### Uppsättning

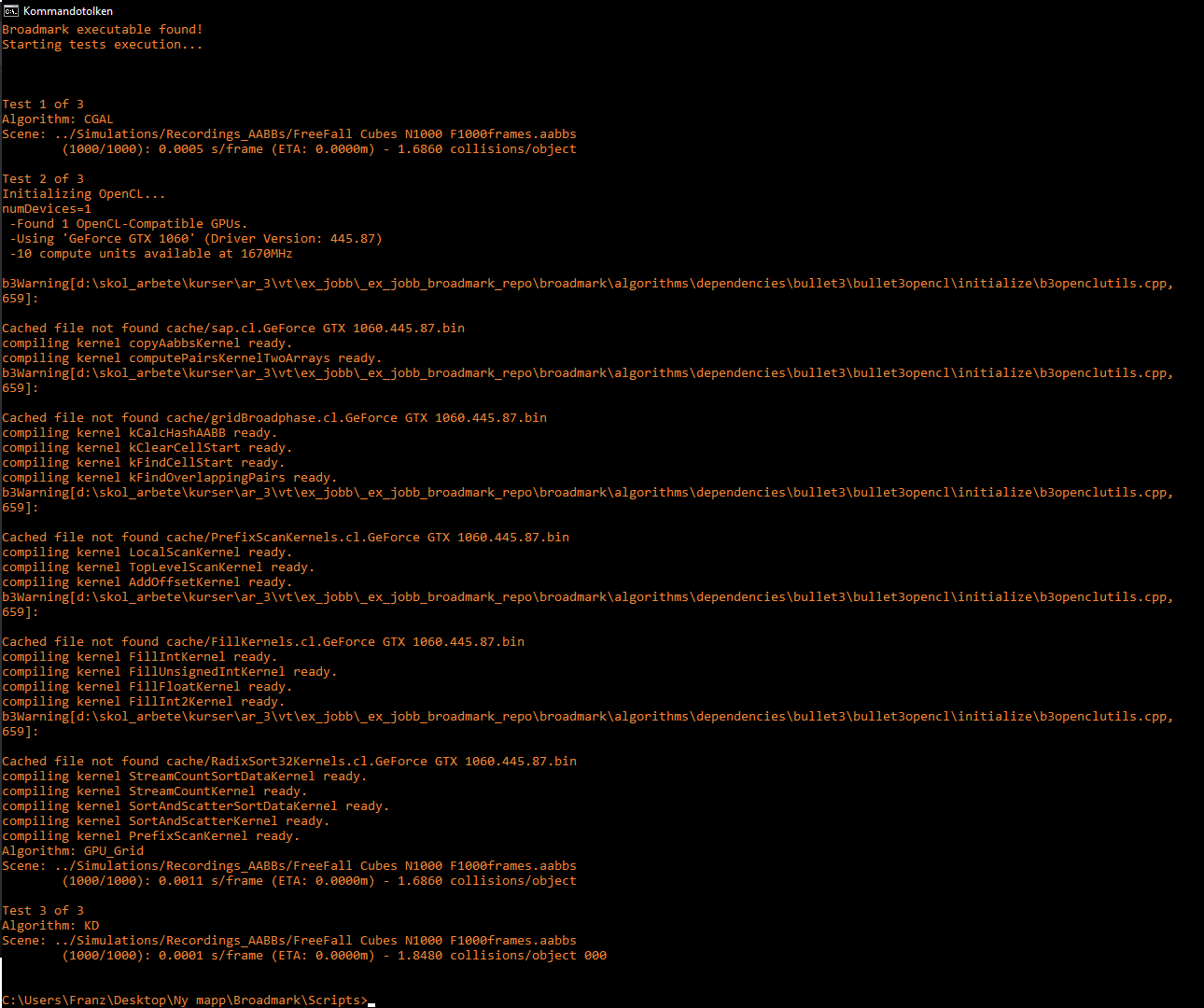
Broad mark är implementerad i utvecklingsmiljön *Visual Studio 2017* och är skriven med programmeringsspråket c++. Konsekvent är det där med även det som används i denna undersökning för att vidareutvecklingen av Broadmark.  
Brodmarks källkod finns att ladda ner från internet (Serpa & Rodrigues, 2019c) 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 inkluderade algoritmerna. De förklarar det är alla grafikprocessorbaserade algoritmerna samt CGAL algoritmen som kräver extra uppsättning, i form av att externa programvarebibliotek laddas och länkas till dem. Plan är att testgrunden vilka BVH-SR, PSM-SAP och KD-SAP ska bestå de tolv övriga algoritmerna som inkluderades i Broadmark så det beslutades att lägga ner arbetet som behövdes för uppsättning av dem. Serpa & Rodrigues (2019c) förklarar i uppsättnings instruktioner att för den grafikprocessorbaserade algoritmer är det nödvändigt att ladda ner *OpenCL SDK* för grafikprocessor som ska använd. För denna undersökning innebar det att ladda ner *CUDA Toolkit 10.2*, (2007). Serpa & Rodrigues (2019c) utvecklar vidare att för CGAL algoritmen krävdes de 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 Brodmark. Till följd av oerfarenhet av att jobba med externa programvarebibliotek i *Visual Studio 2017* var detta något som var något svårt att komma fram till vad exakt problemet var. Det visade sig i slutändan att det fanns två olika varianter av *CGAL* (1995), **figur 10**, och det beskrevs aldrig att båda behövdes för att det skulle fungera.

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



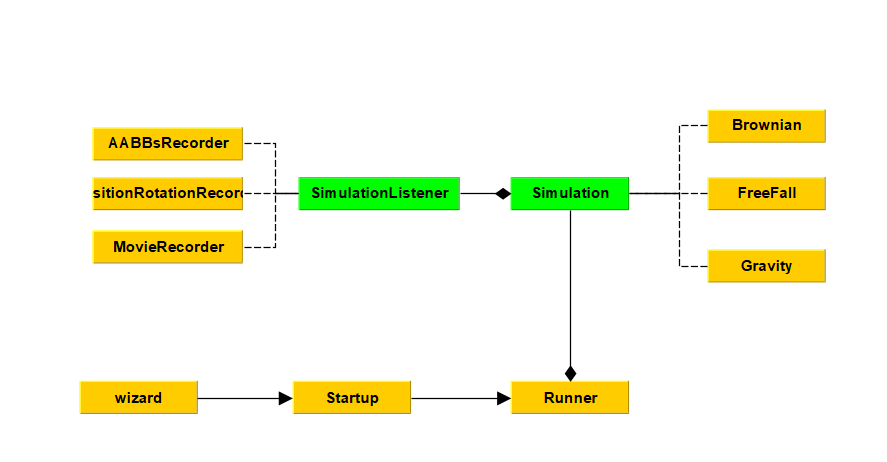
1. Utloggningen från testexekveringen av Broadmark, algoritmerna som testkördes var KD-SAP, CGAL och grafikprocessor-rutnät

## Modifiering utav simulationsgenereringsapplikationen

Simulationsgeneratorn är implementerad i *Unity 2019.2* (2019) vilket är samma program som användes för vidareutvecklingen utav den. Koden assosierad 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 iden var att skapa kopior utav de tre scenerna som Serpa & Rodrigues (2019b) inkluderade i Broadmark och modifiera dem 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 Serpa & Rodrigues (2019b) konstruerade simulationsgeneratorn så att testscenariona implementerar ett standardiserat *interface* vilket används utav en körnings klass, vilken skapar scenariona utifrån de parametrarna som valts, se **figur 13**. Genom *Interface* har körningsklassen tillgång till alla objekt i scenariona, så ett snabbare och enklare sätt att implementera de statiska objekten är att i körklassen direkt efter att scenariot skapats tar och sätter den önskade andelen objekt till att vara statiska. De statiska objekts färg ändras även till röd för att underlätta urskillningen mellan dem och de dynamiska objekten.

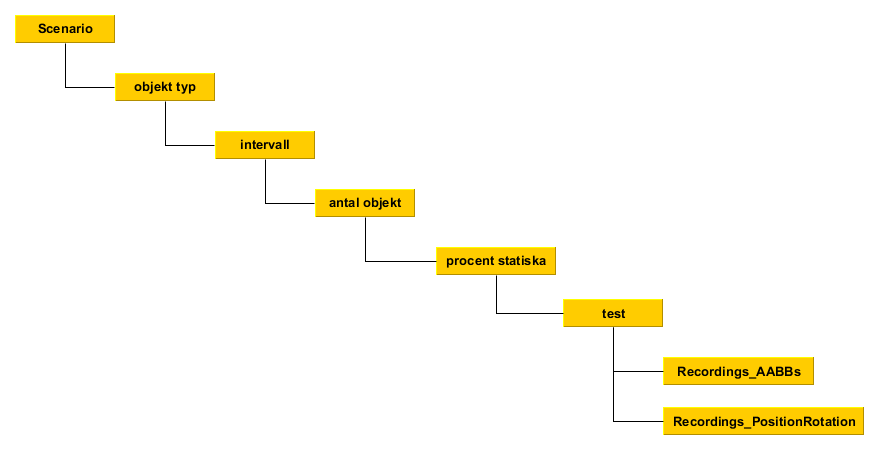


1. Förenklat strukturdiagram som visar de för undersökningen relevanta delarna av simulationsgeneratorn, gröna noderna är *interface* och de noder som är anslutna till de gröna noderna med sträckande linjer är klasser som implementerar det *interface*.

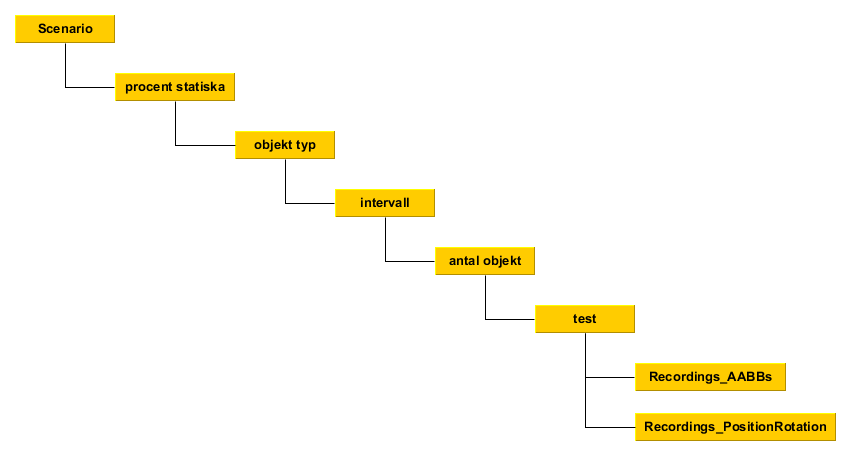
### Automatisering utav testerna

Till följd av den stora andelen testscenarion som ska generas beslutades det att expandera simulationsgeneratorns funktionalitet till att kunna utifrån givna start parametrar automatiskt genera testserier utifrån den 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 återhämtas när testningen väll genomförs.

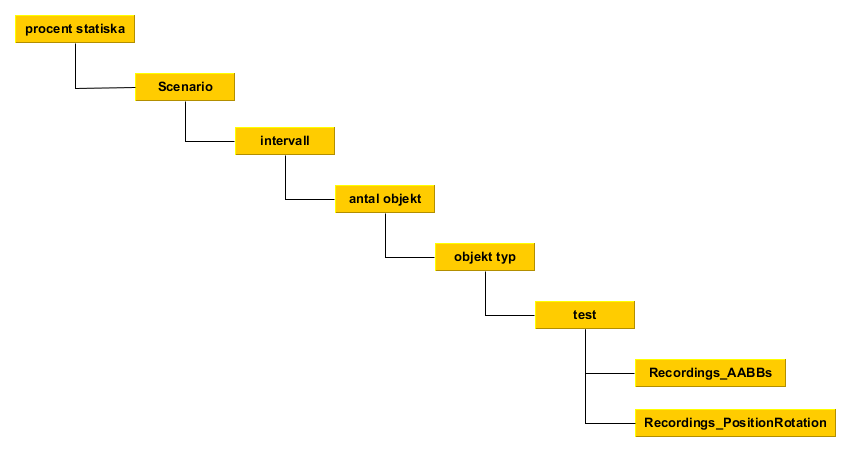
Det första som beslutades var utövningen utav filhierarki till följd av att implementation utav resterande modifikationer kommer vara beroende på hur den utformningen. Tanken med filhierarki är att underlätta användningen av informationen som genraras, vilket resulterade i att den reviderades ett par 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 iteration som filhierarkin genomgick kan ses i **figur 14** till **figur 15**.



1. Första iterationen av filhierarkin



1. Första iterationen av filhierarkin

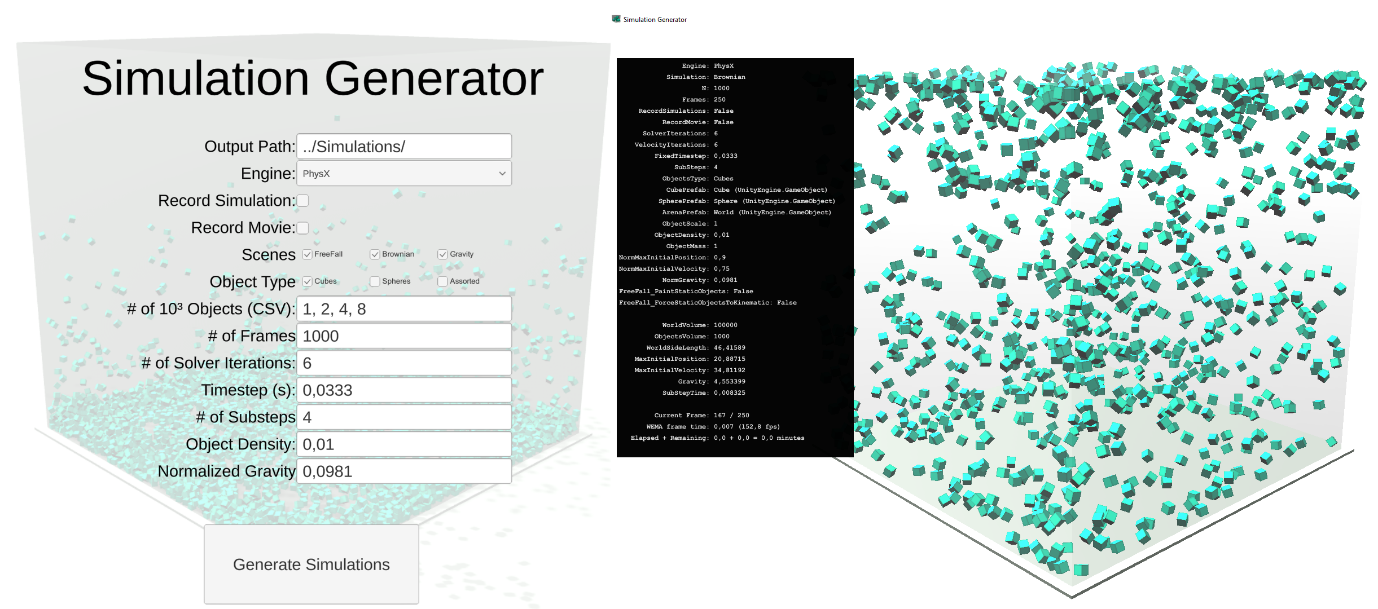


1. Slutliga versionen av filhierarkin som den modifierade simulationsgeneratorn sparar ner scenariona i.

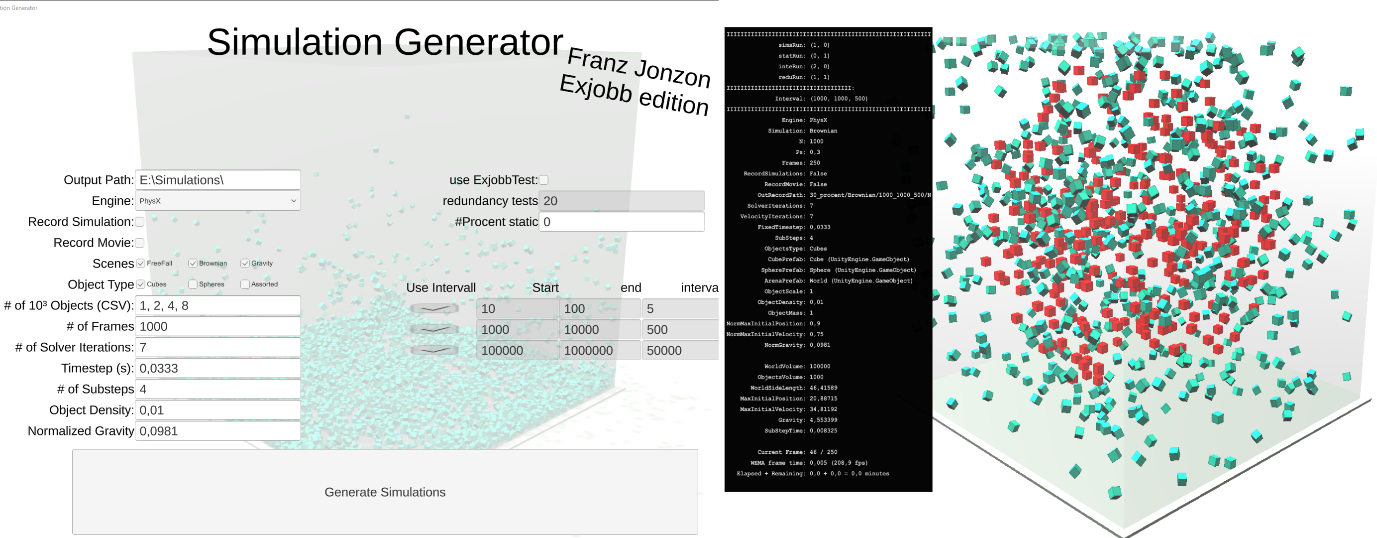
Efter att filhierarkin adderas sedan de parametrar som behövdes för att automatisera testningen. Parametrarna som behövde adderas är,

* Andelen gånger testerna ska upprepas samt.
* Vilka förhållanden mellan statiska och dynamiska objekt som ska testas.
* Tre intervall parametrar vilka består av ett start värde, ett slut värd 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 på variabler så att de går att avaktivera intervalparameterara infall inte alla behöver användas, det gjordes även för att underlätta testningen av den nya automatiseringsdelen. Slutligen lades även en av och på variabel till så att det går att växla mellan en nya automatiserade och den ursprungliga simulationsgenerationen, detta lades till för att kunna snabbt och enkelt generera enskilda testscenarion för vilka den automatiserade generationer är överflödig. **Figur 17** visar på hur simulationsgeneratorns utseende ut innan des funktionalitet expanderades och **figur 18** visar på des utseende efter.



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