa vare simulering mer än en gång, vilket låser upp tid för att testa flera instanser av varje simulering. Att endast köra varje simulation endast en gång är även något som Serpa & Rodrigues (2019b) gör när de utför testerna.

Även om simuleringarna exekverar på exakt samma sätt varje gång de körs så innehåller fortfarande skapandet av den slumpbaserade variabler som objektens startposition, deras utgångshastighet samt vilka objekt som är statiska. Slumpningen av dessa parametrar ger ett mer verklighets liknande testscenario, eftersom att det i verkligheten inte går att garantera en optimal situation. Det gör dock att en situation som missgynnar en specifik algoritm kan uppstå. För att minimera risken att en algoritm stjälps av slumpen kommer flera instanser av varje simulering att testas, utifrån vilka ett medelvärde kan beräknas. Att utvärdera en algoritm med medelvärdet från flertalet körningar av en simulation är något är vanligt förekommande. Det använda exempelvis av Schmidtke & Erleben (2018) och Weller, Debowski & Zachmann (2017) när de utvärdera hur sina algoritmer. Just för detta arbete kommer varje scen mer specifikt dupliceras tjugo gånger. Det beslutades med tjugo dubbletter för att eliminera slumpens inverkan så mycket som möjligt. Att inte fler tester görs beror på tidsbegränsningen arbetet har i sig. Notera dock att om mer tid finns över efter att testerna har gjorts så kan potentiellt mer tester göras.

### Testning av presterande berodde på typen av objekt

Alla tester kommer även att utföras två gånger: En där alla objekten är likformiga kuber och en där objekten består av olikformade rektanglar. Detta görs för att kunna undersöka hur algoritmerna påverkas av objektens komplexitet. Broadmark har även stöd för sfärer, dock anses det inte nödvändigt att testa dem då Brodmark ändå bara stödjer AABB, så resultaten kommer därav att till stor grad vara snarlikt det med sfärer. Det ska noteras att när Serpa & Rodrigues (2019b) gjorde sina tester så utelämnade de testning med olikformade rektanglar. Detta utvecklade Serpa & Rodrigues (2019b, s.9) beror på att de sedan tidigare kunde påvisa att testning med olikformade objekt påverkar majoriteter av algoritmerna likvärdig. Serpa & Rodrigues (2019b, s.9) förklarar att den enda algoritmen som påvisade en nämnvärd prestandaförsämring när olikformade rektanglar användes var den BVH-baserad algoritm. En av algoritmerna som inte tidigare testats i Broadmark men som kommer testas i denna undersökning är BVH-SR. Därav anses det även vara av intresse att testa algoritmerna mot olikformade rektanglar, det gör även scenariona mer lika datorspels miljöer då sällan alla objekt är exakt lika stora i dem.

Under simuleringarna sparas exekveringstiden, mät i sekunder, det tar för algoritmerna att lokalisera alla potentiella kollisioner för varje enskild bilduppdatering. Detta är samma prestanda mått som Zhang & Liu (2015) och Serpa & Rodrigues (2019a, 2019b) använder sig av. Denna data kommer att användas för att utvärdera hur en algoritm presterade överlag samt hur dess prestanda förändras under beroende på antal objekt som testats.

Resultaten kommer att presenteras med ett violindiagram, vilka ger en nyanserad bild av hur stabila algoritmerna är i sin prestanda. Vilket är extra intressant för spelutvecklare eftersom om en algoritm har stor variation i hur den presterar skulle vara ett dåligt alternativ. Om prestanda ofta faller kommer det nämligen ha negativ inverkan på interaktivitet.

### Metoddiskussion

Metoden som valts för detta arbete är inte perfekt då den inte testar mot exakta datorspel-scenarion. Metoden använder snarare mer abstrakta generaliseringar av de situationer som kan uppstå i datorspel. Detta är en svaghet med testningen då den blir mindre specifik för spelmiljöer, dock är det till fördelen då en större andel av spelgenres potentiellt kan använda sig av information från testerna. Mer generella tester gör nämligen, som namnet antyder, testdata mer generell, exempelvis gravitet scenariot skulle kunna tänkas representera hundratals datorstyrda karaktärer som kolliderar med varandra i ett strategispel eller ett massivt antal objekt som flyttas omkring i ett fysikpusselspel. Största svagheten för nuvarande metod är dock att det inte går att testa övriga användningsområden som bredkollisionsdetektionsalgoritmer används för i dataspel. Dock finns det inte tillräckligt med tid för att implementera det som en del i detta arbete.

Ett alternativ till den metoden som presenteras här skulle vara att för varje algoritm utföra en algritmanalys, vilken går ut på att utvärdera algoritmerna och komma fram till vilken dess tidskomplexitet är (sämsta exekveringstid). Det positiva med det är att tidskomplexitet används just för att jämföra algoritmer, vilket samstämmer med ett av målen för arbetet. Däremot säger tidskomplexet ingenting om hur algoritmen presterar i ett specifikt scenario. Det är även värt att notera att majoriteten av undersökningar som görs på bredkollisionsdetektion använder sig av testning snarare än algoritmanalys som Capannini & Larsson (2018), Wang m.fl. (2018), Serpa & Rodrigues (2019a), Li m.fl. (2018) och Avril m.fl. (2014) gör för att bara nämna ett fåtal. Även de som använder sig av algoritmanalys som Weller m.fl. (2017, s.136) utför ändå oftast experimentet för att konkretisera algoritmens förmågor.

Arbetet är tänkt att vara riktade mot spelbranschen så ett relevant alternativt sätt för arbetet skulle kunna ha gjorts på vore genom att implementera faktiska spelscenarion eller testa gentemot redan existerande spel. Detta skulle ge mer precisa data om specifika spelscenarion, vilket skulle avgränsa andelen personer som skulle ha nytta av studien. Ett exempel vore om testet utformades för att simulera en första-persons-skjutar-spel så skulle endast utvecklare av just denna genre av spel direkt nyttja dem. Detta är dock ett problem som skulle kunna motverkas genom att utveckla simuleringar av flera olika genrers. Datorspel använder inte endast bredkollisionsdetektionsalgoritmerna för kollisionsdetektionen utan även för att snabba på andra arbetsuppgifter, och Broadmark kan inte testa hur algoritmerna presterar för dessa användningar. Optimalt hade därav varit att antingen expandera funktionalitet för Broadmark så den kan testa denna typ av funktionalitet, alternativt utveckla ett eget testramverk som inriktar sig specifikt på att testa detektionsalgoritmerna gentemot datorspel. Största nackdelen med att göra detta är dock att det kommer ta tid att utveckla både testsimuleringar av spelen och nya testverktyg, vilket är något som det helt enkelt inte finns tid för. Att testa gentemot genuina datorspelscenarion som ska spelas av riktiga spelare skulle också introducera en stor slumpfaktor i testerna, i form av den mänskliga faktorn, så att en mycket större kvantitet av data skulle behövas samlas in. Möjligen skulle detta underlättas genom att testa med hjälp av artificiell intelligens, dock kommer även detta alternativ att kräva markant med tid för utvecklingen av den artificiella intelligensen.

### Vad som ska implementeras

För denna undersökning kommer totalt sex saker behöva implementeras:

* BVH-SR algoritmen som Wang, Tang, Manocha & Tong (2018a) presenterar.
* PSM-SAP algoritmen som Capannini & Larsson (2018) presenterar.
* Modifierad version av testscenariot *Fritt fall*
* Modifierad version av testscenariot *Slumpvandring*
* Modifierad version av testscenariot *Gravitet*
* Modifierad version av uppsättningsapplikation.

De tre modifierade versioner av testscenariot kommer vara identiska med de som inkluderades, med undantaget att det kommer att gå att ställa in andelen objekt som ska vara statiska från början. Uppsättningsapplikationen kommer att modifieras så att det går att variera antalet statiska objekt, samt för att möjliggöra så att ett mindre antal objekt kan testas då nuvarande system stödjer minst tusen objekt. Det är även tänkt att automatisera genereringen av simuleringarna.

# Implementation

[Kapitlet kan rubriksättas *Genomförande* eller *Implementation* eller *Projektbeskrivning* eller liknande. Innehåll kan variera beroende på projektets innehåll men nedan finns förslag på upplägg.

Viktigt i detta kapitel är beskrivningen av progression och dessignval, och då även att diskutera och problematisera alternativ som valdess bort, varför detta skedde och stickspår till den valda lösningen.

För en programmerare kan detta innebära att algoritmer och lösningar som förkastats vägs mot den valda lösningen.]

Implementation av testramverket är uppdelat i fem delar, första delen är uppsättningen och valideringen utav Broadmark. Den andra delen är anpassningen av simulationsgeneratorn för att den ska kunna skapa scenarion med statiska objekt i samt för att effektivisera simulationsgenereringen. Tredje delen består av att anpassa hjälp skripten som ramverket använder för att automatisera testningen och visualisera resultaten. Fjärde och sista stege är implementation och integrationen av BVH\_SR och PSM-SAP i Broadmark.

## Uppsättning och validering utav av Broadmark

### Uppsättning

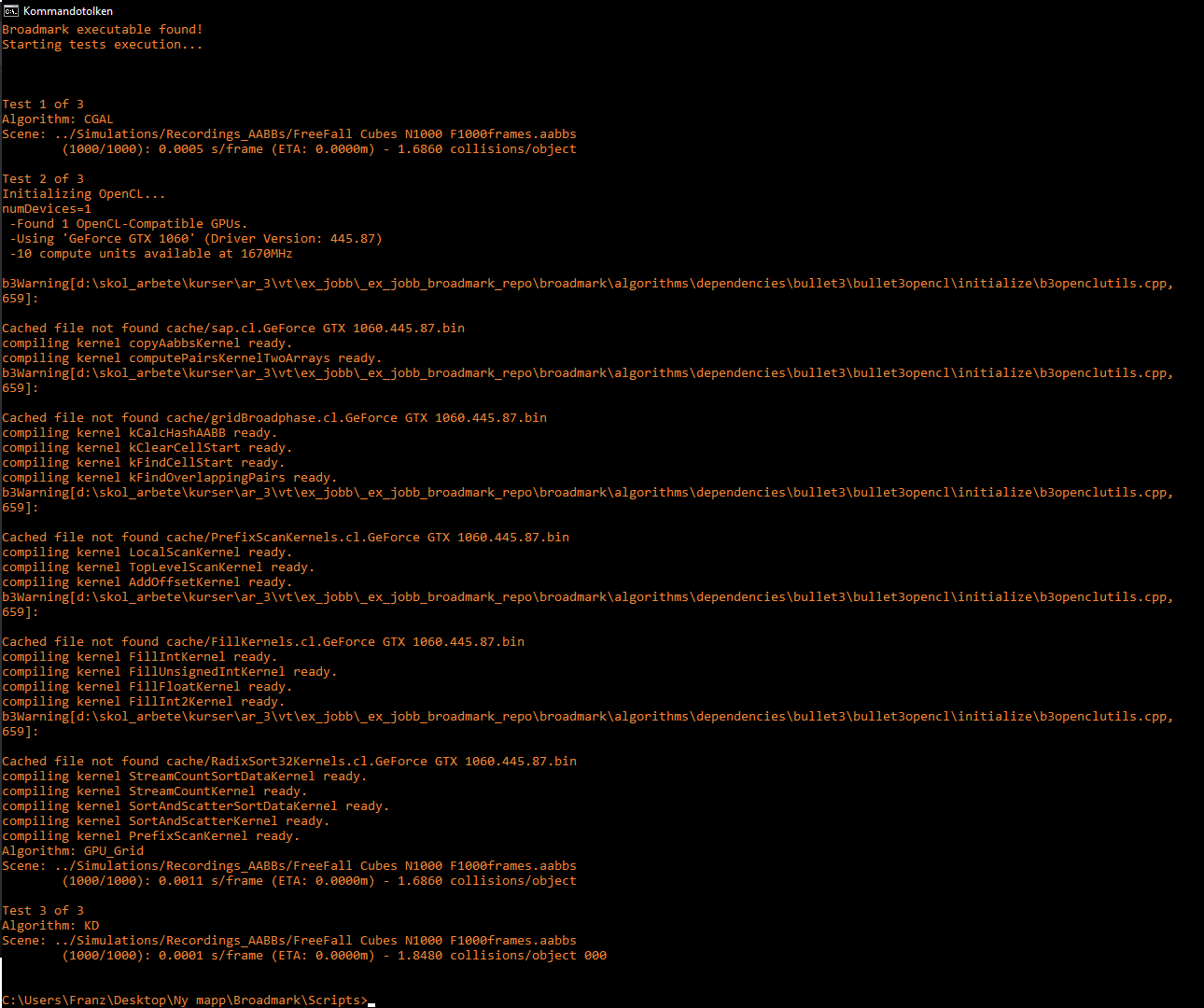
Broad mark är implementerad i utvecklingsmiljön *Visual Studio 2017* och är skriven med programmeringsspråket c++. Det är därmed konsekvent att det är det som används i denna undersökning för att vidareutveckla Broadmark.  
Broadmarks källkod finns att ladda ner från internet (Serpa & Rodrigues, 2019c) och den behövde inte extra uppsättningen för att fungera när den väl laddats ner. Dock förklarar Serpa & Rodrigues (2019c) att om det önskas implementera egna algoritmer i Broadmark så krävs det extra uppsättning för några av de algoritmerna som inkluderats. De förklarar det är alla de grafikprocessorbaserade algoritmerna samt CGAL-algoritmen som kräver extra uppsättning, i form av externa programvarebibliotek som laddas och länkas till dem. Planen är att testgrunden som BVH-SR, PSM-SAP och KD-SAP ska testas emot ska bestå av de tolv övriga algoritmerna som inkluderas i Broadmark, så det beslutades att lägga ner arbetet som behövdes för uppsättning av dem. Serpa & Rodrigues (2019c) förklarar i uppsättningsinstruktionerna att för de grafikprocessorbaserade algoritmerna är det nödvändigt att ladda ner *OpenCL SDK* för grafikprocessorn som ska användas. För denna undersökning innebar det att ladda ner *CUDA Toolkit 10.2*, (2007). Serpa & Rodrigues (2019c) utvecklar vidare att för CGAL-algoritmen krävdes att ladda ner, kompilera och länka *CGAL* (1995). Vidare rekommenderade de att enklaste sättet att göra detta var med hjälp av pakethanteraren Vcpkg (2016).



1. De två versionerna av CGAL (1995) listad i Vcpkg (2016)

Uppsättningen för de grafikprocessorbaserade algoritmerna gick problemfritt, dock var uppsättningen av CGAL-algoritmen mer problemfylld, för även efter att *CGAL* (1995) laddats ner så gick det inte att bygga Broadmark. Till följd av oerfarenhet av att jobba med externa programvarebibliotek i *Visual Studio 2017* var detta något som var svårt att komma fram till, vad exakt problemet var. Det visade sig i slutänden att det fanns två olika varianter av *CGAL* (1995), **figur 10**, och det beskrevs aldrig att båda behövdes för att det skulle fungera.

När väl båda versioner av *CGAL* (1995) laddats ner och länkats så fungerade bygget av Broadmark, i **figur 11** kan testkörningen på testscenariot som skickades med i Broadmark ses.



1. Utloggningen från testexekveringen av Broadmark, algoritmerna som testkördes var KD-SAP, CGAL och grafikprocessor-rutnät

### Validering bakgrund

En av grundegenskaperna som ett testramverk ska uppfylla förklarar Kistostwski, Arnold, Huppler, Lange, Henning, & Cao(2015 ss. 334, 335) är reproducerbarhet. Kistostwski m.fl. (2015) utvecklar att reproducerbarhet syftar både till för att ett testramverk ska resultatet för en testdata vara likartat över flera separata körningar samt att olika användare oberoende av varandra ska kunna upprepa testet och få samma resultat. Kistostwski, Arnold, Huppler, Lange, Henning, & Cao(2015 s. 334) tar även upp att ett testramverk ska vara verifierbart vilket definieras som att det ska vara möjligt att verifiera att ramverket är tillförlitligt.

Dessa tre egenskaper kommer att testas för Broadmark genom att ett testscenario och en algoritm kommer att testas upprepade gånger. Under testningen kommer, utöver Broadmarks egen tidtagning, även en egen tidtagning utföras. Utifrån detta kommer det att vara möjligt att sammanställa ett medelvärde och standardavvikelsen för tiden det tog algoritmen att finna alla potentiella kollisioner. Detta kommer att beräknas både för den egna tidtagningen samt för Broadmarks tidtagning, samt kommer differensen mellan den egna tidtagningen och Broadmarks tidtagning beräknas. Slutligen kommer medelvärdet Broadmark uppmätte att jämföras med det medelvärde som Serpa & Rodrigues(2019b) uppmätte under testerna som de utförde. Utifrån denna data kommer följande slutsatser kunna dras:

* Om den egna tidtagningen påvisar samma värde som Broadmark kan den antas uppfylla verifierbarheten.
* Om Broadmark påvisar en låg standardavvikelse kan Broadmark antas uppfylla första delen av reproducerbarhet kravet.
* Om Broadmark uppmäter samma medelvärde som Serpa & Rodrigues(2019b s.12) uppmätte kan Broadmark antas uppfylla andra delen av reproducerbarhets kravet.

Själva testet som ska utföras måste utifrån ovanstående krav vara ett av de test som Serpa & Rodrigues(2019b) utförde samt testningen själva bör ta så lite tid som möjligt, detta för att mer tid ska kunna läggas på det övriga arbetet. Utifrån dessa krav beslutades det att valideringstestningen skulle utföras med följande inställningar:

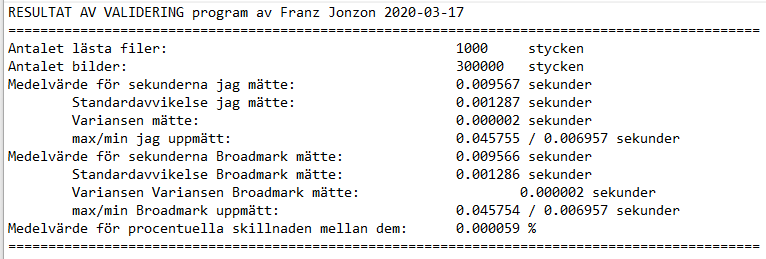
* Testscenario, Fritt fall
* Antal objekt, 8000 stycken
* Antal uppdateringar, 300 stycken
* Algoritm, CGAL
* Antal testkörningar, 1000 stycken

Motiveringen till dessa är följande: Fritt fall valdes eftersom den stabiliserar sig fortare än de andra två. Det tar nämligen runt 300 uppdateringar för samtliga objekt att bli vilandes på marken varefter testet kan avbrytas. Det är även motiveringen till att 300 uppdateringar valdes. När Serpa & Rodrigues(2019b s.12) utförde sina tester var det lägsta antalet objekt som resulterade i ett uppmätbart värde 8000 stycken, så det avgjordes att det var denna mängd som skulle användas. Utifrån detta valdes sedan CGAL som algoritm då den påvisade en av de lägsta uppmätbara tiderna för 8000 stycken. Det tog i genomsnitt 0.01 sekunder för den att detektera alla överlappningar. Slutligen beslutades det att 1000 testkörningar skulle göras med de definierade parametrarna för att minimera slumpens inverkan på resultat så mycket som möjligt. 1000 testkörningar kan användas då det kommer ta runt 50 minuter att utföra valideringstestningen, förutsatt att det, som Serpa & Rodrigues(2019b s.12) uppmätte, i genomsnitt tar 0.01 sekunder att utföra överlappningstestningen för varje uppdatering, vilket anses vara acceptabelt med tid att lägga på valideringstestning.

### validering Implementation

För att automatisera valideringstestningen skapades hjälpskript i python för att automatisera testkörningen. Ett validerings script skapades i c++ och integrerades in i Broadmark för att utföra den egna tidtagningen, logga den egna och Broadmarks tidtagning till textfiler samt för att sammanställa resultat när valideringstestningen är slutförd.

### validering Resultat



1. Resultatet av valideringstestningen

Resultatet av valideringstestet kan ses i **figur 12** och det går att från det dra slutsaten att Broadmark uppfyller både kravet för reproducerbarhet och verifierbarhet. Den egna uppmätta tiden skiljer sig från Broadmarks med endast mikrosekunder. Med en sådan liten skillnad kan de i stort anses vara lika och därmed kan verifieringsbarhet sägas vara uppfylld. Avrundas till två decimaler, som Serpa & Rodrigues(2019b s.12) gjorde, uppmätte både Broadmark och den egna tidtagningen 0.01 sekunder som medelvärde, vilket är exakt det värde som Serpa & Rodrigues(2019b) uppmätte. Detta i kombination med att variansen endast uppgår till 2 mikrosekunder påvisar att Broadmak uppfyller kravet för reproducerbarhet.

## Modifieringen av simulationsgeneratorn

Kort förklaring av vad det var som gjordes, referera till apandexA

## Implementation av hjälpskript

Kort förklaring av vad det var som gjordes, referera till apandexb

## Implementeringen av BVH-SR

*Visual Studio 2015*

*CUDA Toolkit 10.2* (2007)

Integration utav BVH-SR underlättades av att Wang, Tang, Manocha & Tong (2018b) publicerat dess källkod på internet. Detta gjorde att istället för att försöka imitera deras lösning så var det möjligt att använde den, om än något modifierad för att få den att fungera i Broadmark. Det antogs att integerineng utav BVH-SR i Broadmark skulle vara trivial då källkoden fanns tillgänglig, det visade sig dock vara en mycket problematiskt uppgift. Faktum är att just implemetnationav BVH-SR tog mångfaldit mycket mer tid än vad som var tänkt.

### Vad som gjordes för integreringen

Gjorde rapperskripten BVH\_SR\_RUN och BEVH\_SR\_ENTRY. Adderade funktionalitet för ersatte planes med aabber

För integrationen så var bvh\_sr tvungen att ha en funktion som returnerade antalet funna överlapnings par. Orignalt användes defult cashen och övelapnings sökningen kopierades från den andra gpu lösningen.

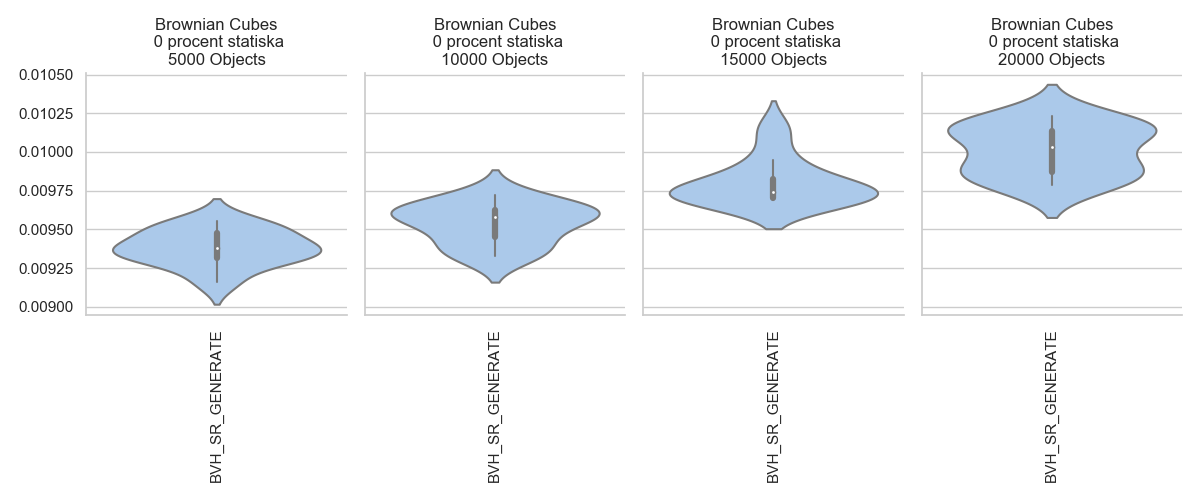
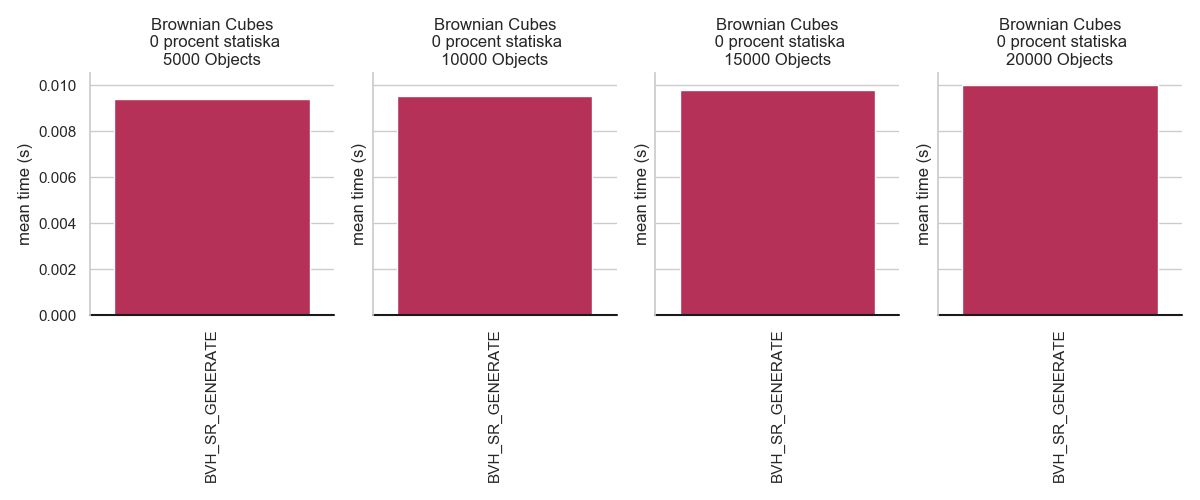
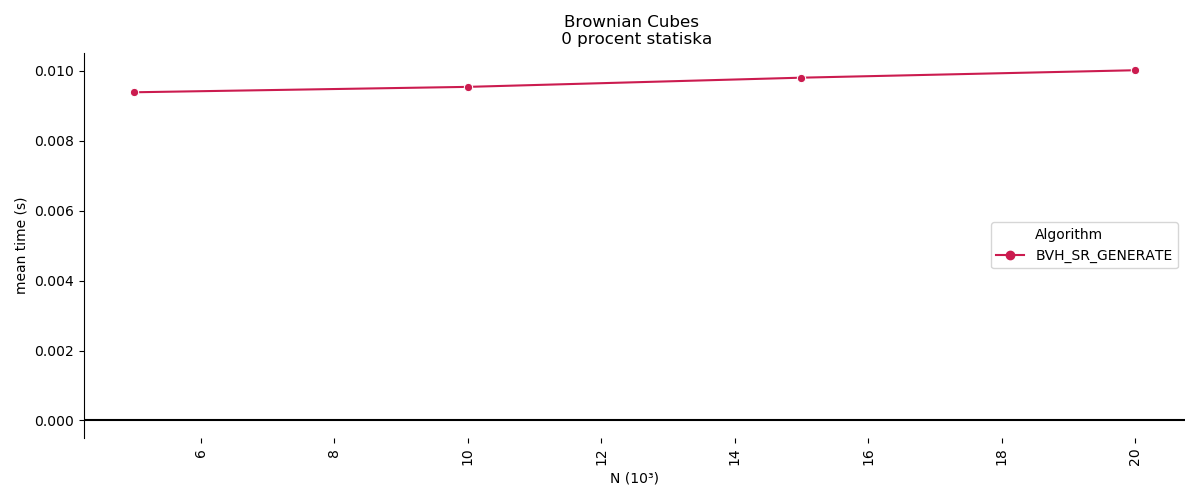
### Problemen som uppstod under arbetet med integereingen.

Det fanns dock ett par problem, källkod Wang, Tang, Manocha & Tong (2018b) publicerade innehöll inte endast BVH-SR algoritmen

### Validering av BVH-SR

Testscenariona som Wang, Tang, Manocha & Tong (2018a) använde är alla från *UNC Dynamic Scene Benchmarks collection* (Gamma, 2012). Problematiskt nog så är ingen av dem exakt matchningen mot de scenarion som finns i Broadmark, både hur de är utformade men även då de innehåller deformerbara kroppar. De test som dock liknar varandra mest är slumpvandringen från Broadmark och *N-body* (Gamma, 2012), där av det slumvandringen som kommer användas för testningen.

Antalet objekt som ska användas i testningen är även det inte en perfekt matchningen mot testet Wang, Tang, Manocha & Tong (2018a) använde.N-body:s objekt består av hundratals sfärer och fem koner dock finns inga exakta antalet sfärer som användes beskrivet, utan endast att simulation bestod utav 146000 trianglar. Det beslutades att grovt approximera antalet objekt genom della antalet trianglar med åtta vilket resulterar i 18250 stycken objekt. Denna approximation valdes då det grovt delar upp den potentiella kollisionsytan i N-bodys alla tringlar på en motsvarande kollisionsyta för aabb:er. Med detta som grund beslutades det att testa ett objekt intervall som börjar på 5000 och går upp till 20000 med 5000 steg. Valet att testa ett intervall istället för att bara testa de beräknade antalet objekt togs för få en bild av hur BVH-SR prestanda utvecklas när antalet objekt ökar. För valideringen beslöts det även att använda kuber istället för sfärer då det som tidigare notera,3.1.4, inte är någon större skillnad mellan kuber och sfärer i Broadmark då båda använder aabb gränsvolymer för överlappningstestningen. Sist ska det nämnas att andel statiska objekt är noll för testerna då det inte finns någon tydlig grund för N-body innehöll några statiska objekt.

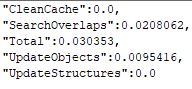


1. Resultat från testningen

Wang, Tang, Manocha & Tong (2018a, s. 236) Utförde två tester med N-body med två olika grafikprocessorer:

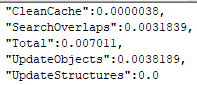
* GTX 780, resultat 10.5 mikrosekunder,
* GTX 1080, resultat 4.3 mikrosekunder.

Valideringsresultatet, **figur 13**, för det tidigare beräkna antalet objekt är ungefär 10 mikrosekunder. Detta är när på identiskt med resultatet Wang, Tang, Manocha & Tong (2018a) presenterar för testet de gjorde på GTX 780:in, dock är det problematiskt. Valideringstestet gjordes nämligen på en dator utrustad med en GTX 1060 vilken borde resultera i prestanda mycket närmare GTX 1080. Närmare undersökning visade problem låg i överlapningssöknings funktionen som implementerats, som kan ses i **figur 14** så går en majoritet av tiden åt till överlappningssökningen. Trots att implementation av BVH-SR har tagit markant mycket mer tid än tänkt var detta något som var tvunget att åtgärdas. Det som drar ner prestandan är nämligen en något som adderats för Broadmarkintegrationen, och det är inte rätt att bedöma BVH-SR:s prestanda utifrån ett resultat som den inte har att göra med.

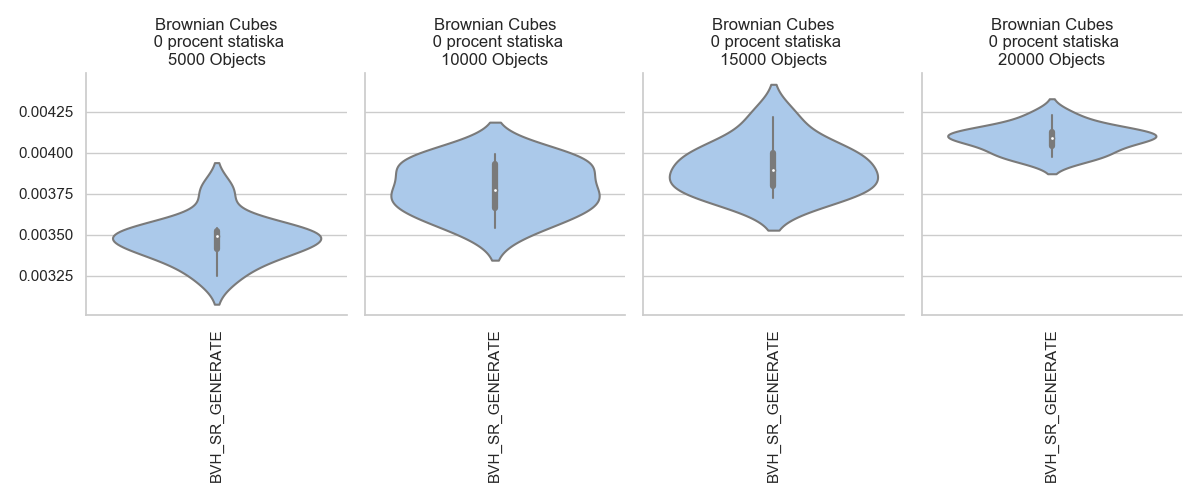
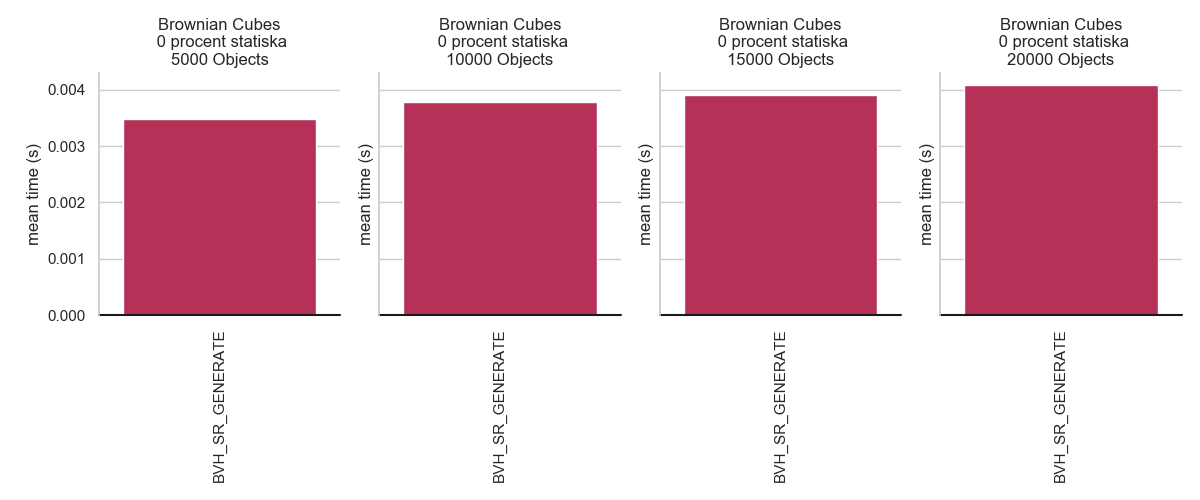
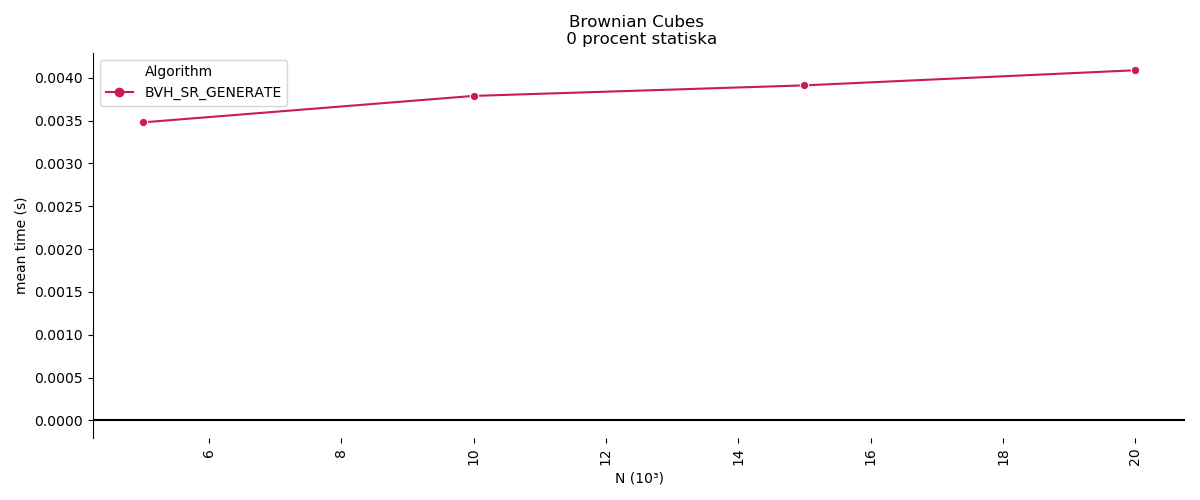


1. Data från en uppdatering av testet. Notera att tiden för överlappnings är 59 procent av den totala tiden och 3.2 gånger större än tiden för uppdatering av

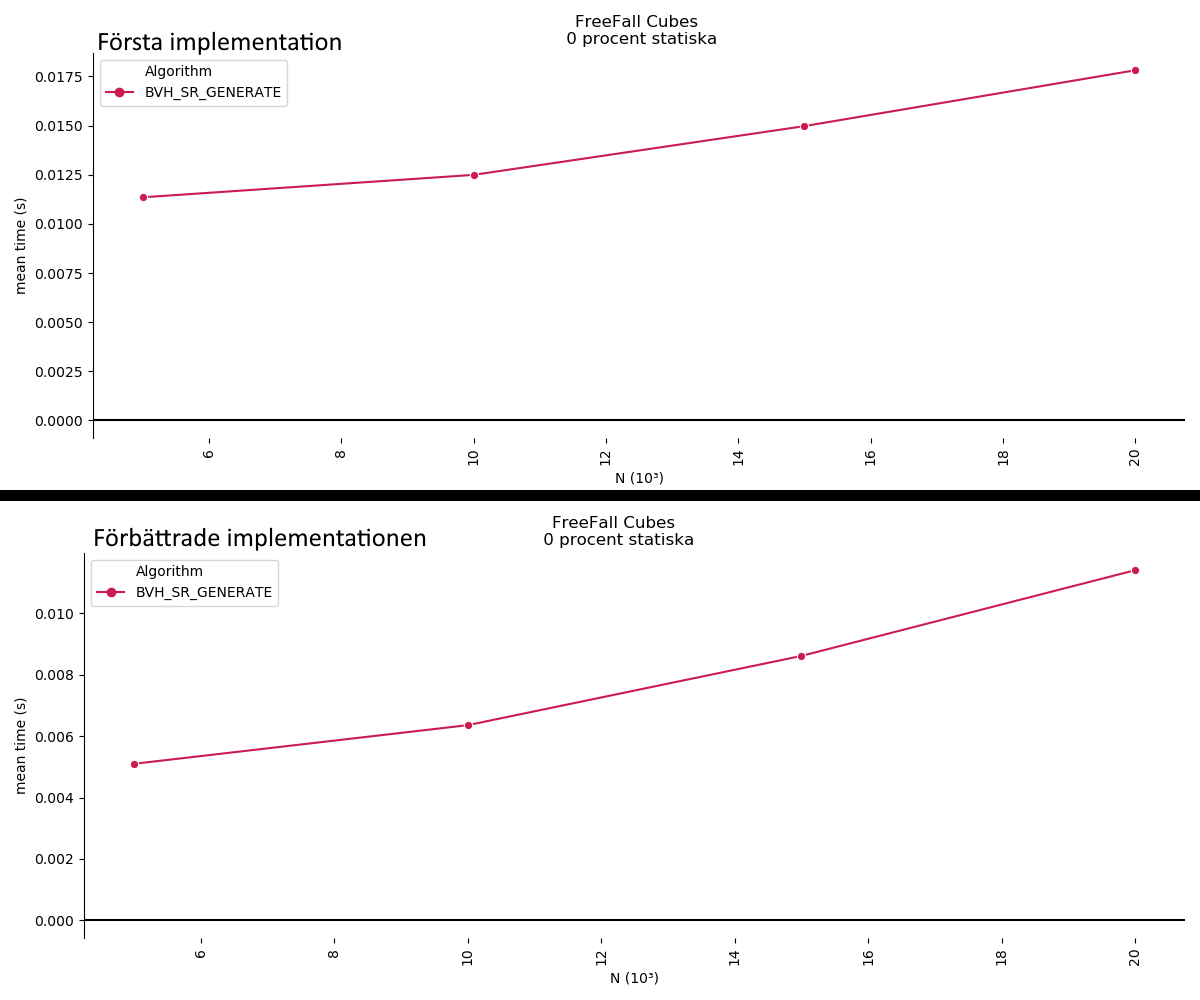
Arbetet med förbättringen av överlappningssökning avhandlas **Appendix C**, men kortfattat visade det sig att BVH-SR inte behövdes någon extra överlappningssökning. BVH-SR skapade nämligen en lista med alla överlappnings par när den uppdatera BVTT-fronten. s



1. Data från en uppdatering av testet efter förbättringen.



1. Resultat från testningen efter förbättringen.



1. Jämförelsen av prestanda före(överst) och efter förbättringen(nederst)

1 Förklarar rakt uppoch ner vad det var som behövdes göras

2 Förklara att det var mycket problematiskt och drog utpå tiden.

2.1 fel vestrion vs, inte viktigt problem men gav en till felkäla

2.2 första gången biblotekt användes på detta sättet i vs

2.3 första gången cuda användes

2.4 svårt att särkilja de nödvändiga bitarna av algortimen från mätninganra

2.5 två vestioner, började titta på fel

2.6 påbörjade en engen modifeirad variatn som riktade in sig mot rigida kroppar (visa bilder)

2.7 första försöket med interface klass (bvh\_sr som bordmark anväde) fungerade inte pga kuda och lognings klasserna initsierades inte på rätt sät.

(fixa bilder på första andra versione, samt strukt diagram på deras start upp, min första version och den slutliga vrsion)

2.8 fick problem med att den när den väl funkade plötsligt krackade vid ett vist antal bojekt, visade sig att antallet objekt som den kunde testa (fronten var för liten, fik expandera den, tetsade dock med upp mot 2000.

3 understryck att nu i efethan verkar dessa problem bannala men att då alal problem var tvunga att ahenteras i stort set samtidigt, samt då helt enkeöt erfarenhetetn med att jobbe med denna sortetns arbete saknades, så många gåner vista jag itne vilken fråga som skulle ställas för att få fram rätt svar, vilket försvårade felsökningen

## Implementeringen av PSM\_SAP

Förklarar att kälkoden inte var tillgängleig för denna så arbete började med att adapatera denutifrån ppesudokoden och beskrivning i raporten, samt att det visade sig att de hade utvecklat denna algoritm över loppet utav tre raporter Capannini & Larsson(2016a, 2016b, 2018)

Referara bost hemsidan, visa bilder på icke färdiga koden

*Boost* (1999)

Capannini & Larsson(2016a) första, succtree

Capannini & Larsson(2016b)

Capannini & Larsson(2018)

## Pilot studie

## Research / Förstudie

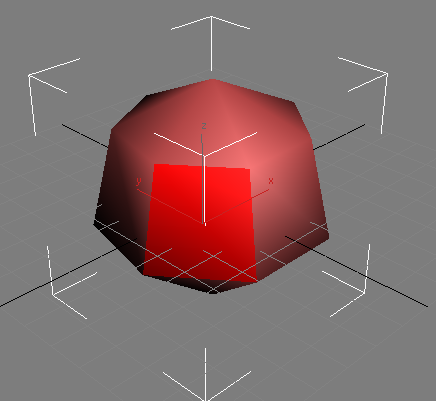
[Ett research- eller förstudiekapitel kan vara viktigt. Om man inspirerats av något, exempelvis ett program, en film, en bok eller något annat så kan detta listas under denna rubrik.

Källor som underbygger genomförandet men inte problemet eller metoden kan ligga i bakgrunden, men de passar ofta bättre i ett separat kapitel i början av genomförandedelen än som en del av bakgrunden.

Så mycket som möjligt av det som inspirerat genomförandet ska listas i detta kapitel]

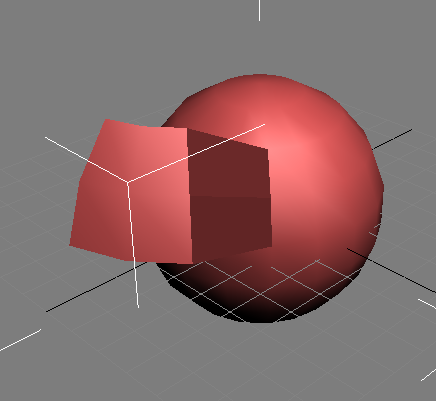
## Progressionsexempel: modellering

[Exempel: progression och dessignval inom området grafik. Till en början användess ett sfäriskt utgångsobjekt med få hörn. Detta gav tyvärr ett alltför kantigt utseende, se figur 3.



1. Ett sfäriskt objekt med få hörn

Alternativet blev då att använda ett objekt som hade fler hörn, se figur 4. De främsta nackdelarna med att använda så många hörn är att man får poler där hörnen går samman i toppen och botten av objektet samt att det ger lägre skärmuppdateringshastighet på grund av fler hörn i modellen.]



1. Ett sfäriskt objekt med många hörn

# Utvärdering

[Kapitlet *utvärdering* ska innehålla en presentation av den genomförda undersökningen, en analys av utfallet och de slutsatser som kan dras därav.

Kapitlet kan variera ifråga om struktur beroende på projektets utformning.

]

## Presentation av undersökning

## Analys

## Slutsatser

# Avslutande diskussion

## Sammanfattning

[

Här sammanfattas rapporten som helhet från frågeställning till slutsats. En läsare som vill få en snabb överblick av uppsatsen ska kunna gå direkt från kapitel ett till kapitel sju.

]

## Diskussion

[I diskussionskapitlet sätts problemet och resultatet i ett större sammanhang utanför examensarbetets specifika problemformulering. Det är viktigt att koppla till andras arbete - till exempel centrala artiklar som används i bakgrundskapitlet.

Diskutera resultatets trovärdighet. Lyft fram faktorer som påverkar trovärdigheten. Diskutera utifrån den genomförda studien.

Diskussionskapitlet är även en lämplig plats för att ta upp de saker som täcks av kriteriet ”Rapporten innehåller en diskussion kring relevanta samhälleliga och etiska aspekter på arbetet”. Välj några av nedanstående aspekter och diskutera dem i relation till ditt arbete. Beroende på typen av arbete varierar vilken typ av aspekter som är relevanta att undersöka, om man till exempel gjort ett spel eller en illustration som innehåller människor kan genus och kulturella aspekter vara intressant medan samhällelig nytta blir intressant om man har utvecklat en algoritm eller ett program.

* Etiska aspekter
* Forskningsetiska aspekter hos arbetet eller undersökningsmetoden
* Samhällelig nytta hos arbetet
* Genus och/eller kulturella aspekter …… mit liv som progare: hur fan ska jag assosiera det här med 3002 rader c++ kod?

]

## Framtida arbete

[I det framtida arbetet ska en hypotetisk fortsättning på examensarbetet diskuteras. Detta gäller både i det korta perspektivet - om arbetet skulle fortsätta några extra dagar eller månader. Projektet ska även sättas in i ett större sammanhang och ses ur ett bredare perspektiv. Till exempel om ett företag skulle kunna fortsätta på examensarbetet och driva resultatet vidare för att bygga ett komplett spel. Detta kan med fördel relateras till det större sammanhanget som diskuteras i diskussionsdelen.]

Referenser

Avril, Q., Gouranton, V. & Arnaldi, B. (2014) Collision detection: Broad phase adaptation from multi-core to multi-GPU architecture. *Journal of Virtual Reality and Broadcasting*, Digital Peer Publishing NRW, 2014, 6 (11), ss.1-13. hal-01018759v2.

Capannini, G. & Larsson, T. (2018) Adaptive Collision Culling for Massive Simulations by a Parallel and Context-Aware Sweep and Prune Algorithm. *IEEE transactions on visualization and computer graphics*.24(7), ss. 2064–2077. doi: 10.1109/TVCG.2017.2709313.

Coumans E. (2018) btdbvtbroadphase. github.com/bulletphysics/bullet3/blob/master/src/BulletCollision/BroadphaseCollision/btDbvtBroadphase.h[2020-02-20]

Du, P., Zhao, J.-Y., Pan, W.-B. & Wang, Y.-G. (2015) Gpu Accelerated Real-Time Collision Handling in Virtual Disassembly. *Journal of Computer Science and Technology*. 30(3), ss. 511–518. doi: 10.1007/s11390-015-1541-2.

Ericson, C. (2004). *Real-time collision detection*. Amsterdam: Elsevier.

Fukuhara, A., Tsujita, T., Sase, K., Konno, A., Jiang, X., Abiko, S. & Uchiyama, M. (2014) Proposition and Evaluation of a Collision Detection Method for Real Time Surgery Simulation of Opening a Brain Fissure. *ROBOMECH Journal*. 1(1), pp. 1–14. doi: 10.1186/s40648-014-0006-7.

Friston, S. & Steed, A. (2019) Real-Time Collision Detection for Deformable Characters with Radial Fields. *IEEE transactions on visualization and computer graphics*. 25(8), ss. 2611–2622. doi: 10.1109/TVCG.2018.2859924.

Geleri, F., Tosun, O. & Topcuoglu, H. (2013) Parallelizing Broad Phase Collision Detection Algorithms for Sampling Based Path Planners. *Proceeding of the 2013 21st Euromicro International Conference on Parallel, Distributed, & Network-Based Processing*. Belfast, UK, 27 Feb.-1 mars 2013, ss. 384-391. doi: [10.1109/PDP.2013.62](https://doi.org.libraryproxy.his.se/10.1109/PDP.2013.62)

Luque, R., G., Comba, J., L., D. & Freitas C., M., D., S. (2005) Broad-phase collision detection using semi-adjusting BSP-trees. *Proceedings of the 2005 symposium on interactive 3D graphics and games*. ACM, ss. 179–186

Li, C., Ding, J., Hong, Z., Pan, Y. & Liu, P.X. (2018) A Surface Mass-Spring Model with New Flexion Springs and Collision Detection Algorithms Based on Volume Structure for Real-Time Soft-Tissue Deformation Interaction. *IEEE access*. 6, ss. 75572–75597. doi: 10.1109/ACCESS.2018.2883679.

Serpa, Y. R. & Rodrigues, M. A. F. (2019a) Flexible Use of Temporal and Spatial Reasoning for Fast and Scalable Cpu Broad-Phase Collision Detection Using Kd-Trees. *Computer Graphics Forum*. 38(1), pp. 260–273. doi: 10.1111/cgf.13529.

Serpa, Y. R. & Rodrigues, M. A. F. (2019b) Broadmark: A Testing Framework for Broad‐phase Collision Detection Algorithms. *Computer Graphics Forum*. (2019-10-24), doi: 10.1111/cgf.13884.

Schmidtke, R. & Erleben, K. (2018). Chunked Bounding Volume Hierarchies for Fast Digital Prototyping Using Volumetric Meshes. *IEEE transactions on visualization and computer graphics*. 24(12), ss. 3044–3057. doi: 10.1109/TVCG.2017.2784441.

Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M.-P., Faure, F., Magnenat-Thalmann, N., Strasser, W. & Volino, P. (2005) Collision Detection for Deformable Objects. Computer Graphics Forum. 24(1), ss. 61–81.

Tracy, D. J., Buss, S. R. & Woods, B. M. (2009) Efficient largescale Sweep and Prune methods with AABB insertion and removal. In *Proceedings of the 2009 IEEE Virtual Reality Conference,* Lafayette, LA, USA, IEEE, ss. 191–198.

Tian, Y., Hu, Y. & Shen, X. (2019) A Multi-Gpu Finite Element Computation and Hybrid Collision Handling Process Framework for Brain Deformation Simulation. *Computer Animation and Virtual Worlds*. 30(1). doi: 10.1002/cav.1846.

Wang, X., Tang, M., Manocha, D. & Tong, R. (2018a). Efficient Bvh-Based Collision Detection Scheme with Ordering and Restructuring. *Computer Graphics Forum*. 37(2), ss. 227–237. doi: 10.1111/cgf.13356.

Weller, R., Debowski, N., & Zachmann, G., (2017) Kdet: Parallel Constant Time Collision Detection for Polygonal Objects. Computer Graphics Forum, 36(2), ss.131–141. doi: 10.1111/cgf.13113.

Zhang, X. & Liu, Y. (2015) A Fast Algebraic Non-Penetration Filter for Continuous Collision Detection. *Graphical Models*, 80, pp. 31–40. doi: 10.1016/j.gmod.2015.06.001.

Zou, Y., Liu, P. X., Yang, C., Li, C. & Cheng, Q. (2017) Collision Detection for Virtual Environment Using Particle Swarm Optimization with Adaptive Cauchy Mutation. *Cluster Computing : The Journal of Networks, Software Tools and Applications*. 20(2), ss. 1765–1774. doi: 10.1007/s10586-017-0815-6.

Kistostwski, K., K., Arnold, J., A., Huppler, K., Lange, K., D., Henning, J., L. & Cao, .P (2015) How to Build a Benchmark. *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering(ICPE ’15)*. New York, USA, 27 Feb.-1 januari 2015, ss. 333–336. doi: https://doi.org/10.1145/2668930.2688819

*Boost* (1999). Version 1.72.0. [c++ programvarebibliotek] . Boost organization. Tillgänligt på internet: https://www.boost.org/

*CGAL* (1995). Version 5.0.2. [programvarebibliotek för geometriska beräkningsalgoritmer] The CGAL Project. Tillgänglig på Internet: https://www.cgal.org/index.html

*Vcpkg* (2016). Version 2020.02.04-nohash. [pakethanterare] Microsoft. Tillgänglig på Internet: https://github.com/Microsoft/vcpkg

*CUDA Toolkit 10.2* (2007) version 10.2. [programvaruverktyg som ger stöd för programvaru utveckling på GPU:er med CUDA] Nivida Corporation. Tillgängligt på Internet: https://developer.nvidia.com/cuda-downloads

*Visual Studio 2017*. (2017). Version 15.9.22. [programutvecklingsmiljö] Microsoft. Tillgängligt på internet: https://visualstudio.microsoft.com/vs/older-downloads/

*Visual Studio 2015*. (2015). [programutvecklingsmiljö] Microsoft. Tillgängligt på internet: https://visualstudio.microsoft.com/vs/older-downloads/

*Notepad ++.* (2003).version7.8.6. [texteditor] Ho .D. Tillgängligt på internet: https://notepad-plus-plus.org/downloads/

*Unity 2019.2* (2019). Version 2019.2.0f1*.* [spelmotor] Unity Technologies. Tillgängligt på internet: https://unity3d.com/get-unity/download/archive

*Pandas* (2008). Version 0.25.3. [data analyserings och manipulations bibliotek] Mckinney .W. Tillgängligt på internet: https://pandas.pydata.org/

*Seaborn* (2013). version 0.9.0. [visualiserings program för statiskt data] Waskom .M. Tillgängligt på internet: https://github.com/mwaskom/seaborn

Wang, X., Tang, M., Manocha, D. & Tong, R. (2018b). *Efficient Bvh-Based Collision Detection Scheme with Ordering and Restructuring*. [källkod]. Tillgängligt på internet: https://github.com/littlemine/BVH-based-Collision-Detection-Scheme

Serpa, Y. R. & Rodrigues, M. A. F. (2019c) Broadmark. [källkod]. Tillgängligt på internet: https://github.com/ppgia-unifor/Broadmark

Hintzer, J., L. & Nelson, R., D.(1998). *Violin Plots: A Box Plot-Density Trace Synergism*. The American Statistician. 52 (2): 181–4. doi:10.1080/00031305.1998.10480559.

Capannini, G. & Larsson, T. (2016a) Efficient collision culling by a succinct Bi-dimensional sweep and prune algorithm. *Proceedings - SCCG 2016: 32nd Spring Conference on Computer Graphics*. Smolenice Castle, Slovakien, 27 April 2016 till 29 April 2016, ss. 25 – 32. doi:10.1145/2948628.2948640

Capannini, G. & Larsson, T. (2016b) Adaptive Collision Culling for Large-Scale Simulations by a Parallel Sweep and Prune Algorithm. *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization EGPGV 2016, 6 juni.* Groningen, Nederländerna: Eurographics Association , s. 1-10. doi: 10.2312/pgv.20161177

Gamma. (2012). *UNC Dynamic Scene Benchmarks*. http://gamma.cs.unc.edu/DYNAMICB/ [2020-05-23]

Upphovsman(år för  
senaste uppdatering). *Titel*. URL [Datum för åtkomst]

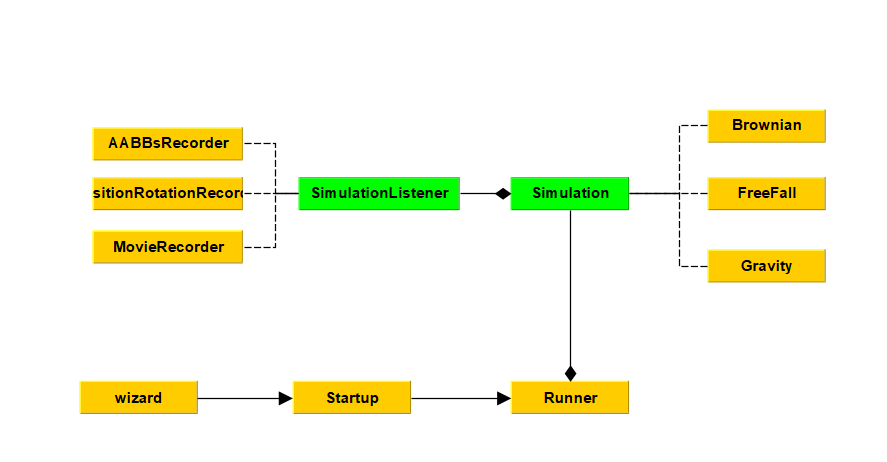
Namn. (datum det släptes). Version number. Bygg nummer. Vad det är. Förfatare. Tillgänglig på internet: ip address.

1. Simulationsgenerator.

Simulationsgeneratorn är implementerad i *Unity 2019.2* (2019) vilket är samma program som användes för vidareutvecklingen av den. Koden associerad med simulationsgeneratorn är skriven i c# då det är det språket som *Unity 2019.2* (2019) stödjer och editeringen av kod kommer göras i *Visual Studio 2017*.

Addering av statiska objekt

Som nämnt i 3.1.6 var idén att skapa kopior utav de tre scenerna som Serpa & Rodrigues (2019b) inkluderade i Broadmark och modifiera dem för att kunna ställa in andelen statiska objekt. Detta var det mest logiska tillvägagångs sättet eftersom Serpa & Rodrigues (2019b) designade generatorn just för att det skulle vara enkelhet att addera scenarion, samt då det är i scenariona som objekten skapas. Dock konstruerade Serpa & Rodrigues (2019b) simulationsgeneratorn så att testscenariona implementerar ett standardiserat *interface* vilket används av en körningsklass, som senare skapar scenariona utifrån de parametrarna som valts, se **figur 1**. Genom *Interface* har körningsklassen tillgång till alla objekt i scenariona, så att ett snabbare och enklare sätt att implementera de statiska objekten är att i körklassen, direkt efter att scenariot skapats, ta och sätta den önskade andelen objekt till att vara statiska. De statiska objektens färg ändras även till röd för att underlätta urskillningen mellan dem och de dynamiska objekten.

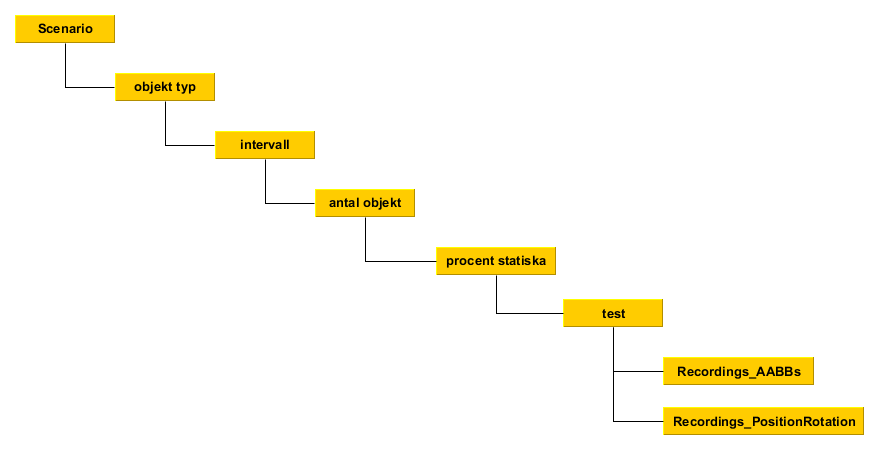


1. Förenklat strukturdiagram som visar de för undersökningen relevanta delarna av simulationsgeneratorn, gröna noderna är *interface* och de noder som är anslutna till de gröna noderna med streckade linjer är klasser som implementerar det *interface*.

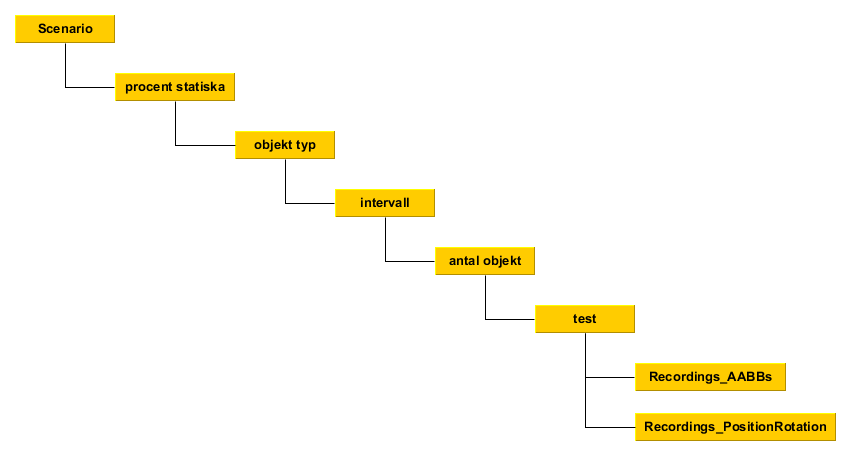
Automatisering av testerna

Till följd av den stora andelen testscenarion som ska generas beslutades det att expandera simulationsgeneratorns funktionalitet till att kunna, med utgångspunkt i givna startparametrar, automatiskt genera testserier utifrån det format som definierades i metodbeskrivningen, 3.1.0 till 3.1.4, och spara ner i filhierarki. Detta kommer göra att uppsättningen av simulationsgeneratorn kommer att ta längre tid, dock är det tid som snabbt kommer att återhämtas när testningen väl genomförs.

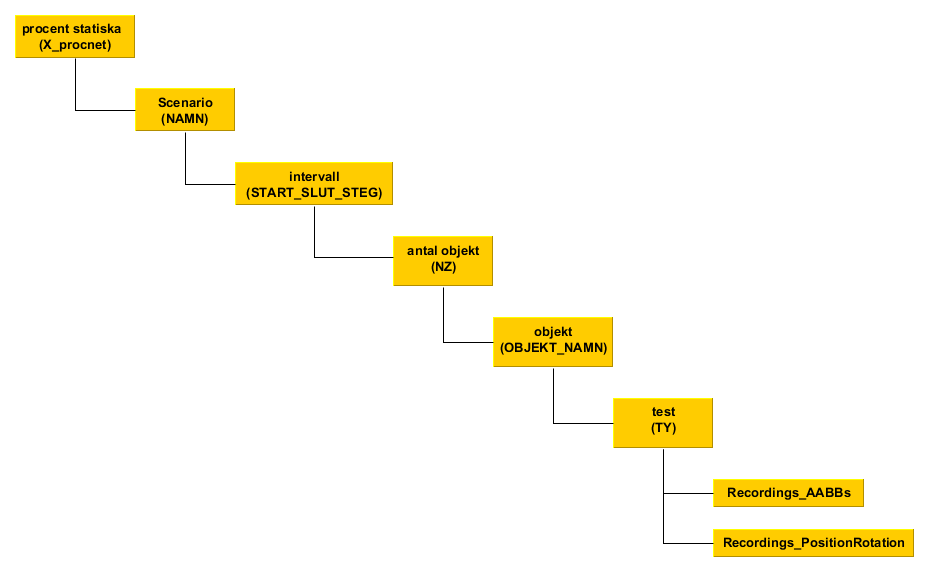
Det första som beslutades var utövningen av filhierarki till följd av att implementeringen av resterande modifikationer kommer att vara beroende på hur den utformas Tanken med filhierarki är att underlätta användning av informatio som genereras, vilket resulterade i att den reviderades ett par gånger. Första iteration visade sig vara sig vara svårhanterlig när den skulle användas i praktiken och andra iterationen var däremot svårimplementerad utifrån hur scenariona skapades. De iterationer som filhierarkin genomgick kan ses i **figur 2** till **figur 4**.



1. Första iterationen av filhierarkin



1. Första iterationen av filhierarkin

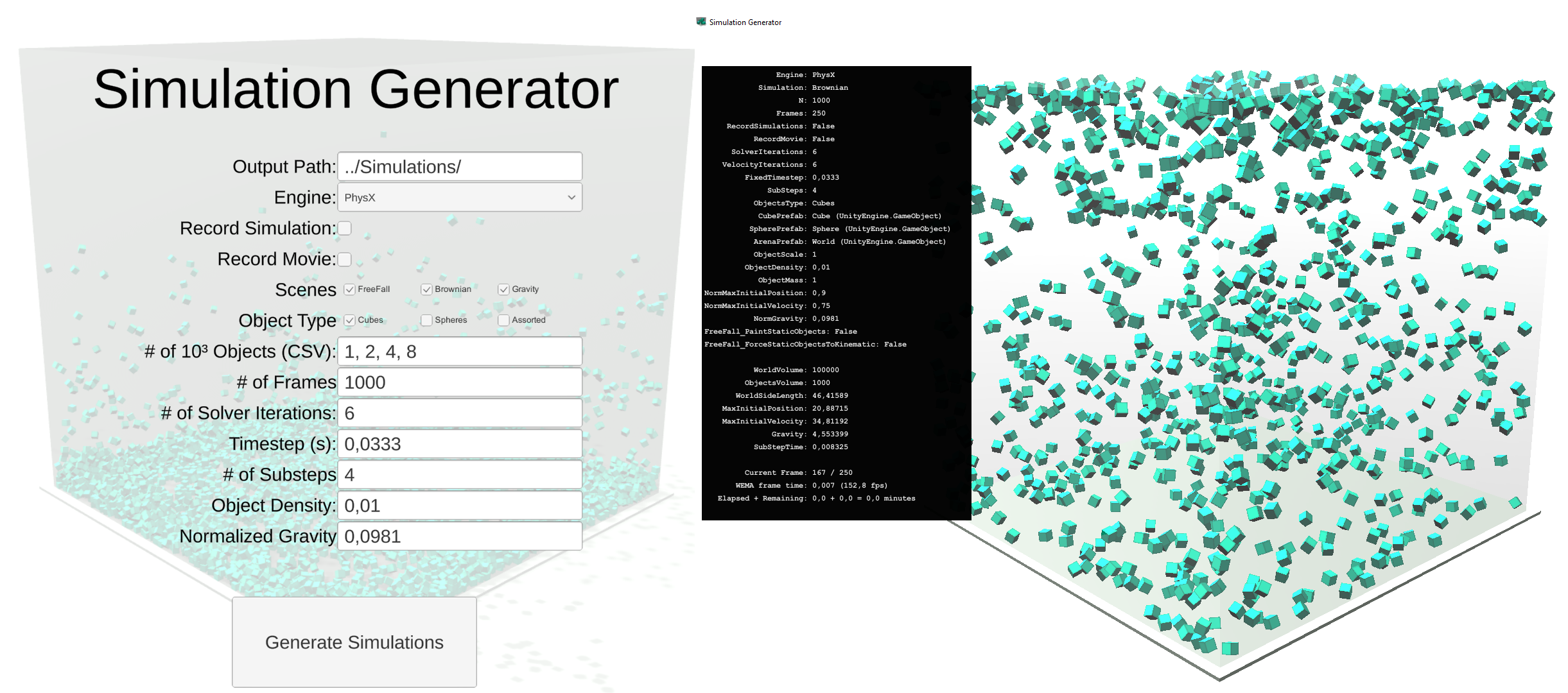


1. Slutliga versionen av filhierarkin som den modifierade simulationsgeneratorn sparar ner scenariona i. Namn standarden för varje del av hierarkin står i parenteserna, med undantag för de Recordings\_ mapparna.   
   Namn variablernas betydelse är följande: X procentuell andel statiska objekt, NAMN är scenariots namn, STAR intervallets start värde, SLUT intervallets slutvärde, STEG steglängden intervallet använder mellan START och SLUT, Z antalet objekt, OBJEKT\_NAMN är objekt typens namn, Y test index.

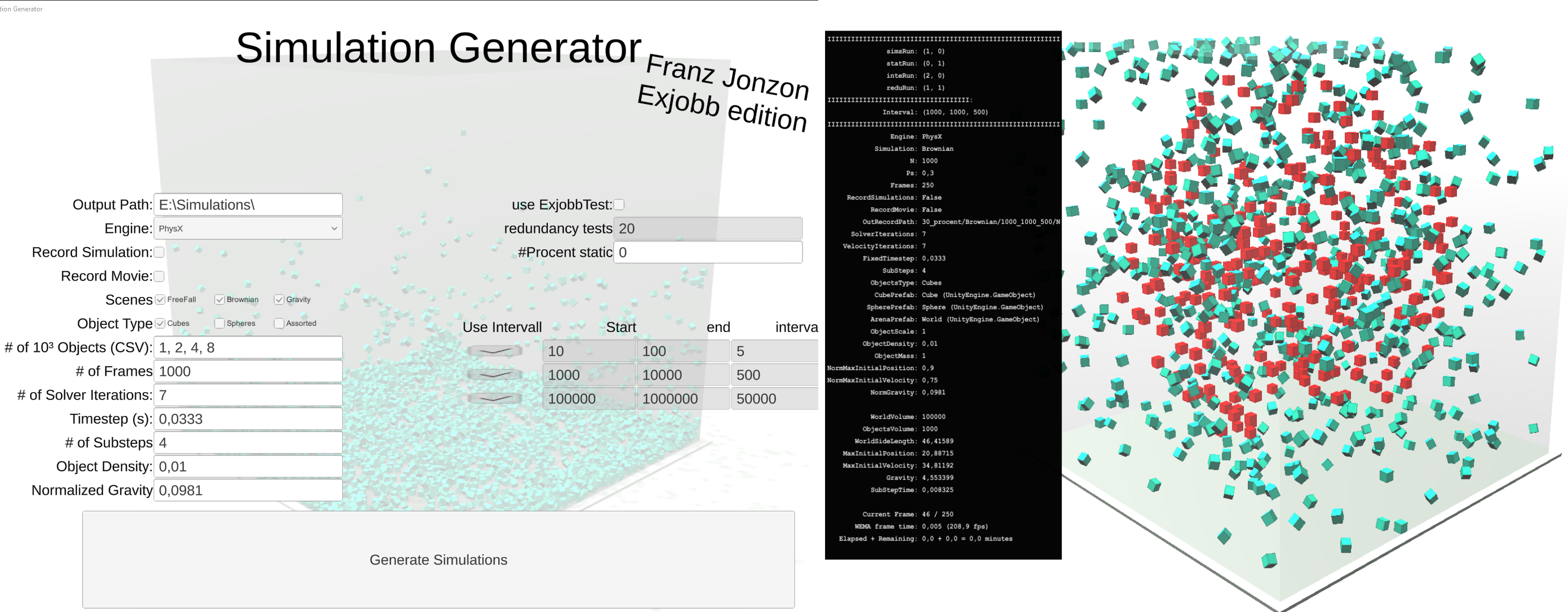
Efter filhierarkin adderas sedan de parametrar som behövdes för att automatisera testningen. Parametrarna som behövde adderas är,

* andelen gånger testerna ska upprepas samt,
* vilka förhållanden mellan statiska och dynamiska objekt som ska testas,
* tre intervallparametrar vilka består av ett startvärde, ett slutvärde samt steglängden. De adderas totalt tre uppsättningar av dess vilket gjordes för att kunna testa alla intervaller som definierades i 3.1.1 med en körning.

Utöver dessa adderades även av- och påvariabler så att för intervallparametarna, så ifall endast en intervall behöver prövas så kan de övriga två avaktiveras. Slutligen lades även en av- och påvariabel till så att det går att växla mellan den nya automatiserade och den ursprungliga simulationsgenerationen. Detta lades till för att snabbt kunna och enkelt generera enskilda testscenarion för vilka den automatiserade generatorn är överflödig. **Figur 5** visar på hur simulationsgeneratorns utseende ser ut innan dess funktionalitet expanderades och **figur 6** visar på dess utseende efteråt.



1. Simulationsgeneratorns huvudmeny och ett exempel på hur scenariogenereringen ser ut före modifieringen hade genomförts



1. Simulationsgeneratorns huvudmeny och ett exempel på hur scenariogenereringen ser ut efter modifieringen hade genomförts

Sparningen av andelen statiska objekt

I sin redogörelse över Broadmark förklarar Serpa & Rodrigues(2019b, s.6) att simulationsgenerator sparar ner de genererade simulationerna i ett filformat de kallar aabbs. De utvecklar att det som sparas ner i filen är scenen namn, antalet objekt, typ av objekt, antalet bilduppdateringar samt gränsvolymen för varje enskilt objekt för alla bilduppdateringar.

Utöver detta skulle expanderade version av simulationsgeneratorn även behöva spara hur stor andel av objekten som är statiska, eftersom det kommer vara nödvändigt vid utvärderingen hur väl olika algoritmer hanterade det specifika scenariot. Första idén var att addera detta värde till aabbs filformatet, dock övergavs detta relativa fort. Om ändringar gjordes av aabbs filformat skull nämligen Broadmarks inläsning av det behöva uppdateras. Detta skulle vara bortkastad tid då informationen inte används för själva testningen utan endas för att särskilja mellan resultaten. Information om andelen statiska objekt finns även redan nedlagrad i filhierarkin, se **figur 4**, då första uppdelningen är just av utifrån andelen statiska objekt. Beslutat togs där med att vid behov så kommer andelen statiska objekt hämtas från filhierarkin.

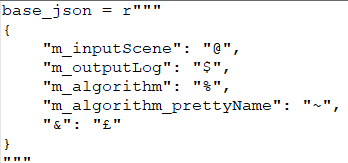
1. Hjälpskripten etc.

[Appendix ska fungera som referenslistan - dvs det ska finnas referenser till den från texten. Appendix ska inte vara numrerade utan ska namnges med: Appendix A, Appendix B osv. De ska vara sidnumrerade (I, II, III ...) men de ska inte finnas med i innehållsförteckningen. Varje nytt appendix ska börja på toppen av sidan.]

Utöver Broadmark och simulationsgeneratorn tillhandahöll Serpa & Rodrigues(2019c) även hjälpskripten *run* och *plot*, vilka båda är python skript. Serpa & Rodrigues(2019c) redogör att runär ett skript som automatiserar testningen genom att generera de testfiler, vars struktur kan ses i **figur 7**, som Broadmark använder. Testfilerna innehåller vägbeskrivning till testscenarion samt de algoritmen som ska testas mot det, ett sådant test kommer här efter refereras till som en *testsimulation*. Slutligen tar runtar .json filen som Broadmark producerar, för exempel se **figur 8**, och använder dataanalyserings och editerings verktyget *Pandas* (2008) för att formatera datan från dem på fyra olika sätt, varje formatering sparas ner i en .csv en .pickle samt en .xlsx fill. De resulterande fyra dataformateringarna är:

* *main\_frame*, innehåller tiderna för alla överlappningssökningar för alla testsimulationer
* *main\_described\_frame*, innehåller antalet objekt medelvärdet, standardavvikelsen, maximala, minimala, tiden 25 procent, tiden 50 procent samt tiden 75 procent utav överlappningssökningar för alla testsimulationer.
* *multi\_index\_frame*, innehåller en multindex som grupperar resultatdata utifrån scenariot och där efter utifrån antalet objekt som används.
* *lines\_frame*, innehåller medelvärdet utav tiden överlappnings sökning tog för alla testsimulationer.

Serpa & Rodrigues(2019c) förklarar att *plot* använder datavisualiseringsprogrammet *Seaborn* (2013) för att skapa diagram utifrån den formaterade datan. Diagrammen som skapas är ett linjediagram vilket visar hur prestandan för algoritmerna utvecklas i förhållande till antalet objekt i testscenariona, ett stapeldiagram som redogör för hur algoritmernas medelprestanda för de olika scenariona. Slutligen sammanställer de även ett violindiagram, vilket är en form av låddiagram, för att visa på spridningen utav algoritmernas medelprestanda. Noterbar är att plot endas redogör violindiagrammet för det första testscenariot. Exempel på de diagram *plot* skapar kan ses i **figur 12** till **figur 14**.



1. Formatet på .json filen som Broadmark använder för testningen



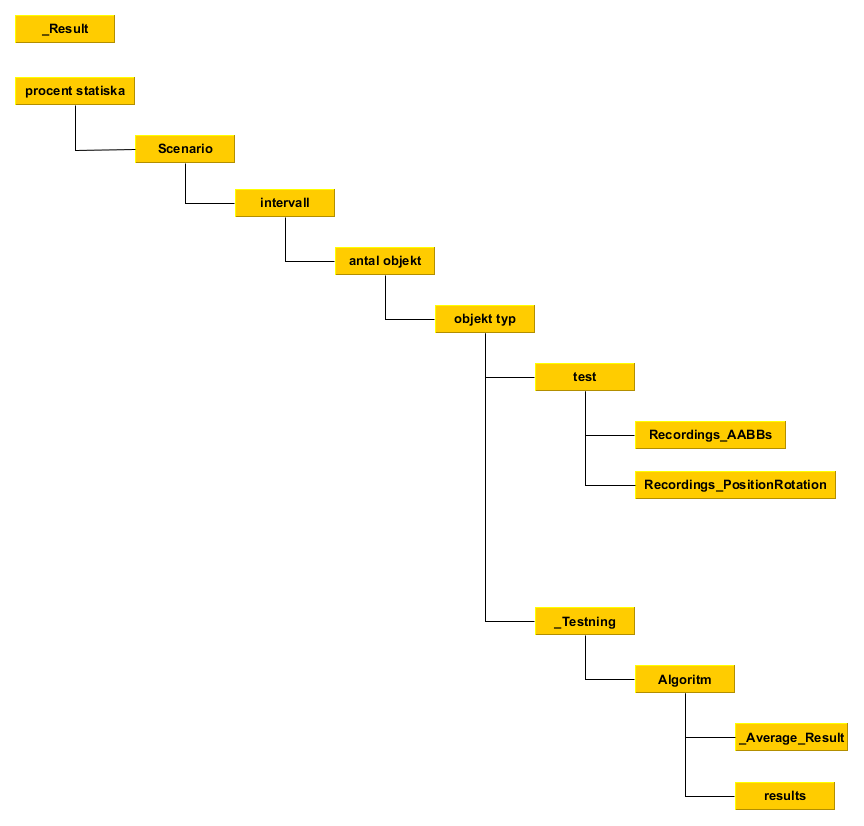
1. Exempel på .json filen innehållandes resultat från en testkörning av CGAL i Broadmarks.

Utifrån deras funktionalitet är run och plot mycket intressanta för denna undersökning. Run skulle underlätta testningen samt så sammanställer det resultatet och tar fram exakt den information som definierades i 3.1.4. Mer specifikt, stapeldiagrammet ger en bra inblick i hur algoritmerna presterar för olika mängder objekt, medan linjediagrammet ger en överblick över algoritmens prestanda utvecklas när antalet objekt ökar. Slutligen violindiagrammet visserligen inte är ett låddiagram men det är besläktat med det och visar samma information ett låddiagram gör.

Dock om run och plot ska användas så kommer det krävas efterforskning kring hur pythons syntax, *Pandas* (2008) samt *Seaborn* (2013) fungerar. Det finns en potentiell risk att de kommer ta för mycket tid att lära sig, om än lite, funktionalitet i två program bibliotek i ett nytt programmeringsspråk. Alternativ är att implementera samma funktionalitet i antingen C# genom att göra ett separat program som utför samma funktionalitet. Alternativt implementera samma funktionalitet med C++ direkt i Broadmark. Oavsett om C# eller C++ skulle väljas kommer det ändå behövas efterforskning för att hitta motsvarande programbiblioteket till *Pandas* (2008) och *Seaborn* (2013) , eller hitta information för att implementera dem från grunden. Utöver det tillkommer även att integrerar det med de redan existerande systemet. Python är dock ett programmeringsspråk vars grundprinciper bygger på lättläslighet, samt är det ett mycket populärt språk så det finns många resurser att hämta information ifrån och lika så är både *Pandas* (2008) och *Seaborn* (2013) mycket väldokumenterade. Det har även fördelen med att de bevisligen fungera i med ramverket och tillhandahåller den önskade funktionarlitet.

Slutsatsen drogs där med att det troligen skulle ta mins lika mycket tid att implementera samma funktionalitet som den som finns i run och plot i C# eller C++ och där utöver måst en sådan läsning även integreras i ramverket. Utifrån det beslutades det där med att vidareutveckla run och plot. För vidareutvecklingen skapades kopior utav run och plot som denoteras EJ\_run och EJ\_plot, det är dessa kopior som vidareutvecklades till att fungera med de expanderade systemet medan run och plot behölls för att ha som referens och för felsökning. Det ska noteras att både run och plot egentligen är uppdelade över flera skript: *plot*, *run*, *bm\_run*, *bm\_parse* och *bm\_plot*. För denna rapport ansågs det dock vara smidigare att endas referera till dem som två enskilda skript. Det kommer även skapas kopior av de övriga skripten, vilka kommer anpassa för att hantera den nya funktionalitet, vilka kommer namnges efter samma standar som kopiorna av run och plot. Dock för enkelhetens skull kommer all funktionalitet som är associerade med run eller plot, även om implementation utav dem är något av de andra skripten, beskrivas som om de är implementerade i dem. Slutligen all python kodediterades med hjälp av *Notepad ++* (2003).

Körnings skript

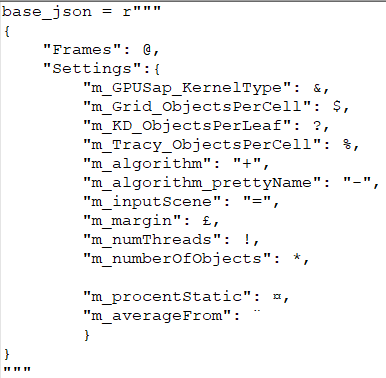


1. Filhierarkin efter att testningen har genomförts.

För att få EJ\_Run att ha den önskade funktionaliteten så kommer grunden i run behöva expanderas för att kunna utnyttja filhierarkin som simulationsgeneratorn skapat. Utöver de kommer dataformatering som run gör behöva modifieras för att ta hänsyn till den nya parametern i form av andelen statiska objekt.

Traverseringe av filhierarkin konstruerades med hjälp utav nästlade loopar, vilka går igenom filhierarkin och testar varje enskild test testscenario. Själva testningen tar även och expanderar filhierarkin i att under objekttyp så adderas en mappen *\_Testning* i vilket innehåller en mapp för varje algoritm som testat. I algoritmerna lagras av upprepningar utav den specifika testsimulationerna samt finns en mapp i vilka resultat från testningen lagarnas. Det finns även en mapp i vilket medelresultat vilket beräknats från alla upprepningar av testscenariot lagras. En kopia av medelresultat lagras även i enseparat \_Resultat mapp, vilken om inget annat anges planerats i hierarkins root, i vilket medelresultat för alla testsimulation lagras. Detta görs för att förenkla EJ\_plot så det inte ska behöva traverserna hela filhierarkin för att komma åt medelresultat. Utseendet av filhierarkin efter att testningen har utförts kan ses i **figur 9**.

Själva beräkningen av medelresultaten beräknades medhjälp av *Pandas* (2008) då de hade funktion som enkelt kunde beräkna medelvärdet får flera datakällor. För medelresultaten så expanderas settingsstrukturen, som kan ses i **figur 8**, för att utöver den redan existerande informationen även innehålla andelen statiska objekt för testet. Samt adderades information från hur många upprepningar som medelresultatet beräknats. Strukturen på den expanderade versionen kan ses i **figur 10**.



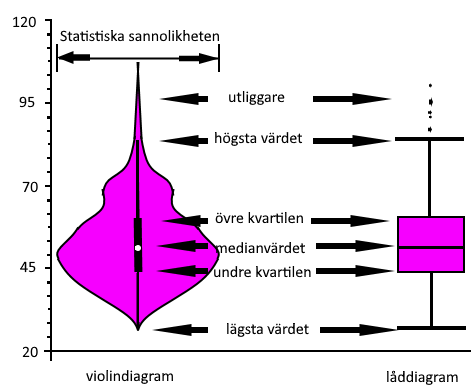
1. Expanderade version av settings strukturen som används för medelvärderesultat utav alla upprepningar utav ett specifik testsimulation. m\_procentStatic har adderats för lagra andelen statiska objekt och m\_averageFrom adderas för lagrat från hur många upprepningar som medelvärdet är beräknat ifrån.

Resultat visualiserar skriptet

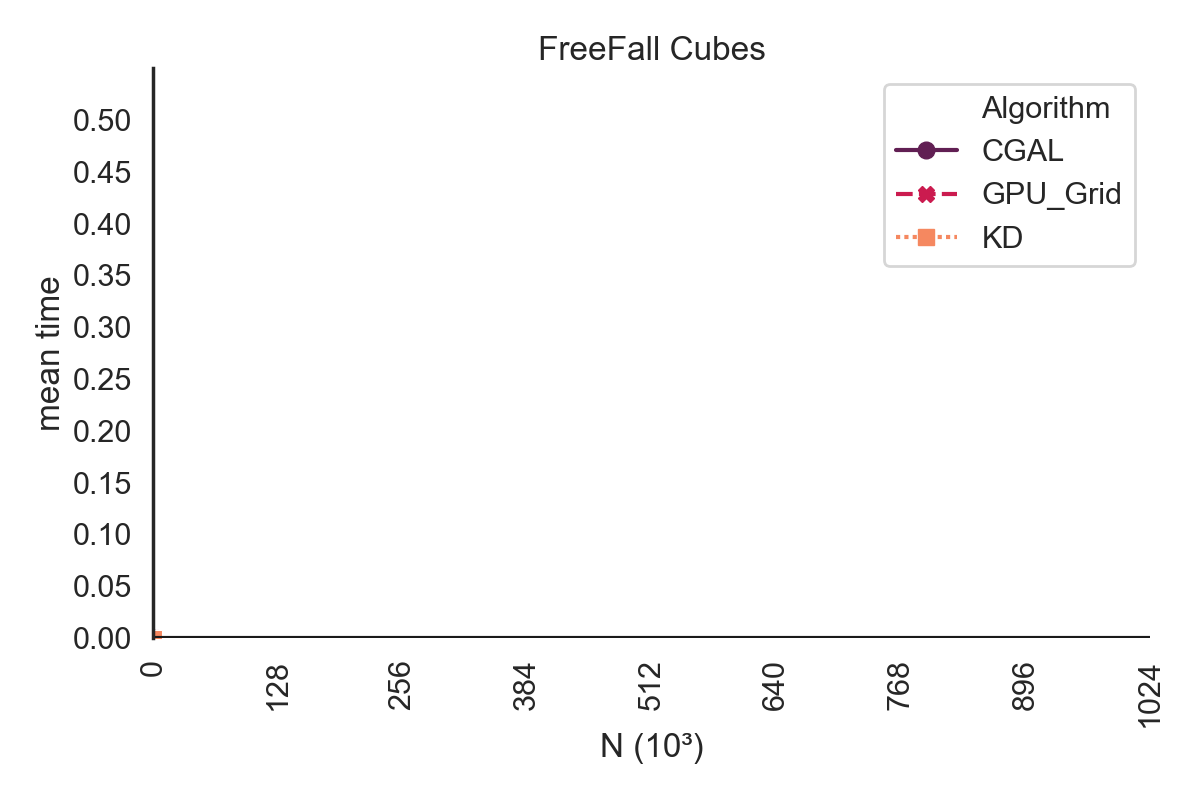
*Seaborn* (2013)

Ej\_plot krävde något mer efterforskning *Seaborn* (2013) och *Pandas* (2008) dock tack vare att de var mycket väldokumenterad var detta inte något som tog mer tid än nödvändigt. Valet gjordes att så långt som möjligt presentera relevanta diagram, exempelvis **figur 28** gjordes med EJ\_plot och vissas alla tester för ett scenario vilket kan jämföras plot som för motsvarande diagram hade presentera varje objekt mängd i individuella diagram som i **figur 25**.

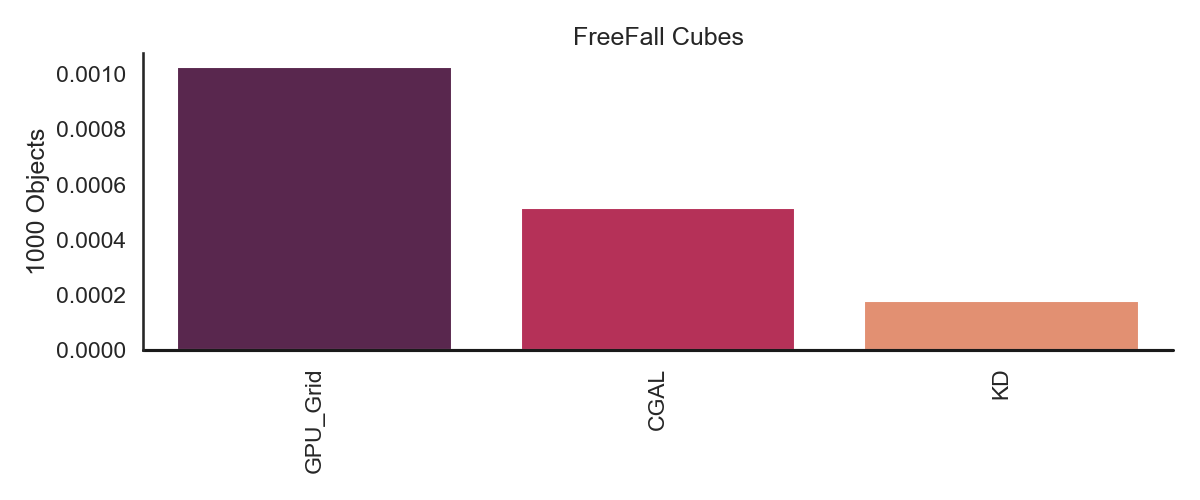
Utifrån tidigare beslut så konverterades violindiagrammet till ett låddiagram, se **figur 18** och **figur 19** för exempel. Vilket var enkelt då *Seaborn* (2013) även hade funktionalitet för att göra låddiagram, samt så hade låddiagram funktionen samma parameter violindiagrammet vilket gjorde modifieringen mycket simpel att implementera. Efter att lite efterforskning kring vad ett violindiagram faktisk var beslutade det dock att återgå till att använda det. Det visade sig nämligen att violindiagram är en kombination av låddiagram och *kernel density estimation*, violindiagrammet visar där med samma data som ett låddiagram gör plus att den använder en täthetsfunktion för att redogöra för sannolikhet för de olika värden Hintzer, J., L. & Nelson, R., D.(1998, s.181). för att se skillnaden mellan jämför **figur 18** och **figur 19** med **figur 20** och **figur 21**



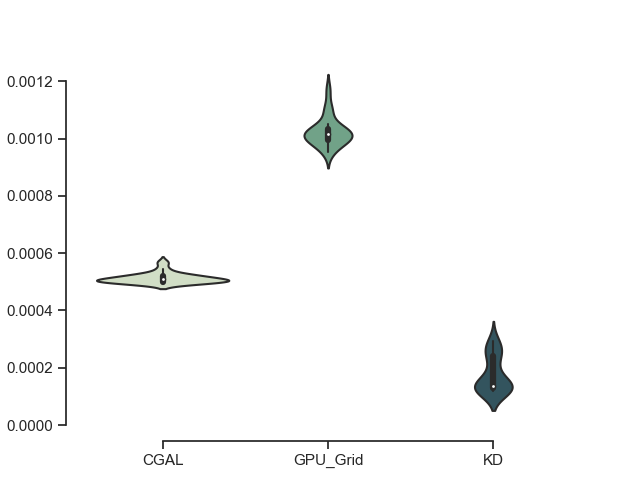
1. Exempel på det de gemensamma delarna för ett låddiagram och ett violindiagram. Det som särskiljer violindiagrammet är att ju bredare fältet är ju större är sannolikheten att ett slumpvist valt värde ur dess datamängd har det värdet.



1. Exempel på linjediagram *plot* producerar, det visar medelvärdet för algoritmernas exekverings tid beroende på storleken på indata. Linjerna går inte att direkt se eftersom endast ett test användes vilket innehöll 1000 objekt.



1. Exempel på stapeldiagrammet *plot* producerar, det Y-axeln visar medel tiden, X-axeln är algoritmen. Antalet objekt som användes i scenariot står på vänster sida och namnet på scenariot och objekten som användes står överst.



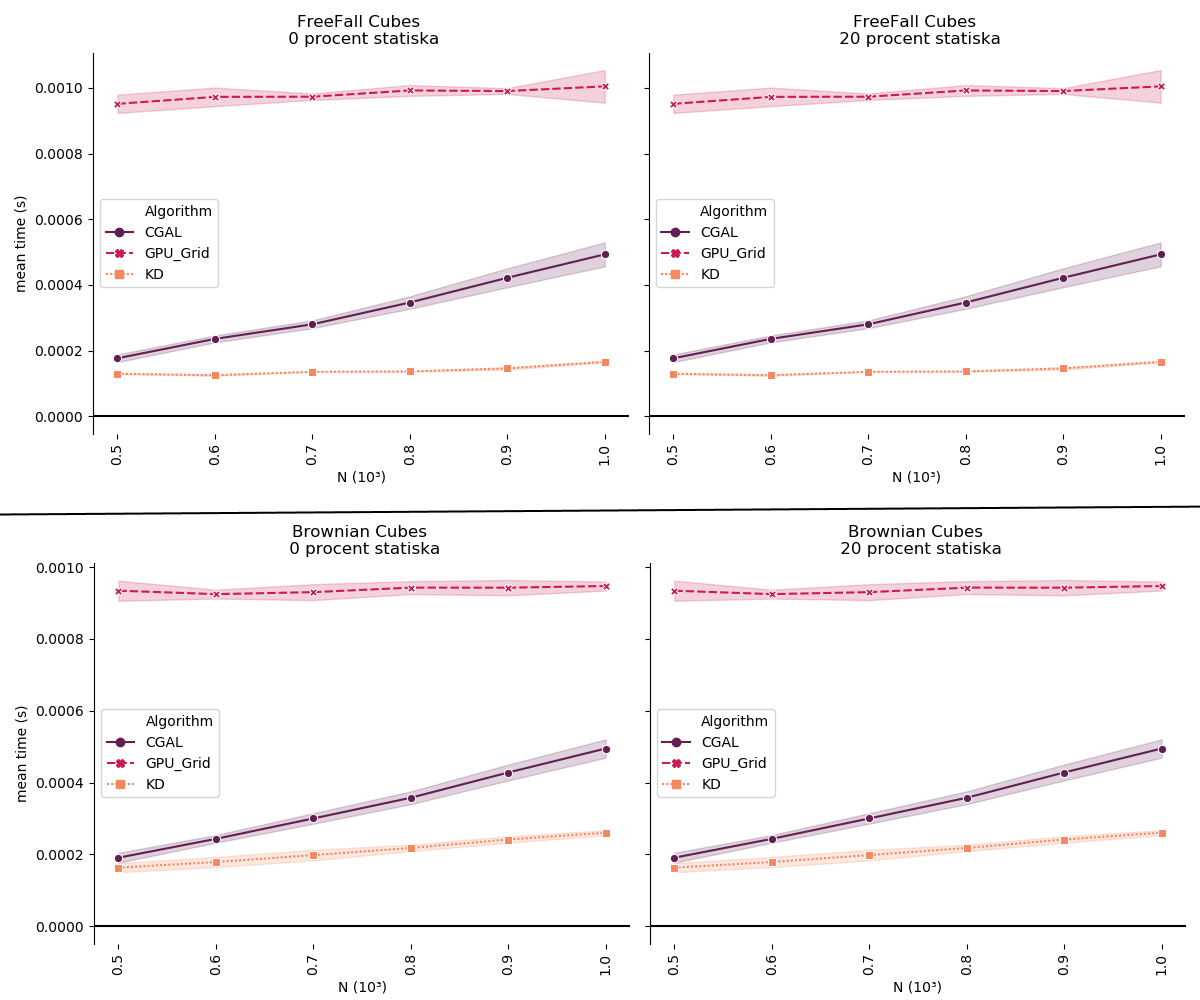
1. Exempel på violndiagram som *plot* producerar, det Y-axeln visar medel tiden, X-axeln är algoritmen.

Resultat utav testkörning utav EJ\_run och EJ\_plot

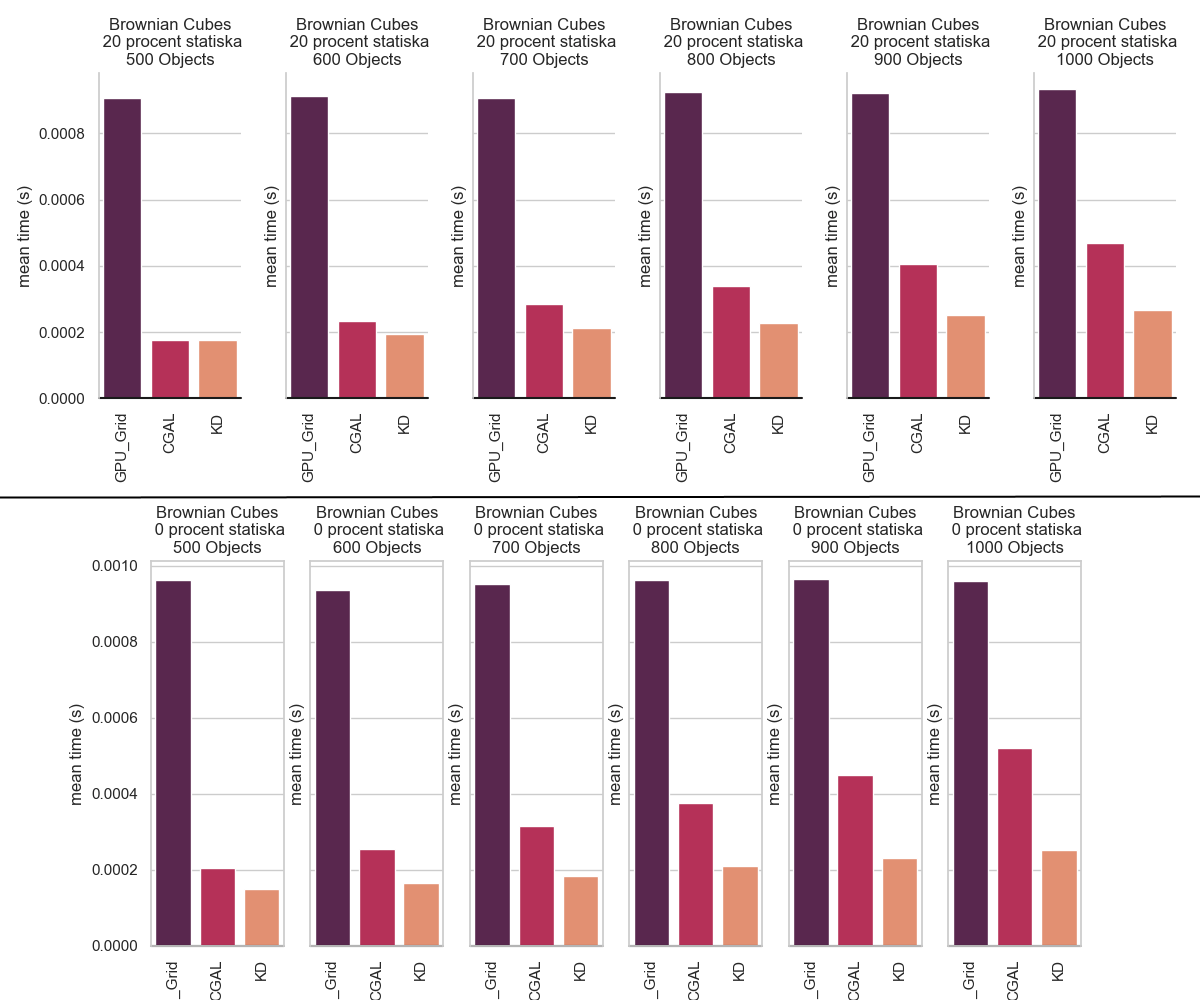
När implementation av EJ\_run och Ej\_plot var klar gjordes en testkörning med följande testserie:

* Scenarion: slumpvandring, fritt fall
* Andel statiska objekt: 0 procent och 20 procnet
* Objekt typ: kuber
* Andel objek: 500, 600, 700, 800, 900, 1000
* Algoritmer: CGAL, KD, grafikprocessor- rutnät

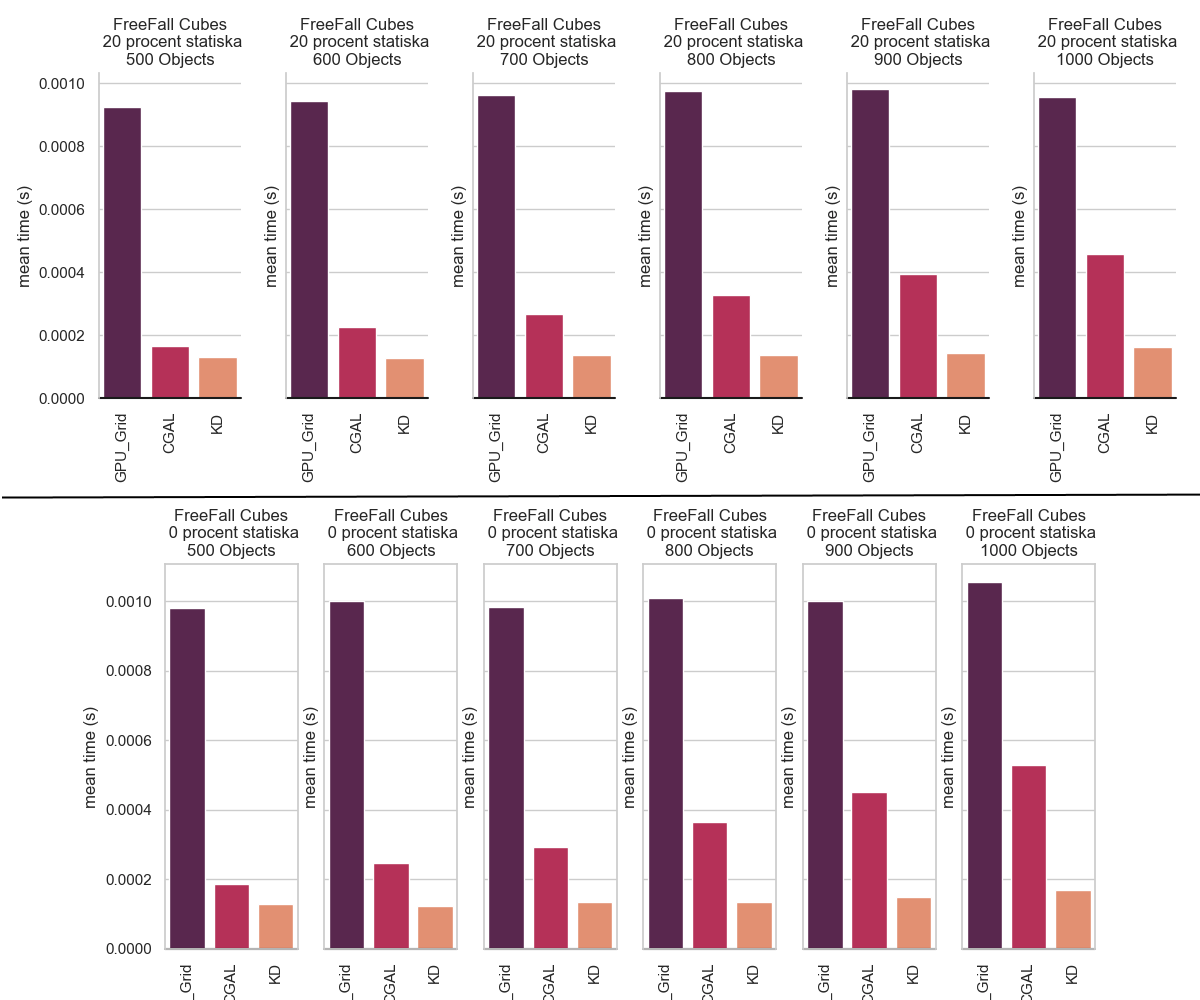
Sedan används EJ\_plot för att producera diagrammen som visas i **figur 15** till och med **figur 21**. **Figur 18** och **figur 19** är som tidigaren nämnt från innan det beslutades att återgå till violindiagrammet som går att se i **figur 20** och **figur 21.**



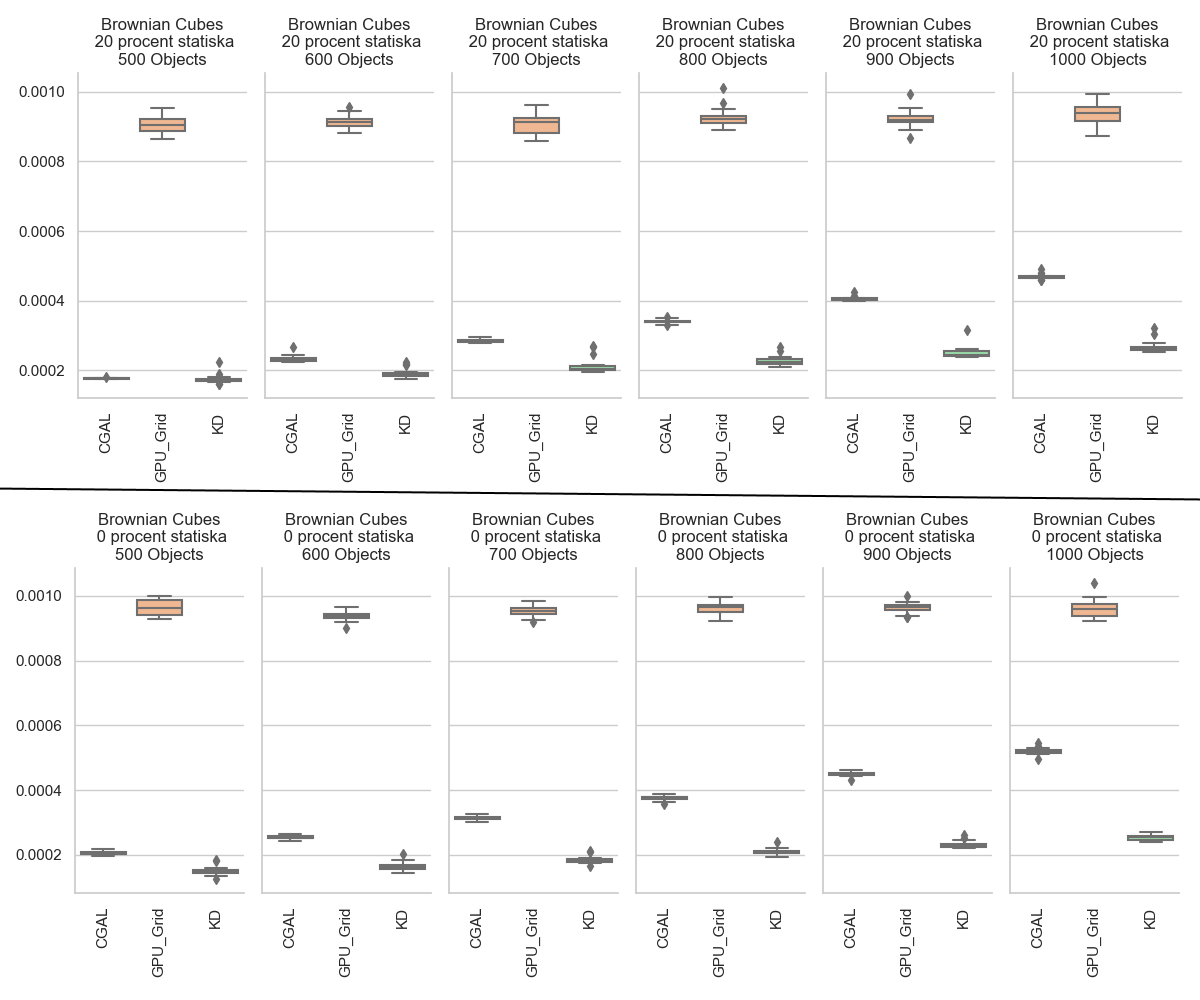
1. Exempel på linjediagram som skapats med EJ\_plot, det visualiserar tre algoritmernas prestanda i förhållande till antalet objetk för två olika scenarion samt två olika förhållanden mellan statiska och dynamiska objekt.

rhållanden mellan dynamiska och feom statiska objekt 

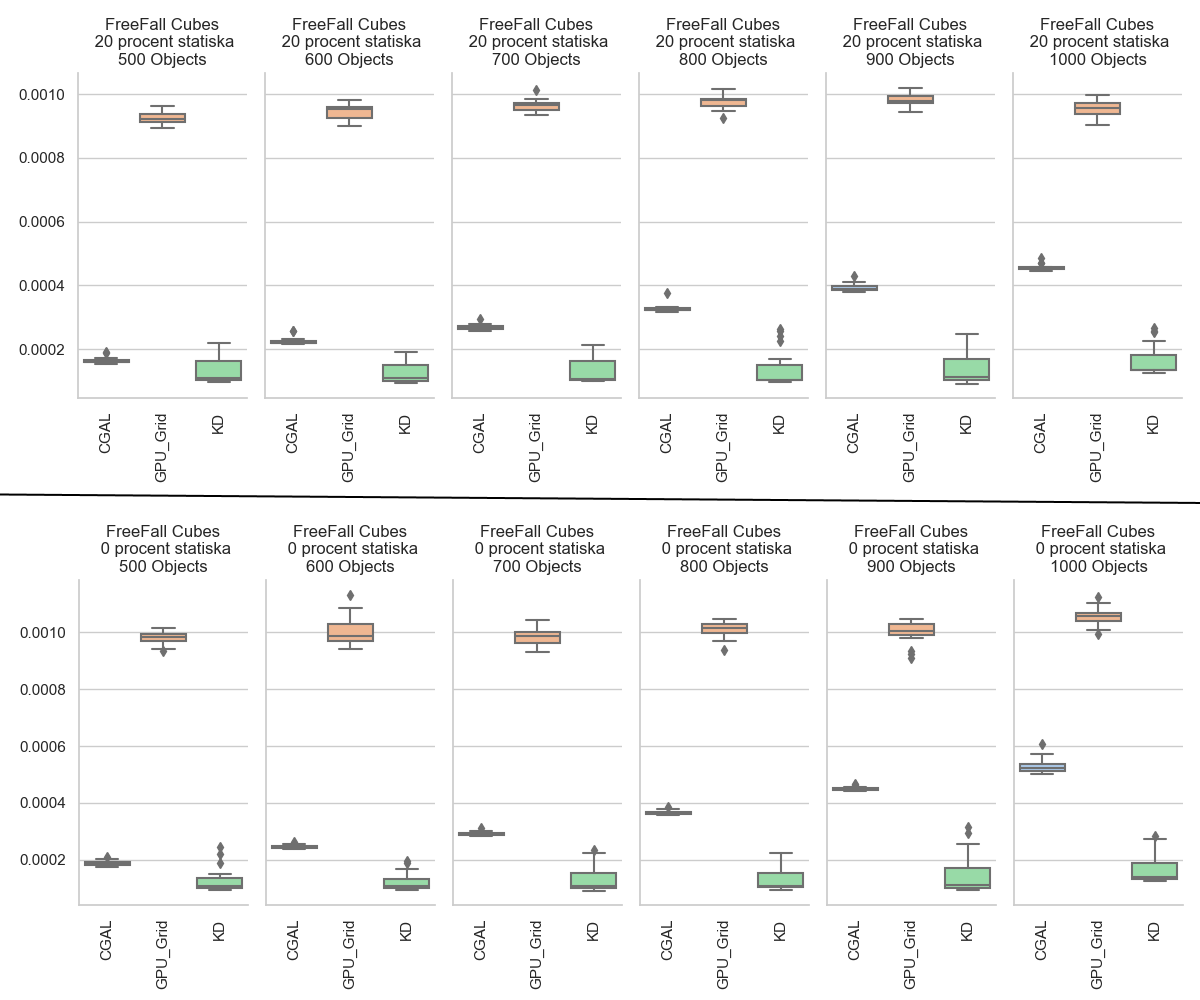
1. Exempel på stapeldiagram från EJ\_plot, resultatet visar medelvärdet för tre algoritmer prestanda för olika mängder objekt i testscenariot slumpvandring med två olika förhållanden mellan statiska och dynamiska objekt.



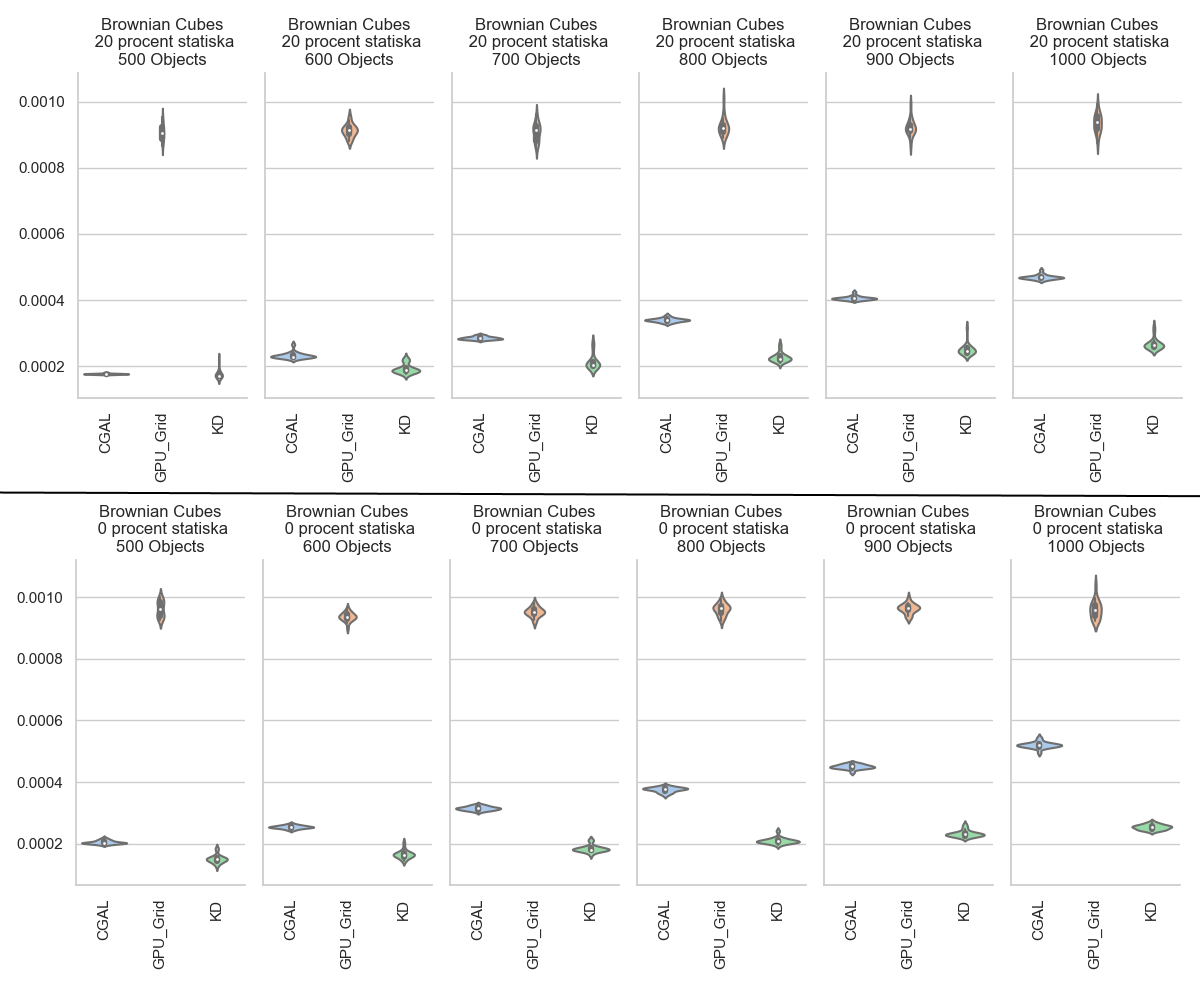
1. Exempel på stapeldiagram från EJ\_plot, resultatet visar medelvärdet för tre algoritmer prestandan för olika mängder objekt i testscenariot fritt fall med två olika förhållanden mellan statiska och dynamiska objekt.



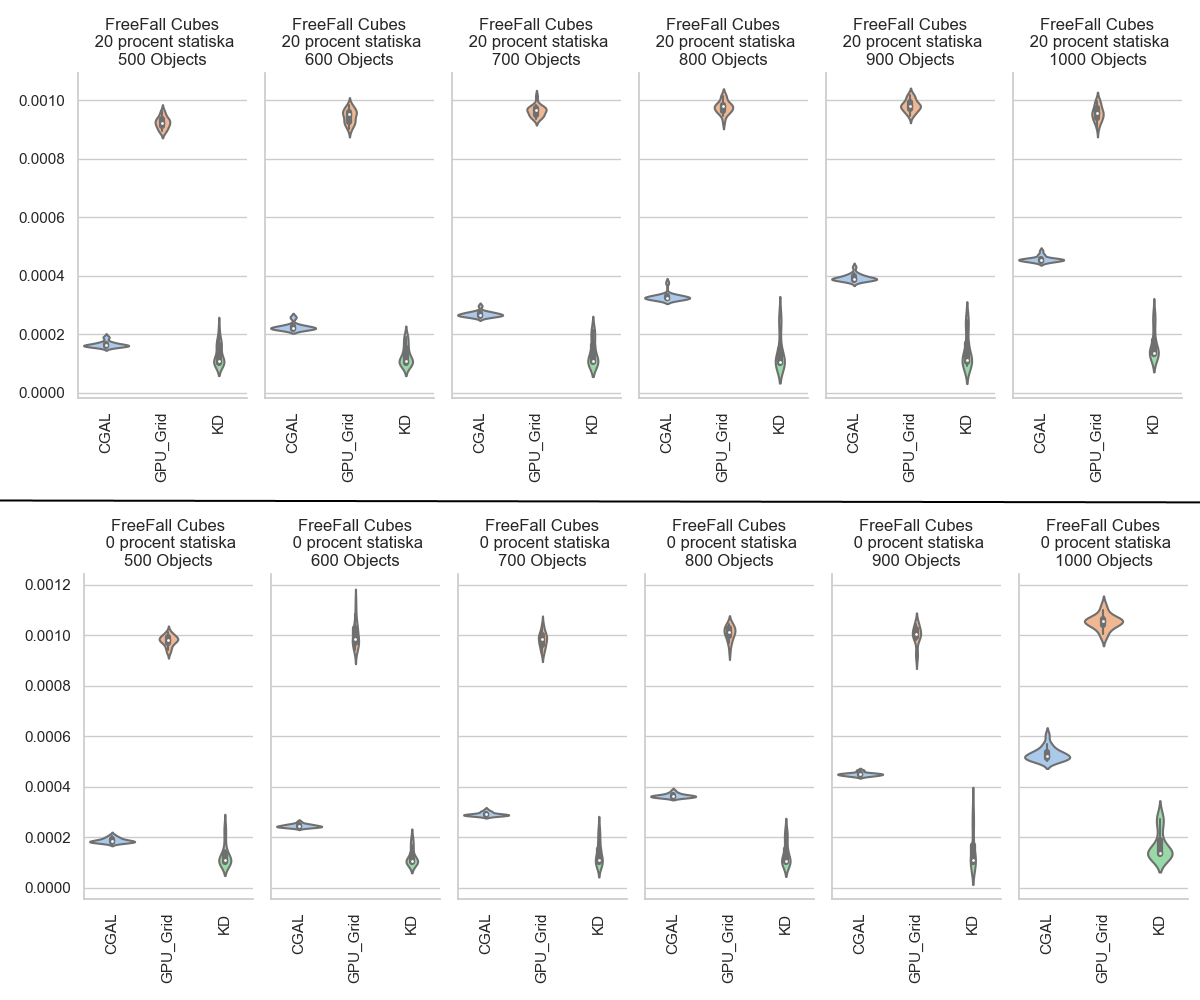
1. Exempel på låddiagram från EJ\_plot, resultatet visar medelvärdets spridning för tre algoritmer prestanda för olika mängder objekt i testscenariot slumpvandring med två olika förhållanden mellan statiska och dynamiska objekt.



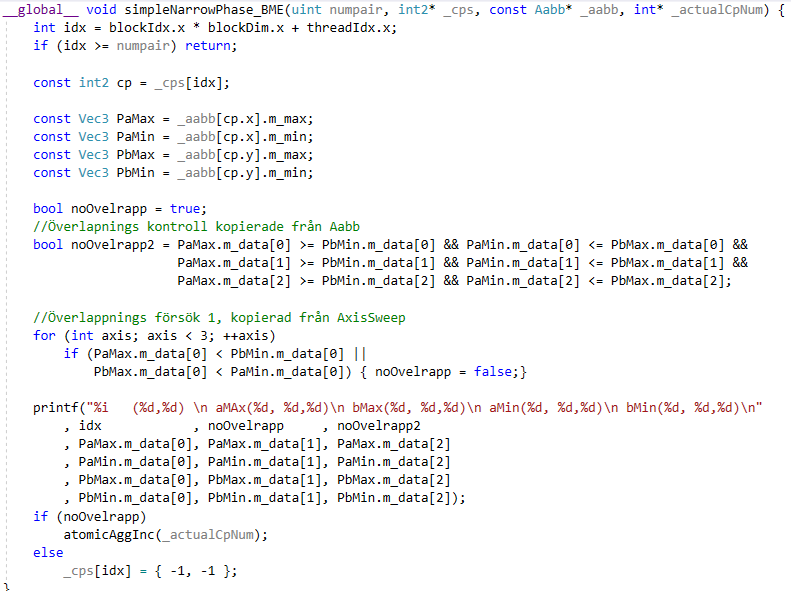
1. Exempel på låddiagram från EJ\_plot, resultatet visar medelvärdets spridning för tre algoritmer prestanda för olika mängder objekt i testscenariot fritt fall med två olika förhållanden mellan statiska och dynamiska objekt.



1. Exempel på violindiagram från EJ\_plot, resultatet visar medelvärdets spridning för tre algoritmer prestanda för olika mängder objekt i testscenariot slumpvandring med två olika förhållanden mellan statiska och dynamiska objekt. Samt visar det den statiska sannolikheten för de olika värdena.



1. Exempel på violindiagram från EJ\_plot, resultatet visar medelvärdets spridning för tre algoritmer prestanda för olika mängder objekt i testscenariot frit fall med två olika förhållanden mellan statiska och dynamiska objekt. Samt visar det den statiska sannolikheten för de olika värdena.
2. Överlappningssökningen



1. Nya överlappningstestningen när det undersöktes om alla par redan överlappade.

För att effektiversara beslutades det att implementera en simpel axelsökningsalgoritmer, som beräknas parallellt på grafikprocessorn. Dock noterades under testningarna att längden på den initiala överlapningsparlistan verkade vara alltid lika lång som den resulterande överlappningsparlistan. Genom att skriva ut resultat från varje enskilt överlappnings test kunde det ses att alla paren alltid överlappade varandra, vilket skulle betyda att igen extra överlappnings sökning skulle egentligt behövas. För att försäkra sig om att det stämde så implementerades även en simpel lådöverlapningstes, se **figur 1**. De två olika överlappningstestet prövades och båda visade att alla paren alltid överlappade varandra. Överlappningstestet för BVH-SR behöver där av endast bestå av att uppdateringen av BVTT-fronten och det enda som behöver vidarebefordras till Broadmark är antalet överlappnings par.