

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

# Bakgrund

## Typer av kollisioner

## Typer av kollisionsdetektion

## Gränsvolymer

1. Flertalet olika gränsvolymer har utvecklats, här visas ett axplock av dem.
2. De tre vanligaste representationerna av AABB:er, (1) min-max värden, (2) min-bredd värden och (3) center-radie värden.

## Bredkollisionsdetektion

1. Exempel på en scen med fem grisar varav två kolliderar med varandra.
2. 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
3. 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.
4. BVH som delar upp objekt tills max två objekt finns i varje gränsvolymsnod.
5. Exempel på bygget av en BVH med toppen-botten, uppdelningen av miljön i
6. Exempel på bygget av en BVHmed botten-toppen, uppdelningen av miljön i större och större gränsvolymer kan följas från bild 1, 2, …, 5.

## small kollisionsdetektion

## Broadmark

1. De tre objekt typerna som stöds i simuleringsgeneratorn. Till vänster sfärer, mitten kuber och till höger olikformade rektanglar.
2. Summering av algoritmer inkluderade i Broadmak (Serpa & Rodrigues 2019b, s.9)

# Problemformulering

## 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. Brodmark 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 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 andelen objekt som testats.

För att få en mer nyanserad bild över hur algoritmerna presterar kommer resultat av testning även användas för att skapa ett låddiagram. Ett låddiagram kommer nämligen tydliggöra 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

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

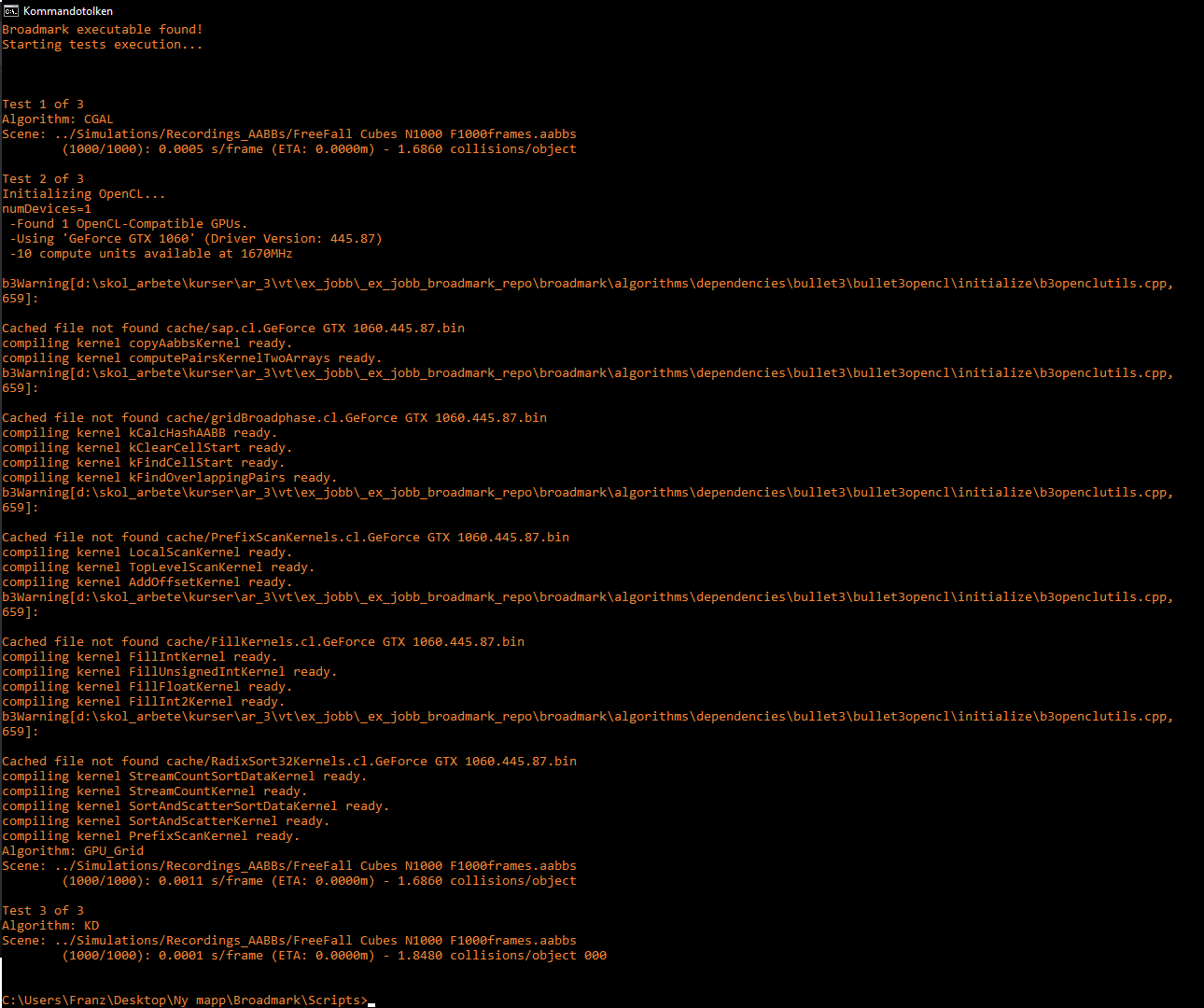
Broad mark är implementerad i utvecklingsmiljön *Visual Studio 2017* och är skriven med programmeringsspråket c++. Det är därmed konsekvent att det är det som används i denna undersökning för att vidareutveckla Broadmark.  
Broadmarks källkod finns att ladda ner från internet (Serpa & Rodrigues, 2019c) 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 Brodmark kan den antas uppfylla verifierbarheten.
* Om Brodmark påvisar en låg standardavvikelse kan Broadmark antas uppfylla första delen av reproducerbarhet kravet.
* Om Brodmark 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:

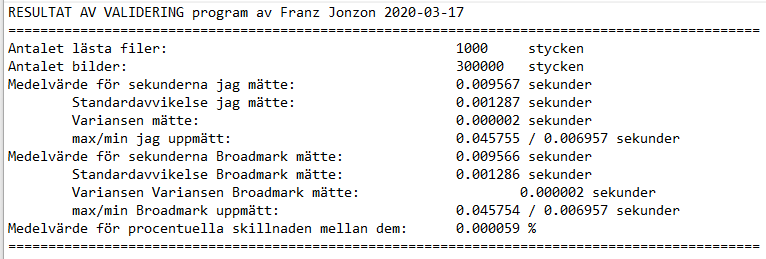
* Testscenario, Fritt fall
* Antal objekt, 8000 stycken
* Antal uppdateringar, 300 stycken
* Algoritm, CGAL
* Antal testkörningar, 1000 stycken

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

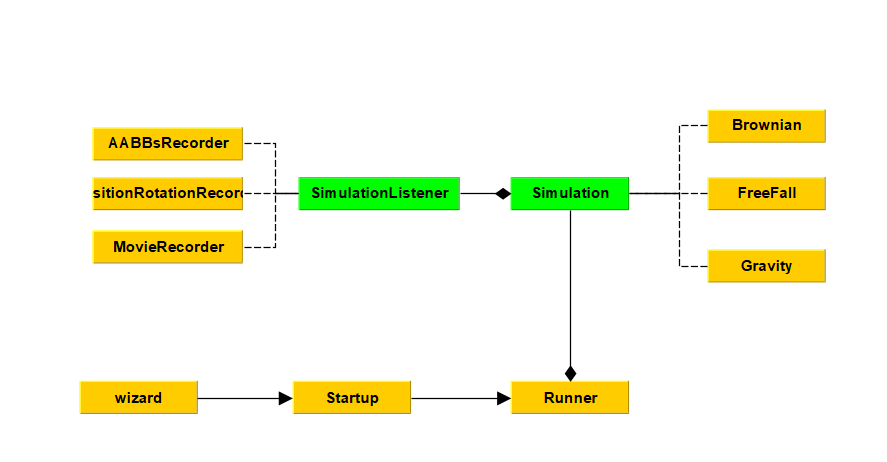
Resultatet av valideringstestet kan ses i **figur 12** och det går att från det dra slutsaten att Brodmark uppfyller både kravet för reproducerbarhet och verifierbarhet. Den egna uppmätta tiden skiljer sig från Broadmarks med endast mikrosekunder. Med en sådan liten skillnad kan de i stort anses vara lika och därmed kan verifieringsbarhet sägas vara uppfylld. Avrundas till två decimaler, som Serpa & Rodrigues(2019b s.12) gjorde, uppmätte både Brodmark 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.

## Modifiering av simulationsgenereringsapplikationen

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 13**. 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.

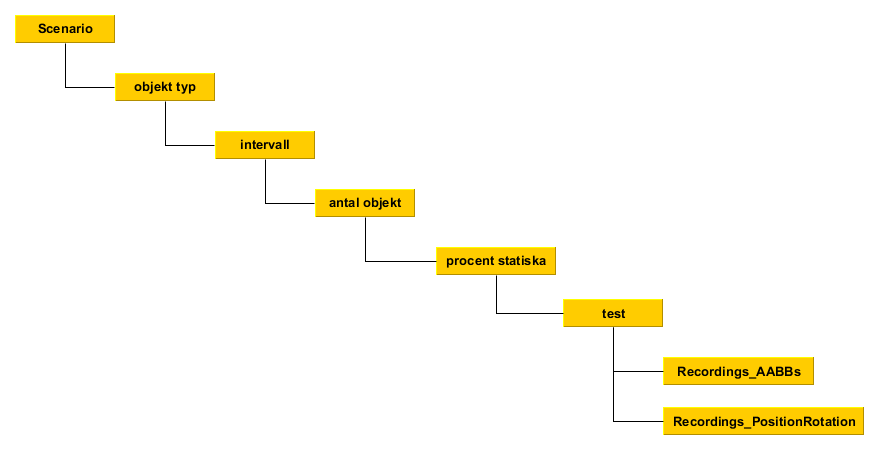


1. Förenklat strukturdiagram som visar de för undersökningen relevanta delarna av simulationsgeneratorn, gröna noderna är *interface* och de noder som är anslutna till de gröna noderna med streckade linjer är klasser som implementerar det *interface*.

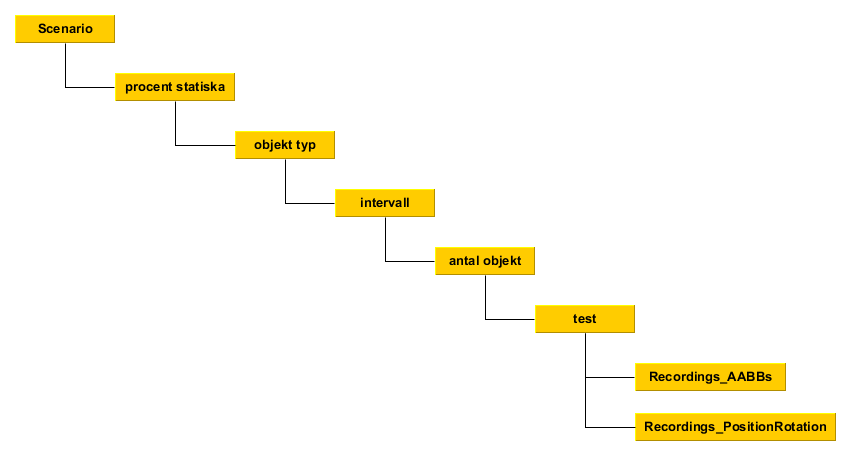
### Automatisering av testerna

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.

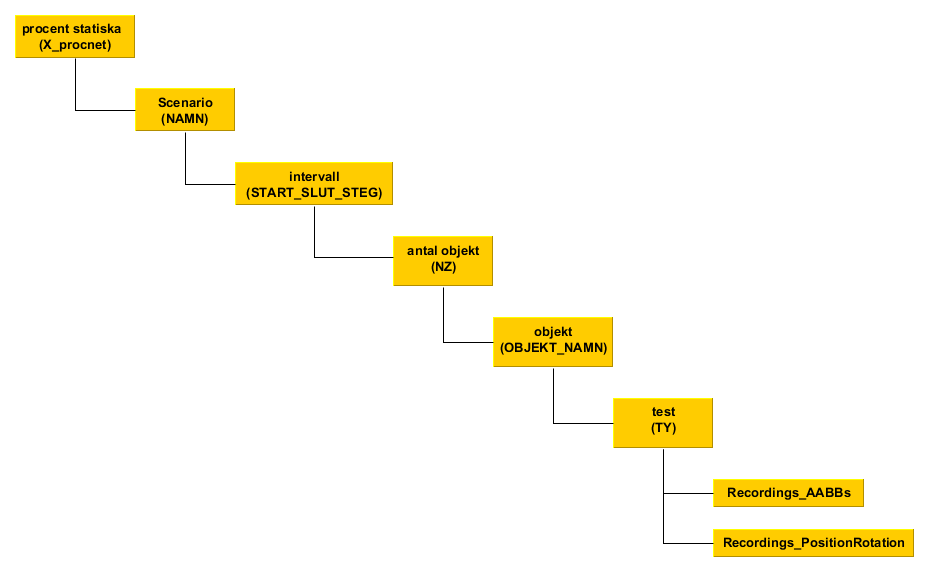
Det första som beslutades var utövningen av filhierarki till följd av att implementeringen av resterande modifikationer kommer att vara beroende på hur den utformas Tanken med filhierarki är att underlätta användning av informatio som genereras, vilket resulterade i att den reviderades ett par gånger. 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 14** till **figur 15**.



1. Första iterationen av filhierarkin



1. Första iterationen av filhierarkin

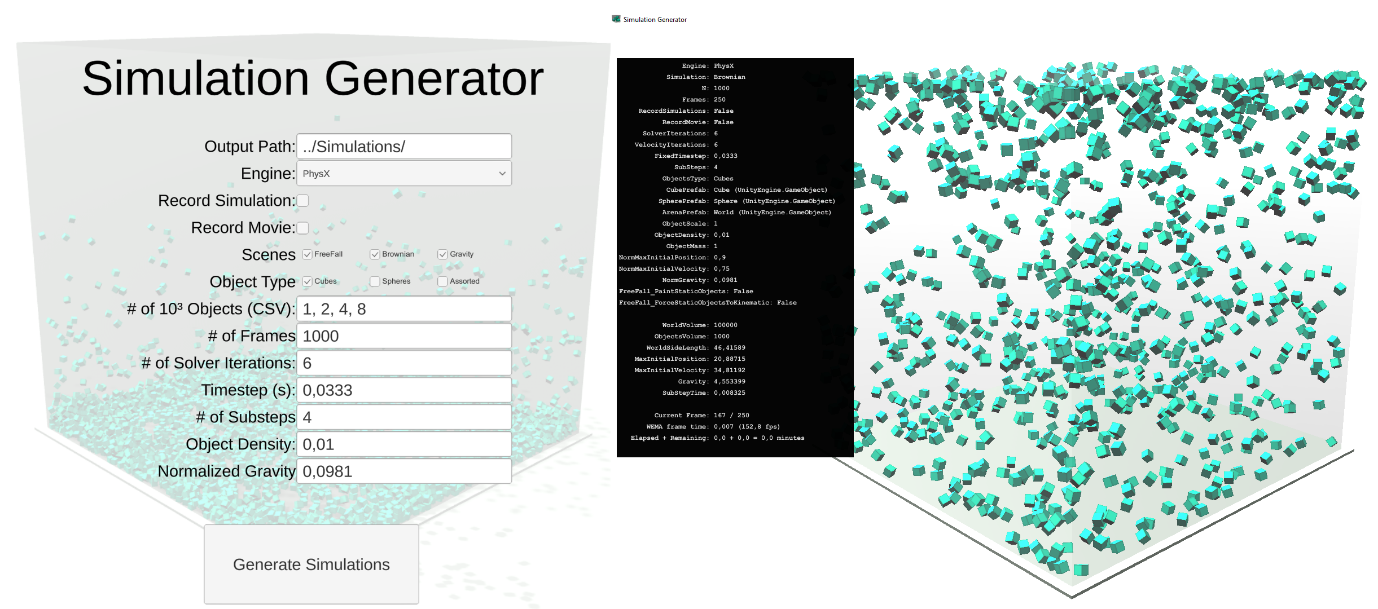


1. Slutliga versionen av filhierarkin som den modifierade simulationsgeneratorn sparar ner scenariona i. Namn standarden för varje del av hierarkin står i parenteserna, med undantag för de Recordings\_ mapparna.   
   Namn variablernas betydelse är följande: X procentuell andel statiska objekt, NAMN 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.

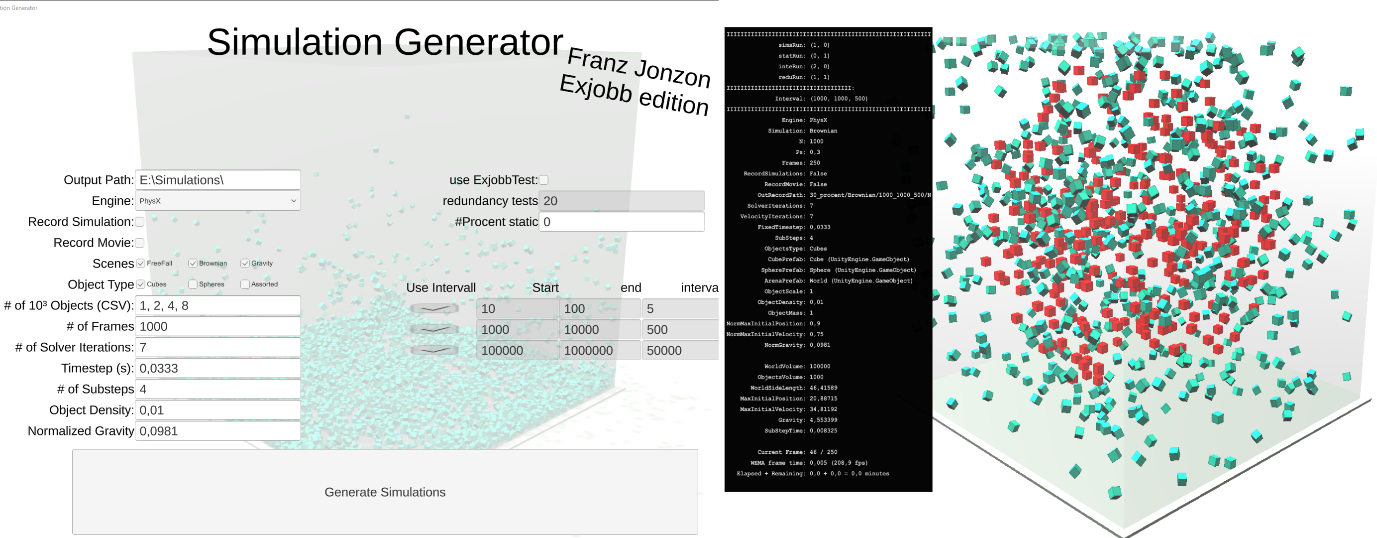
Efter filhierarkin adderas sedan de parametrar som behövdes för att automatisera testningen. Parametrarna som behövde adderas är,

* 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 17** visar på hur simulationsgeneratorns utseende ser ut innan dess funktionalitet expanderades och **figur 18** 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 Brodmark 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.

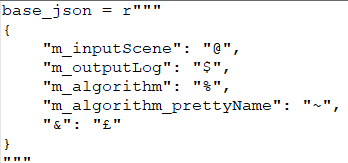
Utöver detta skulle expanderade version av simulationsgeneratorn även behöva spara hur stor andel av objekten som är statiska, eftersom det kommer vara nödvändigt vid utvärderingen hur väl olika algoritmer hanterade det specifika scenariot. Första idén var att addera detta värde till aabbs filformatet, 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 16**, 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.

## Implementation av hjälpskript

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 19**, 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 Brodmark producerar, för exempel se **figur 20**, 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 24** till **figur 26**.



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



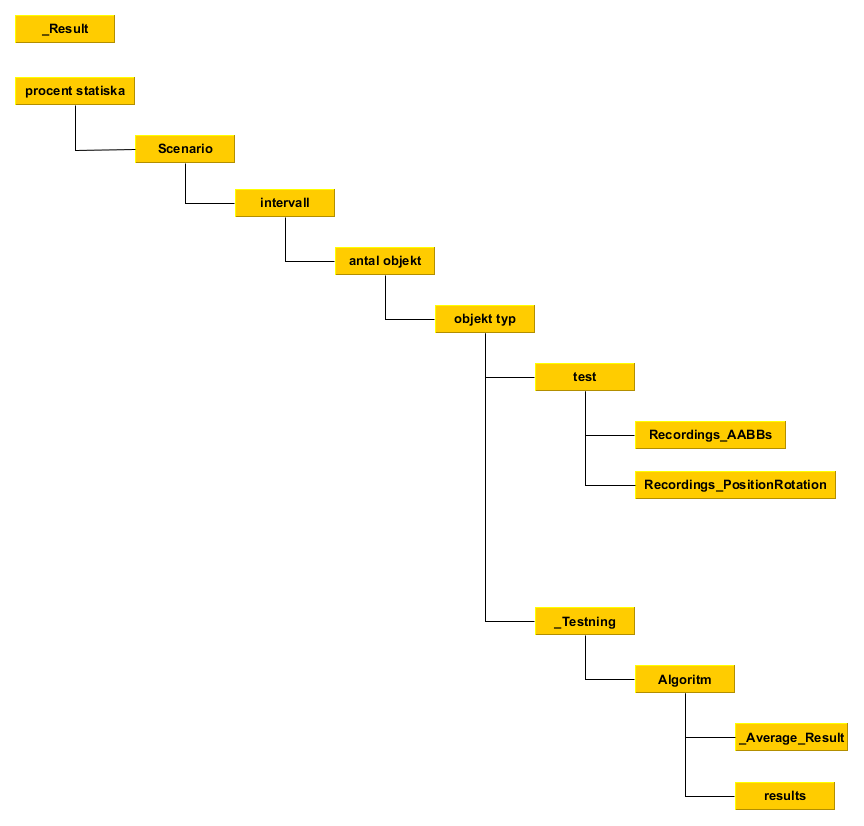
1. Exempel på .json filen innehållandes resultat från en testkörning av CGAL i Broadmarks.

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

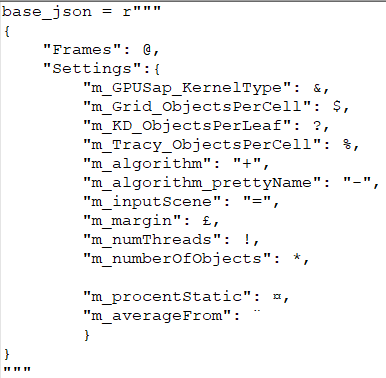


1. Filhierarkin efter att testningen har genomförts.

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 21**.

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 20**, 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 22**.



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.

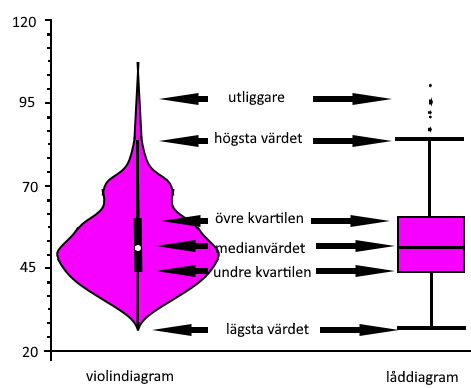
### Resultat visualiserar skriptet

*Seaborn* (2013)

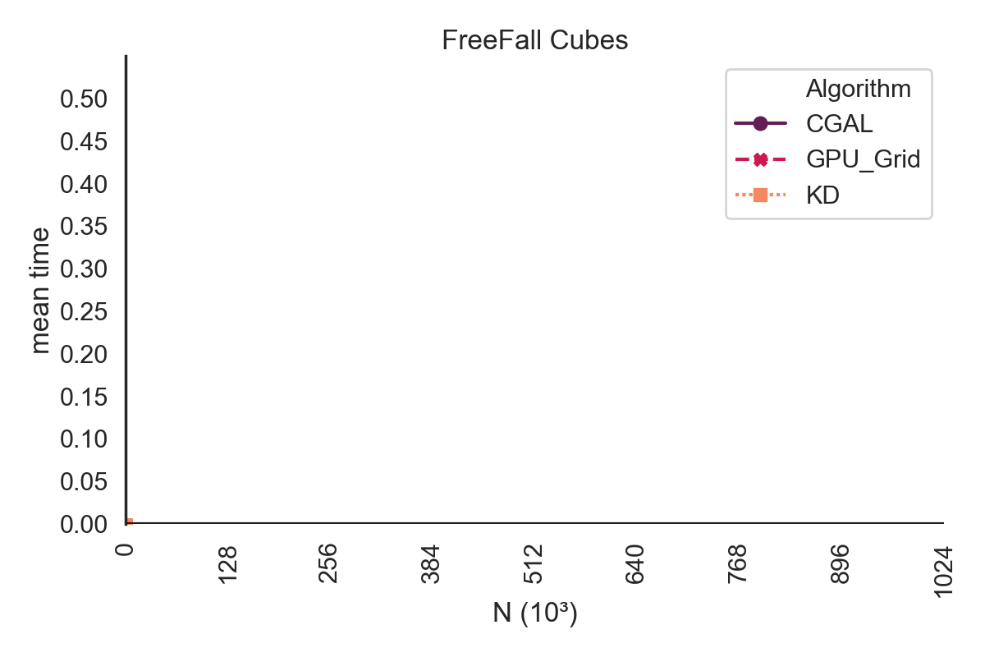
Ej\_plot krävde något mer efterforskning *Seaborn* (2013) och *Pandas* (2008) dock tack vare att de var mycket väldokumenterad var detta inte något som tog mer tid än nödvändigt. Valet gjordes att så långt som möjligt presentera relevanta diagram, exempelvis **figur 28** gjordes med EJ\_plot och vissas 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 30** och **figur 31** 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, se **figur 23** Hintzer, J., L. & Nelson, R., D.(1998, s.181). för att se skillnaden mellan jämför **figur 30** och **figur 31** med **figur 32** och **figur 33**

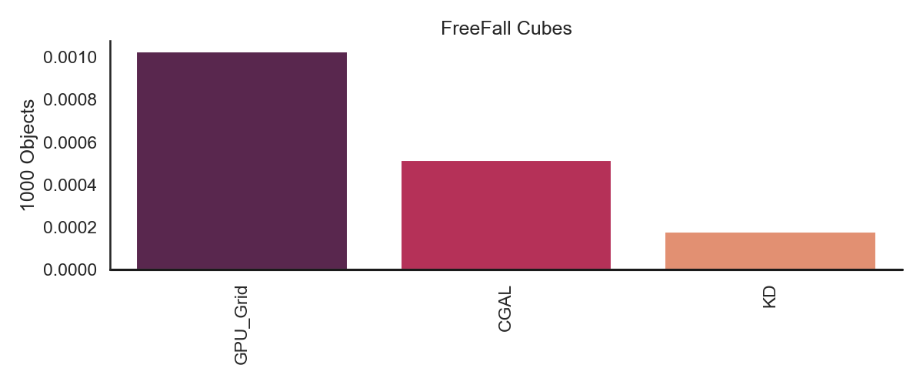
När implementation var klar gjordes en testkörning



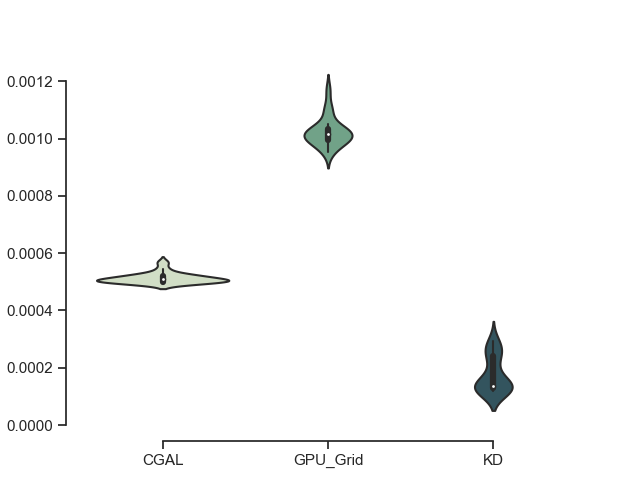
1. Exempel på det de gemensamma delarna för ett låddiagram och ett violindiagram. Det som särskiljer violindiagrammet är att ju bredare fältet är ju större är sannolikheten att ett slumpvist valt värde ur dess datamängd har det värdet.



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.



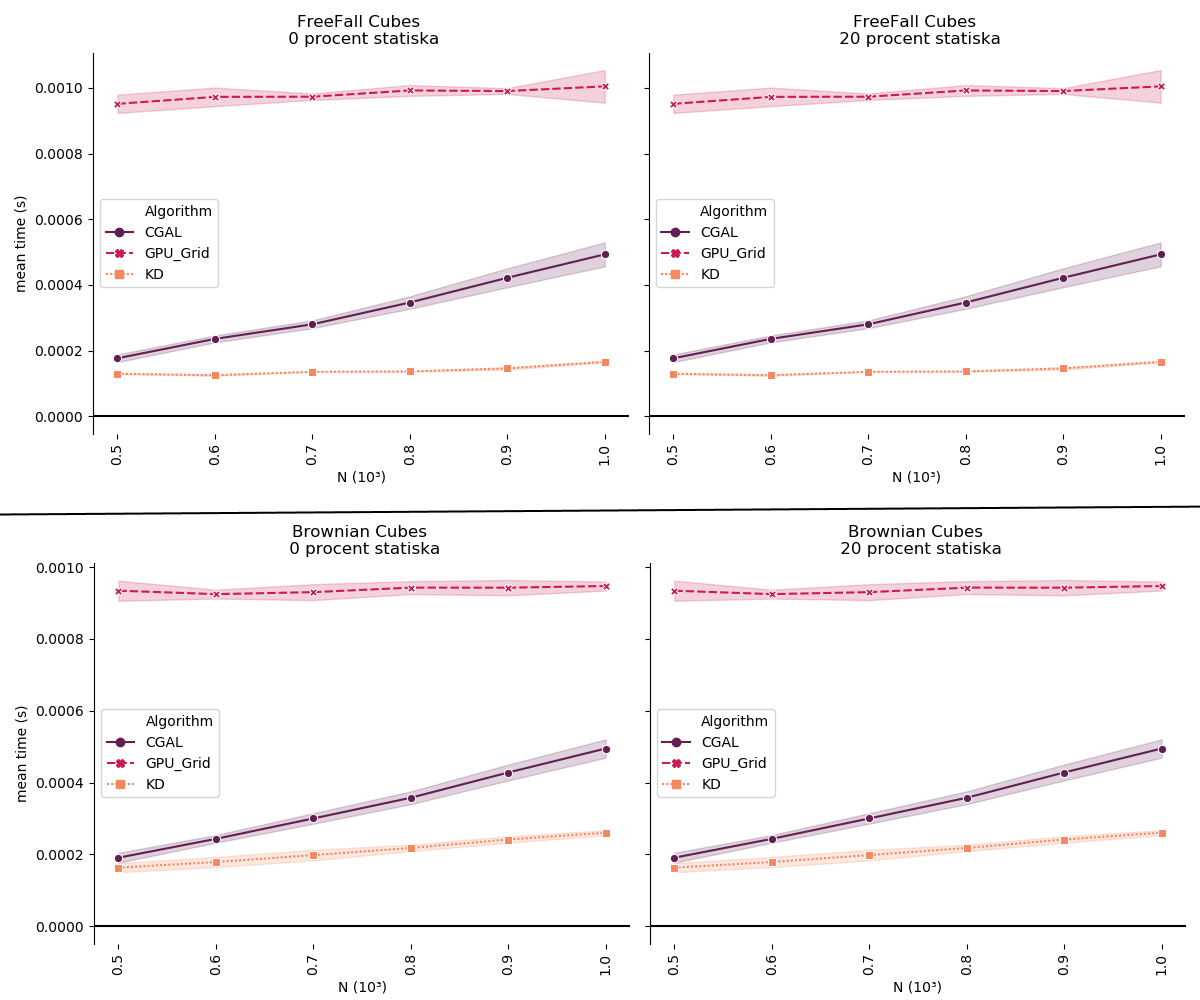
1. Exempel på violndiagram som *plot* producerar, det Y-axeln visar medel tiden, X-axeln är algoritmen.

### Resultat utav testkörning utav EJ\_run och EJ\_plot

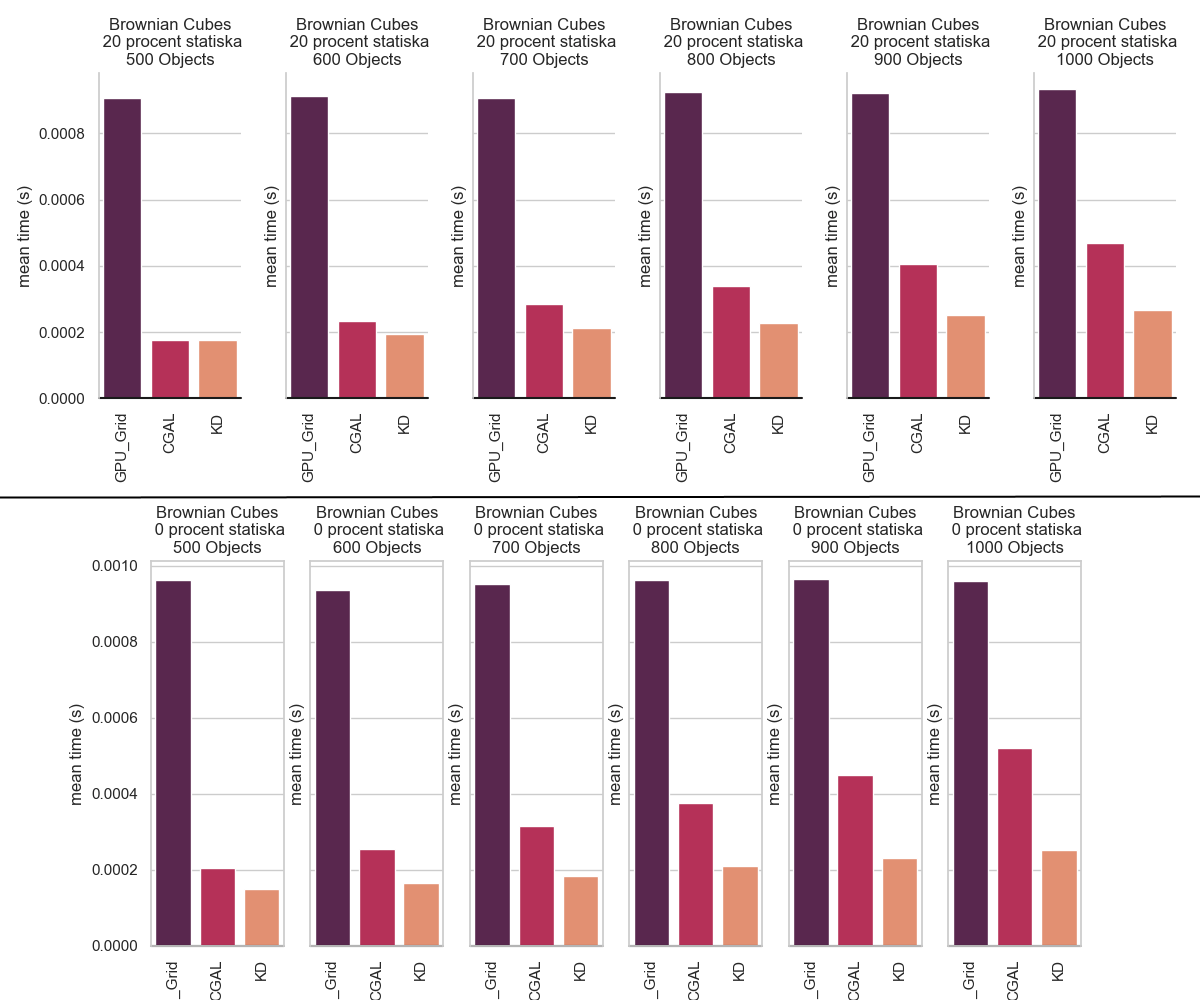
När implementation av EJ\_run och Ej\_plot var klar gjordes en testkörning med följande testserie:

* 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

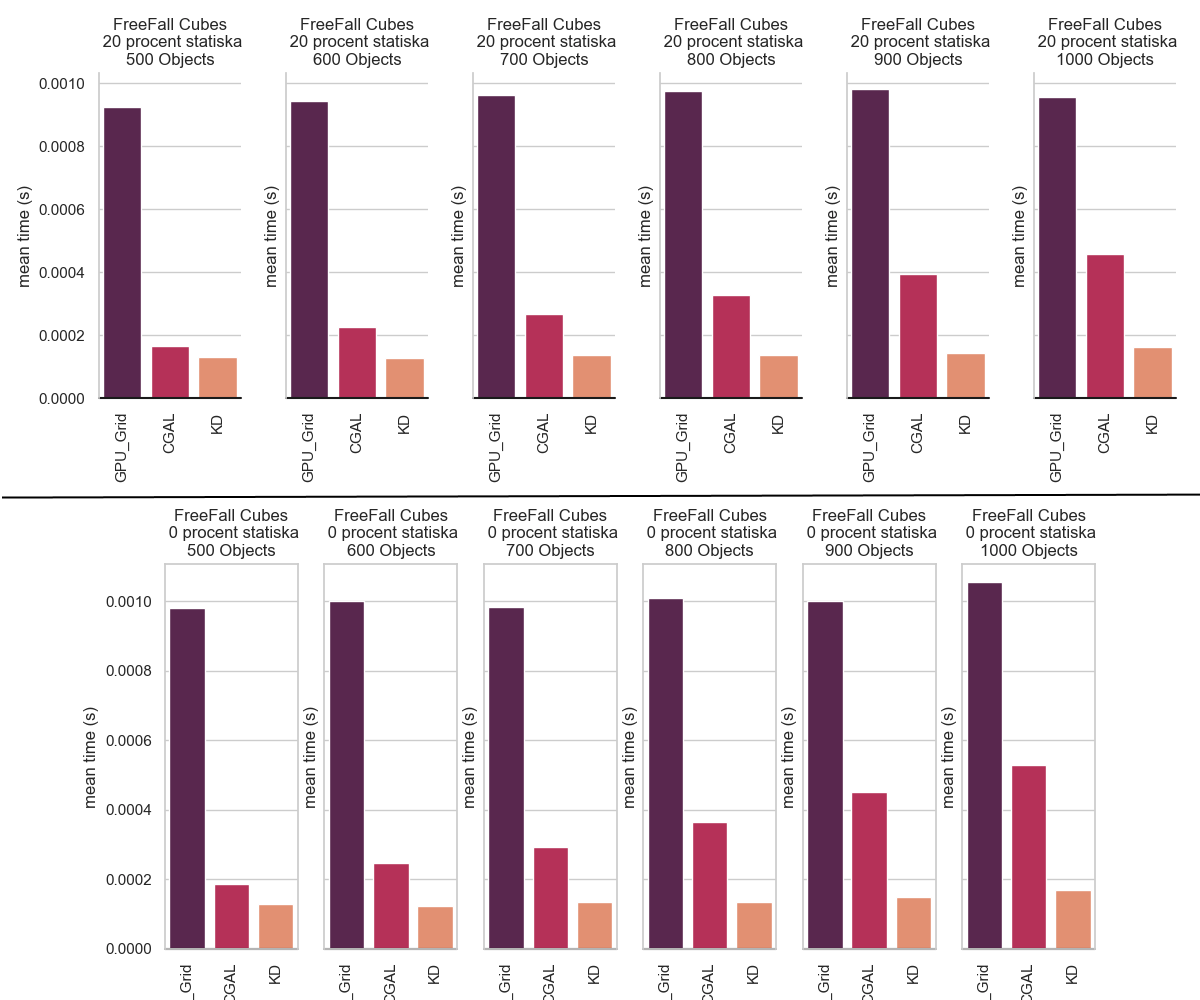
Sedan användes EJ\_plot för att producera diagramen som visas i **figur 27** till och med **figur 34**. **Figur 31** och **figur 32** är som tidigaren nämnt från innan det beslutades att återgå till violindiagrammet som går att se i **figur 32** och **figur 33**



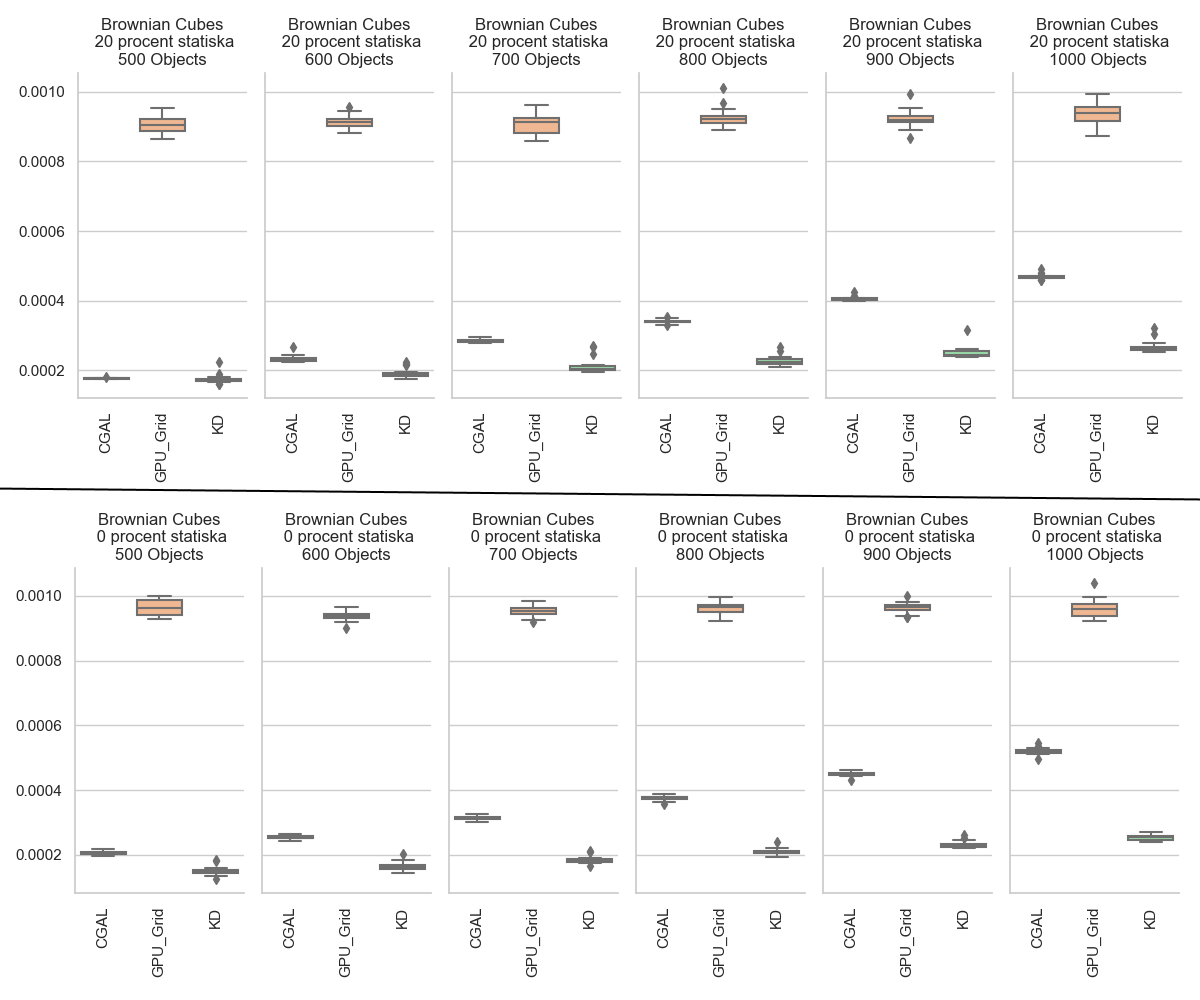
1. Exempel på linjediagram som skapats med EJ\_plot, det visualiserar tre algoritmernas prestanda i förhållande till antalet objetk för två olika scenarion samt två olika förhållanden mellan statiska och dynamiska objekt.

rhållanden mellan dynamiska och feom statiska objekt 

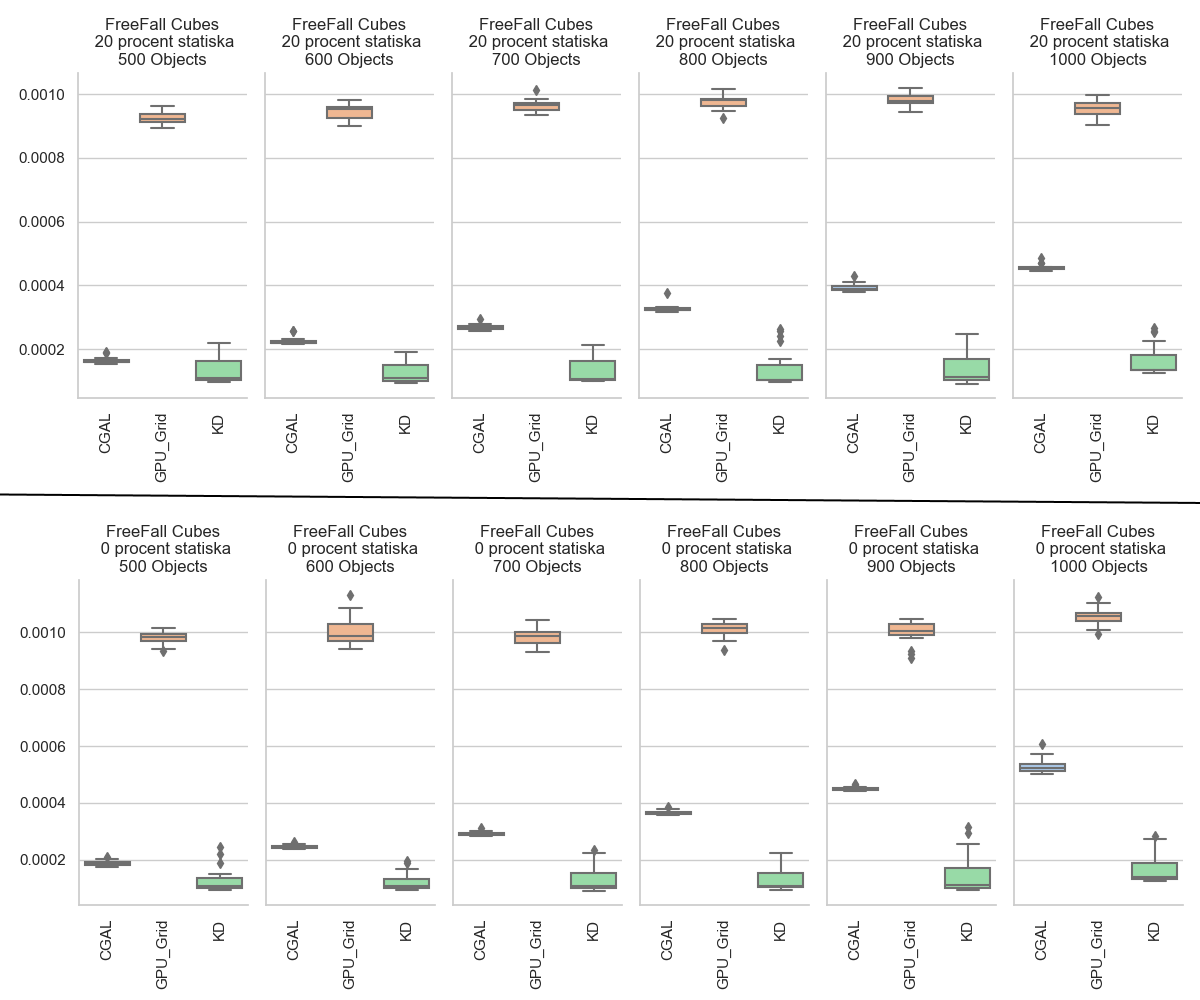
1. Exempel på stapeldiagram från EJ\_plot, resultatet visar medelvärdet för tre algoritmer prestanda för olika mängder objekt i testscenariot slumpvandring med två olika förhållanden mellan statiska och dynamiska objekt.



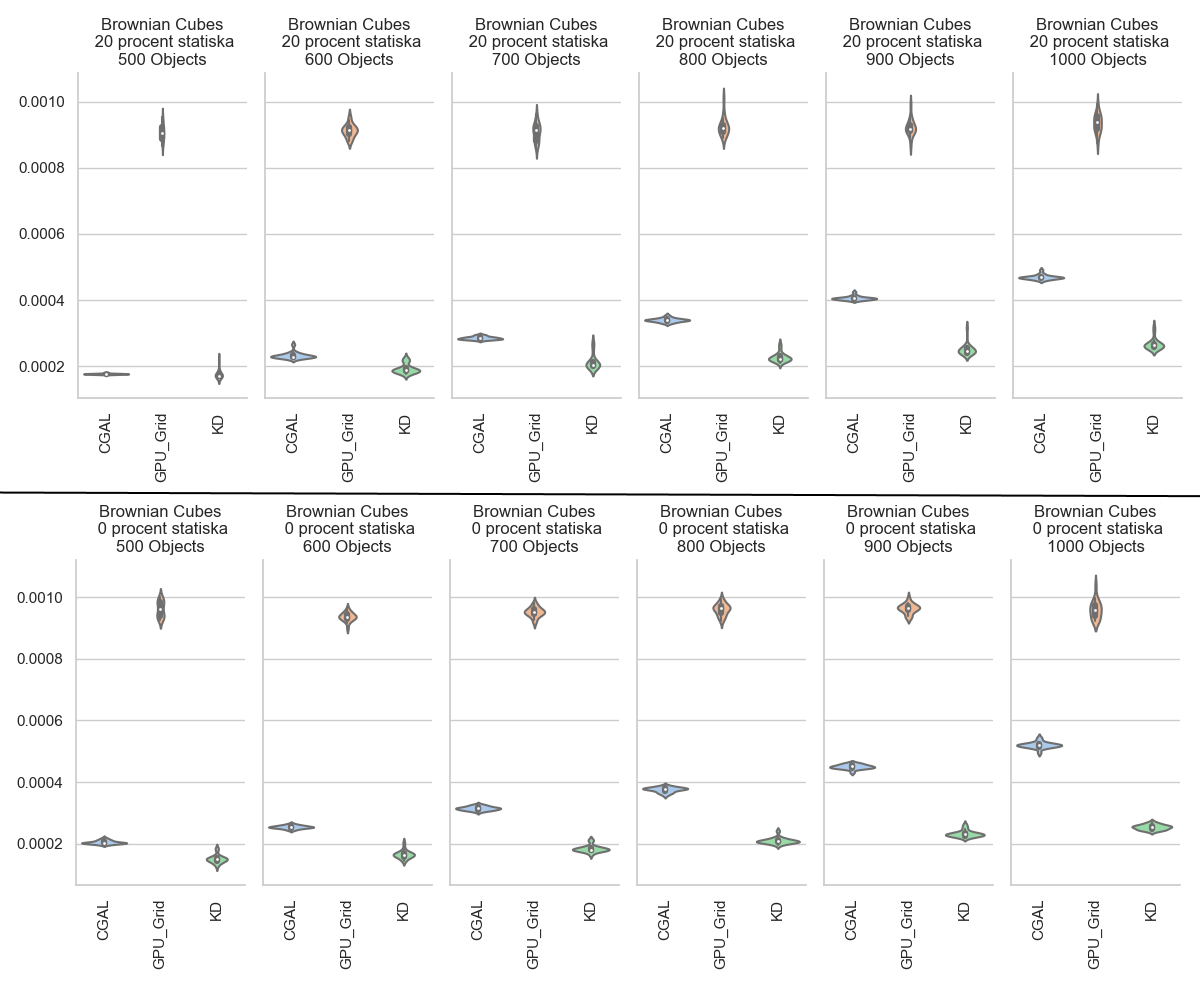
1. Exempel på stapeldiagram från EJ\_plot, resultatet visar medelvärdet för tre algoritmer prestandan för olika mängder objekt i testscenariot fritt fall med två olika förhållanden mellan statiska och dynamiska objekt.



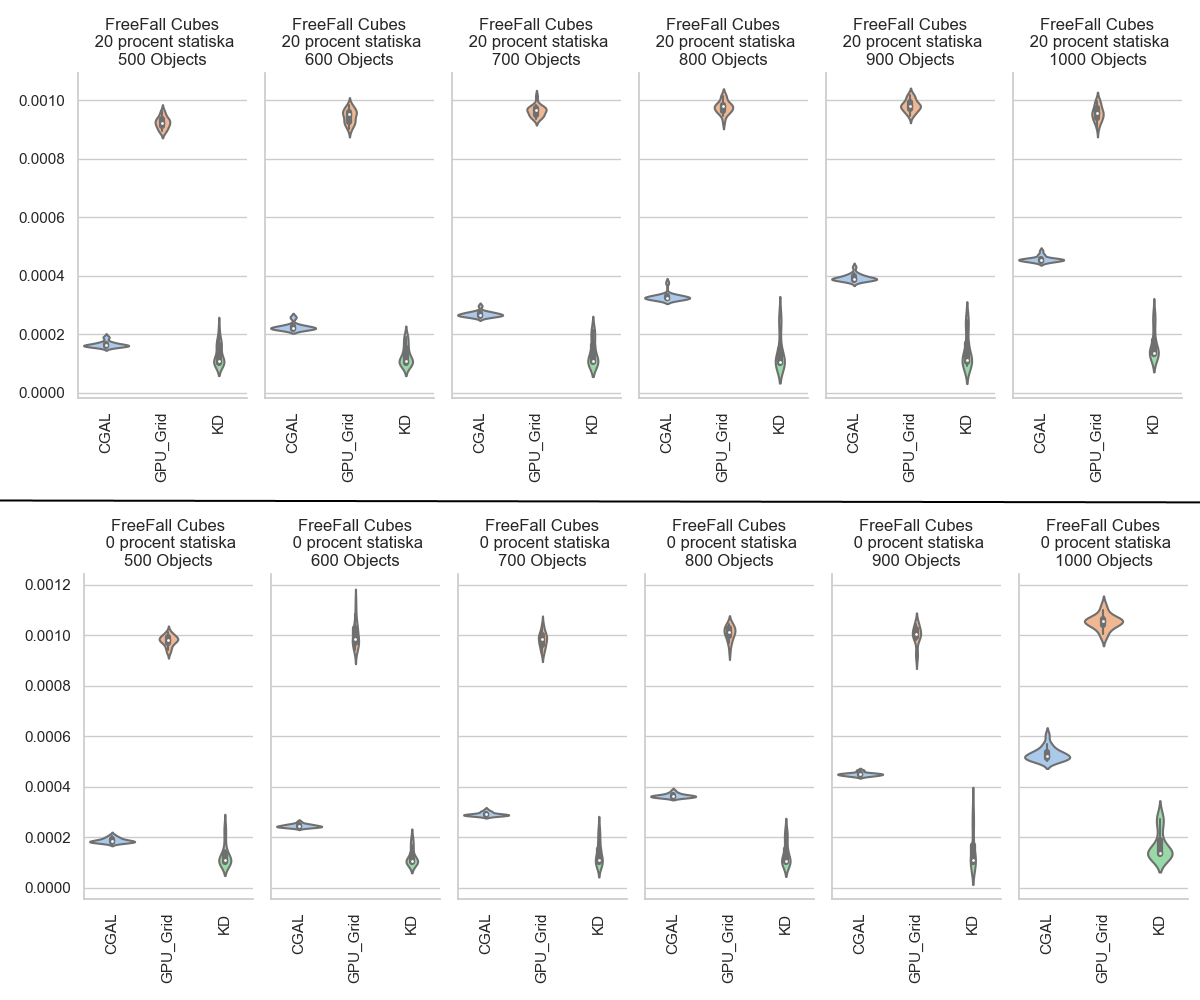
1. Exempel på låddiagram från EJ\_plot, resultatet visar medelvärdets spridning för tre algoritmer prestanda för olika mängder objekt i testscenariot slumpvandring med två olika förhållanden mellan statiska och dynamiska objekt.



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.

Referenser

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