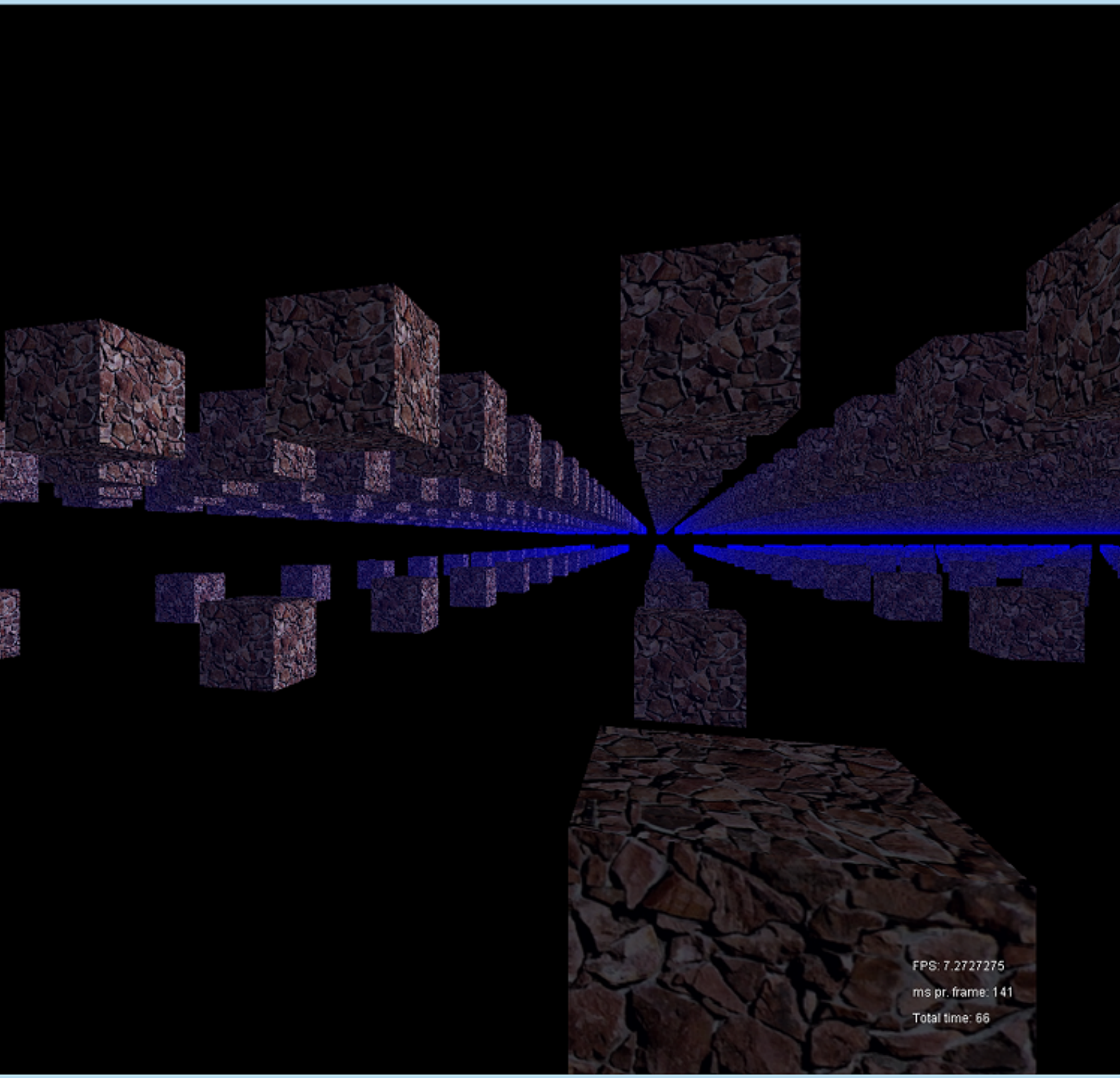
3D-rendering i Java



Roskilde Universitetscenter, Datalogi, 2. modul, Maj 2005.

Vejleder: Bjørn Christensen

Gruppemedlemmer:
Martin Gunneskov, Bjarne Albjerg Falch Hennings og
Kim Sven Russel.

Abstract

Dansk

Projektets mål er at afdække performanceforholdet mellem de to Java3D-implementeringer DirectX og OpenGL, samt undersøge hvorvidt Javas virtuelle maskine 1.5.0_03 performer bedre end den forrige version 1.4.0_06. For at afdække disse performanceforhold, har vi udviklet et benchmark-program til 3D-rendering, der netop er udviklet så det kan teste renderingshastigheden ved brug af DirectX eller OpenGL med tilhørende Java version.

Inspirationen er hentet fra en opgave udarbejdet af Jacob Marner skrevet i 2002 ”Evaluating Java for Game Development”. I denne rapport postuleres bl.a. at Java på sigt vil egne sig som spiludviklingssprog. Om der er grundlag for at benytte Java som spiludviklingssprog anno 2005, vil blive diskuteret i rapporten.

Vi konkluderer at OpenGL-udgaven performer bedre end DirectX-udgaven af Java3D, samt at der kun er en minimal performanceforbedring i JVM-version 1.5.3_03 til JVM-version 1.4.2_06.

English

The project aims to define the performance conditions between the two Java3D implementations DirectX and OpenGL, and furthermore examine the performance of Java’s virtual machine 1.5.0_03 compared to 1.4.2_06. To examine the performance conditions the group has developed a benchmark application which renders 3D. The purpose of the benchmark application is to test the speed of the rendering using DirectX or OpenGL with one of the two Java virtual machines 1.5.0_03 or 1.4.2_06.

The inspiration origins from the report “Evaluating Java for Game Development” written by Jacob Marner in the year of 2002. In this report it is said that Java might be used for game development in the future. Whether or not Java is an appropriate language for game development in 2005 will be discussed in this report.

We conclude that the OpenGL implementation performs better in contrast to the DirectX implementation. It is furthermore determined that the JVM-version 1.5.3_03 only has a minimum of performance improvement compared to the JVM-version 1.4.2_06.

Indholdsfortegnelse

1 Forord

Rapportens målgruppe er personer med Java erfaring. Derfor har vi valgt ikke at oversætte de engelske fagudtryk, som benyttes i rapporten. Ligeledes står vores kommentarer i kildekoden på engelsk, da de derved henvender sig til en større målgruppe.

Det er en fordel at have et elementært kendskab til 3D-principper, men vi har bestræbt os på at give den fornødne information løbende i rapporten. For at hjælpe læseren har vi kursiveret de fagspecifikke udtryk, første gang de optræder i rapporten, hvilket indikerer, at der står en kort beskrivelse af ordet i ordlisten bagest i rapporten.

Ved litteraturhenvisning skrives forfatterens efternavn med årstal for udgivelse, samt hvilket sidetal informationen er fundet i det originale materiale, [Forfatter Årstal, Sidetal]. En lang række detaljeret litteratur om 3D findes på Internettet på troværdige hjemmesider fra firmaer som Sun Microsystems, Nvidia og Microsoft. For ikke at skrive de lange hjemmesideadresser i rapporten benytter vi en reference som [Ref. Nummer], som henviser til referencelisten bagest i rapporten.

1.1 Interview med Deadline Games

Allerede tidligt i processen arrangerede vi et interview med Jacob Marner fra Deadline Games, for at få indsigt i firmaets viden og ekspertise inden for 3D-programmering og performancetestning. Deadline Games er en virksomhed, der udvikler spil til consoller, Mac og PC. Deadline Games blev etableret i 1996, og har bl.a. udviklet det meget anmelderroste adventure-spil "Blackout" og det helt nye 3rd-person-shooter-spil "Total Overdose". [Ref. 06]. Til interviewet fik vi chefarkitekt Jacob Marner stillet til rådighed, som repræsentant for Deadline Games. Jacob Marner har en kandidatgrad i datalogi (cand.scient.dat.) fra Københavns Universitet med speciale i computerspil [Ref. 07]. Han har bl.a. skrevet projektet "Evaluating Java for Game Developement" [Marner 2002], som er grundlag og motivation for vores egen undersøgelse. Dette kommer vi nærmere ind på i afsnit 2.1 "Forundersøgelse" side 4. Vi vil gerne takke Deadline Games og især Jacob Marner for et givtigt samarbejde. En meningskondensering af interviewet kan læses i Bilag 16.1.

2 Indledning

Computerspilsgrafik har i de seneste år gennemgået en udvikling, hvor man efterhånden er blevet i stand til meget realistisk at visualisere virtuelle tredimensionelle (*3D*) verdner. Der er her bl.a. tale om at visualisere naturlige elementer såsom vand, hvor det drejer sig om at vandet skal opføre sig og ligne vand fra den fysiske verden. Dvs. at simulere en 3D-verden indeholdende bl.a. vand med avancerede effekter som spejl-effekt og bølge-effekt, er, udregningsmæssigt en meget krævende proces. En anden væsentlig faktor i computerspil er netop hastigheden. Spillet skal kunne levere flot og realistisk grafik, men også være i stand til at gøre dette i en hastighed, som gør det muligt at bruge spillet. Computerspilsbrugerens computer skal derfor arbejde på højtryk for at levere denne grafik i en passende hastighed. Dermed er computerkraften en begrænsende faktor for kvaliteten af de billeder, der kan leveres.

De fremmeste computerspilsproducenter konkurrerer om, hvem der kan skabe den flotteste og mest realistiske grafik i deres spiludgivelser. Computerspilsentusiaster er derfor ofte nødt til at opgradere deres hardware, hvis de ønsker at være i stand til at spille de nyeste computerspil på markedet. Dette betyder også, at computerspilsproducenterne må sørge for, at deres computerspil afvikles så hurtigt, som muligt. De skal altså bruge en del tid på at optimere det pågældende spils performance. Da computerspillets performance er så vigtig, er det anvendte programmeringssprog til spiludvikling oftest *C/C++*, hvor dele af computerspillet kan være skrevet i *assembler*, for at kunne kontrollere alle computerens ressourcer og dermed optimere den pågældende kodestump.

En anden væsentlig faktor indenfor spiludvikling er dog også, at producenterne ønsker, at publicere deres nye spil så hurtigt som muligt. Samtidig skal man, ud fra et økonomisk synspunkt, tage højde for, hvor længe spillet er under produktion. I denne forbindelse er C/C++ og assembler ikke de bedste programmeringssprog. Det er her Java kommer ind i billedet. Java har et meget højt *abstraktionsniveau*, hvilket gør produktiviteten højere end i bl.a. C/C++ [Marner 2002, 74]. En højere produktivitet vil resultere i lavere udviklingsomkostninger fra producenternes side. Spørgsmålet er bare om Java er et velegnet programmeringssprog til spiludvikling, og hvordan Java skal anvendes, hvis det skal bruges til spiludvikling.

Om Java er et velegnet spiludviklingssprog findes der flere meninger om. Diskussionen går på om Java kan anvendes frem for C/C++ og hvilke fordele og ulemper, der er ved henholdsvis Java og

C/C++ i forhold til hinanden. De argumenter, der taler for Java i denne sammenhæng, er, at Java som sagt kan anvendes med højere produktivitet end C/C++ samt, at Java er platformuafhængig, hvilket gør Java-applikationer lettere at portere end C/C++-applikationer. Argumenter, der taler mod Java som spiludviklingssprog, er, at Java *performer* dårligere end C/C++. Det er bl.a. fordi Java ikke kompileres til maskinkode og samtidig ikke uden *JNI* lader programmøren kontrollere de enkelte systemressourcer på samme måde, som det lader sig gøre i C/C++ eller assembler [Marner 2002, 41].

Disse argumenter og fordomme søger Jacob Marner, at afklare i sin rapport om Java, som spiludviklingssprog. Jacob Marner undersøger netop fordele og ulemper ved henholdsvis Java og C/C++ som spiludviklingssprog og foretager performancetest af de to sprog – bl.a. i forbindelse med 3D-grafik.

2.1 Forundersøgelse

Jacob Marner afdækker i sin rapport, "Evaluating Java for Game Developement", selve performanceforholdet mellem Java og C++ med flere forskellige test. Han tester C++ og Java i 3D-sammenhæng ved at udvikle samme applikation i to udgaver til Java, hvor der anvendes *OpenGL*-bindinger via henholdsvis GL4Java og Java3D, og en udgave til C++, hvor C++-applikationen har en direkte forbindelse til OpenGL-*API*'et.

Resultatet af testen var at C++-applikationen kørte hurtigst. Herefter kom *GL4Java*-applikationen og *Java3D*-applikationen var langsomst. Faktisk var Java3D-udgaven hele 2.5 gange så langsom som C++-applikationen [Marner 2002, 68].

Jacob Marner konkluderer i sidste ende, at Java er langsommere end C++, og dermed ikke på denne baggrund kan anbefale Java til spiludvikling, hvor performance er i højsædet. Til gengæld fastholder han, at der kan reduceres i produktionsomkostningerne ved at anvende Java, da Java fordrer produktiviteten for programmørerne [Marner 2002, 74]. Det er vigtigt at pointere at Jacob Marner skrev sin rapport i marts 2002, og vi talte i interviewet om, at man kunne forestille sig, at Java-teknologien er forbedret siden [Pers. kom. Marner 2005].

2.2 Problemfelt

Projektet tager udgangspunkt i Jacob Marners undersøgelser, og vil besvare nogle af de spørgsmål, som han ikke kommer ind på i sin rapport.

Java3D API'et er interessant, fordi det tilbyder et meget højt abstraktionsniveau i forhold til 3D-programmering. Dette kommer vi nærmere ind på i afsnit 6 "Java3D API" side 10. Java3D kan køre med *DirectX* eller OpenGL og overgangen fra den ene til den anden er transparent for programmøren. Marner tester performance af OpenGL-implementationen af Java3D, men beskæftiger sig ikke med DirectX-implementationen. Det interessante spørgsmål her, er, hvilken af de to implementationer af Java3D, der performer bedst i forbindelse med Java som spiludviklingssprog. Dette vil vi søge at besvare i denne rapport.

Javas virtuelle maskine (*JVM*) har gennem tiden haft stor indflydelse på Java-applikationers performance og har hele tiden været under forbedring. Faktisk er det vurderet, at JVM-version 1.0 er ca. 20 - 40 gange langsommere end C++-kode, og at JVM-version 1.4.2_06 kun er omkring 0,5 - 3 gange langsommere [Marner 2002, 33 og 35], hvilket må siges at være en kraftig forbedring. Spørgsmålet er om disse tal kan forbedres yderligere i fremtidens JVM'er. Dette spørgsmål belyses i denne rapport ved at undersøge om den nyeste version af JVM, 1.5.0_03, er hurtigere end den tidligere version, 1.4.2_06, i forbindelse med Java3D i afvikling af en 3D-applikation.

Dette leder os frem til følgende problemformulering:

2.3 Problemformulering

> *Hvilken af Java3D API'ets implementeringer DirectX og OpenGL, renderer hurtigst?*
> *Og kan der opnås en performanceforbedring ved rendering af 3D-grafik via Java3D ved at anvende JVM-version 1.5.0_03 frem for JVM-version 1.4.2_06?*
>
> *På baggrund af disse spørgsmål diskuteres Java som spiludviklingssprog anno 2005.*

3 Metode

I dette afsnit vil vi overordnet redegøre for, hvordan vi vil arbejde os frem til at besvare problemformuleringen. For at kunne begrunde vores valg, kræver det et indblik i den grundlæggende teori, såsom beskrivelse af hvad *3D-rendering*, Java3D, DirectX og OpenGL er. Derfor efterfølges dette afsnit af nogle teoretiske afsnit, som skal lede op til en refleksion over og begrundelse for vores valg i afsnit 9 "Beskrivelse af test og Java3DTest" side 14.

Vi vil programmere et *benchmarkprogram* i Java3D, som kan teste performance af DirectX og OpenGL i forskellige skærmindstillinger ved at rendere et 3D-univers. Detaljer omkring testene bliver nærmere uddybet i afsnit 9 "Beskrivelse af test og Java3DTest" side 14. Formålet med testene er at give en indikation af eventuelle performanceforskelle mellem Java3D's to implementationer DirectX og OpenGL.

Udover at vi ønsker at teste på eventuelle performanceforskelle mellem DirectX og OpenGL, vil vi også teste performanceforskelle mellem JVM-versionerne 1.4.2_06 og 1.5.0_03. Dette har dog ikke nogen direkte indvirkning på, hvordan benchmarkprogrammet skal programmeres. Det vigtige er dog, at programmet skal kunne kompileres i både 1.4.2_06 og 1.5.0_03, uden at der skal laves ændringer i koden. Alle testene bliver derfor kørt i begge versioner af JVM.

3.1 Afgrænsning

Dette afsnit skal tydeliggøre hvilke parametre vi har afgrænset os fra i projektet. Der vil blive fokuseret på de ting, vi vurderer, at læseren vil have mest gavn af at vide inden rapporten studeres.

For at kunne give et generelt billede af hvordan DirectX og OpenGL performer på nutidens computere skulle vi udføre test på samtlige systemopsætninger. Vi afgrænser os dog fra at se på samtlige systemer, da det tidsmæssigt vil være en umulighed. Som det vil blive beskrevet i afsnit 8 "Benchmark – test af performance" side12, er der flere måder at teste systemer på, og vi har valgt at sammenligne to systemer og teste performance.

Der vil ikke blive programmeret en komplet spil-engine, som også har fysikberegninger med. Grundlaget for denne afgrænsning underbygges af, at det udelukkende er DirectX og OpenGL, som er grafikrelaterede, der testes på, og det vil derfor ikke give mening at arbejde med fysikberegninger.

Da projektet søger at afdække hvilke eventuelle performanceforskelle, der er mellem DirectX og OpenGL i Java3D API'et, har vi valgt ikke at sammenligne med andre programmeringssprog.

Vi har valgt at undlade at teste en ren softwareudgave til rendering, da vi ud fra interview med Jacob Marner, se Bilag 16.1, fik oplyst, at en ren softwareudgave er alt for langsom i forhold til 3D-rendering.

4 3D-rendering

Renderingen er den sidste fase i 3D-grafik-produktionen. Når den grafiske fremstilling af en scene er programmeret skal den renderes for at producere det færdige billede. Overordnet set inkluderer rendering de datalogiske 3D-spilprincipper skygger, *texturing* objekter, lys og opsætning af kameraer. Den sidste proces ved rendering er fasen hvor flader, objekter, lys og bevægelse bliver visualiseret. Kunsten ved rendering er at finde en balance mellem den visuelle kompleksitet, der er nødvendig, og renderingstiden, der bestemmer hvor mange *frames*, der kan renderes i en given periode af tid [Ref. 16].

Disciplinen rendering indebærer altså en større mængde af komplekse udregninger, som computeren skal kalkulere. Computeren skal derfor arbejde på højtryk, for at der kan vises nogle grafisk flotte og flydende scener.

Softwarerendering producerer grafiske billeder med den højeste kvalitet. Her beregnes de komplekse udregninger på CPU'en i modsætning til hardwarerendering, der bruger maskinens grafikkort. Softwarerendering er ofte mere fleksibelt end hardwarerendering, da den ikke er afhængig af computerens grafikkort. Ulempen ved softwarerendering er derimod at renderingen generelt tager længere tid. Hardwarerendering er generelt hurtigere end softwarerendering, men producerer typisk billeder af dårligere kvalitet. Hardwarerendering kan for eksempel ikke, som softwarerendering, producere nogle af de mest sofistikerede effekter, som avancerede skygger og refleksioner [Ref. 16].

To af de mest anvendte typer af 3D-rendering er, ray tracing og polygonmodellering.

Ray tracing minder på sin vis om den fysiske verden, idet der modelleres stråler af lys, bare omvendt. I stedet for at modellere lysstråler fra lyskilden til øjet, sker det modsatte, hvor man modellerer stråler fra øjet til den fysiske verden.

Som man kan forestille sig er det at modellere lysstråler for hver pixel på skærmen meget krævende for computeren. Selvom ray tracing ikke er almindelig i *realtime* 3D-grafik, kan det give nogle ganske realistiske resultater. Ray tracing blev bl.a. brugt til den amerikanske animationsfilm Ice Age fra 2002 [Brackeen 2003, 335f].

Den 3D-renderingsteknik der er almindeligst i computerspil, og de fleste film med computergenererede billeder er polygonmodellering. Med polygonmodellering er den virtuelle 3D-verden fremstillet som rette polygoner. Et polygon er en lige, lukket form med mindst tre sider. Brugen af polygoner øger renderingshastigheden, men runde objekter, såsom kugler, er, vanskelige at oversætte til en polygonmodel. Dette kan dog løses ved at benytte så mange polygoner, at der opbygges en illusion af en rund kugle.

Polygonmodellering er mindre krævende end ray tracing og kan performanceoptimeres til et højt niveau ved at benytte grafikkortets 3D-acceleratorfunktioner. Det er derfor ofte en god idé at benytte grafikkort med 3D-acceleratorfunktioner, hvilke er at finde på næsten alle computere i dag [Brackeen 2003, 335f].

5 DirectX og OpenGL

Da projektet fokuserer på hvilken af Java3D's udgaver, DirectX eller OpenGL, der performer bedst, er det på sin plads at give en kort beskrivelse af disse to bagvedliggende API'er. Dette skal lede op til en sammenligning med vægtning på forskellen mellem dem.

5.1 DirectX

Gennem funktioner i skærm- og lydkort giver DirectX mulighed for at programmer kan vise 3D-grafik og levere musik- og lydeffekter, som forbedrer udnyttelsen af computerens multimedieegenskaber ved f.eks. computerspil og film. Teknologien blev introduceret i 1995 og kan køre på alle Windows-baserede computere med DirectX-kompatible hardware og drivere [Ref. 01]. DirectX er Microsofts egen teknologi, som løbende optimeres og er en integreret del af Windows- platformen.

DirectX gør det muligt for et program at bestemme computerens hardwarefunktioner, og indstiller derefter programparametrene tilsvarende. Det er altså DirectX's grundlæggende funktion, at få forskellig hardware til at kommunikere sammen, hvilket gøres med et sæt API'er på lav-niveau. API'erne fungerer som en form for kommunikationsbro mellem hardware og software og giver multimedieprogrammer forbedret adgang til funktionerne i højtydende hardware, som derved bedre udnyttes. API'erne styrer funktioner på, bl.a. understøttelse af forskellige inputenheder, f.eks. joysticks, tastaturer og mus, og styring af lydmixing og afspilning. Disse funktioner på lav-niveau understøttes af de komponenter, som samlet udgør DirectX [Ref. 02]:

- DirectDraw API'et understøtter direkte adgang til acceleratorfunktionerne i skærmkortet og giver f.eks. spil og digitale videocodecs adgang til funktionerne uden oplysninger fra brugeren om enhedens funktioner.

- Direct3D er et 3D-API på lav-niveau, der giver en grænseflade til de 3D-gengivelsesfunktioner, der er indbygget i moderne skærmkort og understøtter specialiserede CPU-instruktionssæt.

- Derudover indeholder DirectX komponenterne DirectSound API'et, DirectMusic API'et, DirectInput API'et, DirectPlay API'et og DirectShow API'et, som uddybes i Bilag 16.2.

DirectX har sit eget diagnosticeringsværktøj til styring af komponenterne og 4. August 2004 udkom version DirectX 9.0c.

5.2 OpenGL

OpenGL står for Open Graphics Library og er et multiplatform 2D- eller 3D-software interface til grafisk hardware, oprindeligt udviklet af Silicon Graphics, Inc. Siden OpenGL blev introduceret i 1992 er det blevet anvendt i industrien af f.eks. Sun Microsystems, Microsoft og Intel [Ref. 03].

OpenGL har inkorporeret et bredt sæt på flere hundrede procedurer og funktioner som rendering, texture mapping og speciel effekter [Ref. 03], [Ref. 05].

Et typisk program som bruger OpenGL begynder med et kald til at åbne et vindue indeni *framebufferen*, som programmet kan tegne på. Derefter foretages der kald til at allokere grafikbiblioteker, som associeres med vinduet. Herefter kan der udføres OpenGL kommandoer, som f.eks. at tegne et simpelt geometrisk objekt eller påvirke renderingen, hvilket kan indebære belysning eller farve, eller hvordan objekterne bliver vist på computerskærmen. Der findes også kald der kan påvirke den direkte kontrol af framebufferen, såsom at læse og skrive *pixels* [Ref. 04]. OpenGL version 2.0, udkom 7. september 2004.

5.3 Forskellen mellem DirectX og OpenGL

Den største forskel er, at DirectX er en samling af API'er for bl.a. 3D-grafik, lyd, input og netværk, mens OpenGL er et 3D-grafik API. Det er altså komponenten Direct3D i DirectX der teknisk kan sammenlignes med OpenGL. En anden, men mere vigtig, forskel er at OpenGL kan køres på flere forskellige platforme, såsom Windows, Mac og Linux, mens DirectX kun kan køres på Windowsmaskiner. DirectX bliver opdateret hvert eller hvert andet år for at få inkorporeret nye egenskaber, mens OpenGL bliver opdateret meget sjældnere, dog kan nye egenskaber i OpenGL benyttes vha. extensions, der fungerer som opdateringer. OpenGL er en åben standard, hvor man kan se kildekode og læse uddybende dokumentation og det er DirectX ikke.

6 Java3D API

I Java er de to almindeligste måder at drage nytte af grafikkortets 3D-acceleation på, enten via Java3D, hvor der enten bruges DirectX eller OpenGL til at rendere 3D-grafik, eller via en OpenGL-binding – eksempelvis GL4Java. [Brackeen 2003, 336].

Java3D er et API udviklet af Sun Microsystems. API'ets formål er at give programmøren mulighed for at udvikle 3D-applikationer og appletter i et høj-niveau programmeringssprog og på et højt abstraktionsniveau. Java3D er ikke en del af Javas standard *SDK*, men er en tillægspakke. Som nævnt anvender Java3D enten DirectX eller OpenGL til at rendere 3D-grafik. Derfor findes Java3D i to separate udgaver - Java3D DirectX og Java3D OpenGL. Kun en af disse udgaver kan være installeret af gangen.

I Java3D fokuseres der særligt på at rette programmørens fokus mod designet af applikationen og væk fra detaljer omkring selve 3D-renderingen. Java3D tager sig dermed af detaljerne omkring rendering, og programmøren kan koncentrere sig om udformning og placering af 3D-objekter frem for at skulle koncentrere sig om detaljerne omkring, hvordan 3D-verden og objekter skal renderes [Ref. 09].

6.1 Java3D og performance

Java3D tilbyder et højt abstraktionsniveau, hvor renderingsteknikken på forhånd er defineret. Det er altså ikke meningen, at man som programmør forsøger at optimere denne del af applikationen. Pro-

grammørens opgave simplificeres, men spørgsmålet bliver så om dette forsøg på at tilbyde en generel og generisk renderingsfunktion ikke svækker selve applikationens performance, i forhold til at optimere renderingen specifikt til den pågældende applikation. I Sun Microsystems' egen specifikation af Java3D hævdes det, at performance af 3D-renderingen ikke svækkes af denne grund.
I forbindelse med spiludvikling hævder Sun Microsystems, at Java3D vil performe acceptabelt i de fleste sammenhænge. De erkender dog, at Java3D's performance ikke vil leve helt op til den performance, som kan opnås ved at anvende specielle ikke-portable teknikker i og med, at Java3D er et generelt API, men at forhold som at være portabel, produktionstid og produktionsomkostninger skal vægtes imod opnåelsen af optimal performance.

Det er vigtigt at nævne, at vi ikke undersøger Java3D's performance i sammenligning med eksempelvis performance af OpenGL-bindinger til Java eller C++-implementeringer af DirectX og OpenGL [Ref. 09].

7 Java3D, DirectX og OpenGL

Alle versioner af Windows har siden Windows 95 understøttet OpenGL. For at grafikken kører optimalt skal den nyeste version være installeret, og det samme gælder for DirectX [Ref. 10], [Ref. 11].

Der er fordele og ulemper ved at benytte Java3D frem for API'er med lavt abstraktionsniveau – eksempelvis DirectX og OpenGL. En økonomisk fordel ved Java3D er at udviklingsomkostningerne reduceres, fordi meget af koden allerede er skrevet i form af det API'erne udfører og det er på den måde lettere at påbegynde en ny 3D-applikation. Java3D API'et er designet med *multithreading*, så der på et hvilket som helst tidspunkt, kan køre parallelle tråde, som udfører forskellige opgaver [Ref. 11].

Under interviewet med Jacob Marner fortalte han os at *multiprocessing* formodentlig er fremtiden [Pers. kom. Marner 2005]. Derfor er Java3D fremtidssikret på det område, idet det bliver oplagt at arbejde med multithreading.

Der er også nogle ulemper med Java3D og et væsentligt kritikpunkt er at Java3D ikke bliver opdateret lige så ofte som DirectX og OpenGL gør, hvilket betyder, at mange at de seneste 3D-features ikke er tilgængelige for Java3D-programmøren, før næste opdatering. En anden ulempe er, at Ja-

va3D er designet til en standard 3D-hardwaredriver for enten DirectX eller OpenGL, som ikke indeholder fejl. Indeholder driveren fejl, hvilket ikke er unormalt, virker 3D-applikationen ikke og man må vente på, at en ny driver bliver udgivet for sit 3D-grafikkort. Dette er dog ikke så stort et problem, som det har været, idet de store grafikkortproducenter gennem årene har højnet kvaliteten af tilgængelige drivere [Marner 2002, 80].

Vi har konstateret, at begge Java3D's implementationer af DirectX og OpenGL ikke kan være installeret samtidig, og vi er derfor nødsaget til at afinstallere og installere den anden version imellem brugen af dem.

8 Benchmark – test af performance

Nutidens spil bliver stadig mere krævende og derfor tilstræbes det også i højere grad at opnå så høj udnyttelse af performance som muligt. For at teste performance er der adskillige metoder, der samlet går under betegnelsen performancetest. Performancetest udføres for at undersøge hvor hurtigt udvalgte aspekter af et system performer. Disse test kan understøtte forskellige formål. De kan demonstrere om systemet tilfredsstiller nogle opstillede performancekriterier. Man kan sammenligne to systemer og undersøge, hvilket der performer bedst, samt fastslå hvilke dele af systemet, der evt. er flaskehalse og derved begrænser performance. Det er altså vigtigt at have et udgangspunkt for succeskriteriet af performancetesten [Pers. kom. 2005].

Softwareprogrammører benytter adskillige værktøjer til at fastslå hvilke dele af softwaret, der har det højeste tidsforbrug. Disse værktøjer kaldes profileringsværktøjer, hvilket er computerprogrammer, der kan beskrive performance af et andet program. Profileringsvæktøjet kan bl.a. identificere tidsforbruget og frekvensen af brug af forskellige komponenter. Herefter fokuseres på de funktioner, som har det største tidsforbrug, hvilke bliver mål for optimering [Ref. 15].

Profileringsværktøjer varierer dog både i nøjagtighed og specificering. Det er derfor vigtigt at specificere sit performancetestprogram til netop det formål, man ønsker at benytte programmet til. En af de måder man kan måle performance på er via de såkaldte Benchmarktest. Benchmark, der betyder referencepunkt til sammenligning [Ref. 13], giver en metode til at sammenligne performance af forskellige subsystemer f.eks. på tværs af systemets arkitektur.

Ud fra vores interview med Jakob Maner fandt vi, at måleenheden i standard performancetest og performanceanalyser er antallet af millisekunder pr. frame. Umiddelbart kunne man mene, at en passende måleenhed i performancesammenhæng ville være antallet af frames pr. sekund, som den pågældende applikation er i stand til at producere. I computerspilsverdenen er det netop antallet af frames pr. sekund, som ligger til grund for slutbrugerens performancevurdering. Men i selve udviklingsfasen er standardmåleenheden som sagt antallet af millisekunder pr. frame. Denne måling gør det muligt også at afgøre, hvilke dele af applikationen der bruger tid, samt hvor i applikationen det er muligt at reducere tidsforbruget. Antallet af millisekunder pr. frame er også et godt sammenligningsgrundlag mellem applikationer eller en enkelt applikation i forskellige udviklingstrin.

Forskellen på måleenhederne frames pr. sekund og antal millisekunder pr. frame er dog blot et spørgsmål om, hvad der er lettest at anvende. Ud fra den ene værdi er det muligt at udregne den anden.

I forbindelse med selve udførelsen af performancetesten er det vigtigt, at testen køres et bestemt antal frames frem for en specificeret tid [Pers. kom. Marner 2005]. Dette skyldes, at applikationen kan være i gang med at rendere en frame, når tidsgrænsen er nået, hvilket forårsager en upræcis angivelse af antallet af renderede frames under testen, da applikationen kun er i stand til at måle hele frames. Der vil man forsøge at se på hvor mange frames applikationen nåede at rendere på den pågældende tid, men derved opstår problemet, at den sidste frame måske ikke er blevet renderet færdig og derfor ikke talt med.

Hvis man ønsker at performancetesten skal køre i et bestemt stykke tid, kan det gøres på en sådan måde at den pågældende applikation for lov at rendere den sidste frame færdig, før den stoppes pga. tiden. Dette forårsager at den angivne tidsbegrænsning ikke overholdes med nøjagtighed, men kan variere lige så meget som den tid det tager at rendere én frame. I denne situation opnås en nøjagtig måling af det gennemsnitlige antal millisekunder pr. frame ved at anvende den nøjagtige tid, som applikationen har kørt, samt det antal frames, applikationen nåede at rendere. Denne tilgang anvendes i vores eget benchmarkprogram, hvilket vi kommer nærmere ind på i afsnit 9 "Beskrivelse af test og Java3DTest" side 14.

Alternativt kan man vælge at stoppe applikationen efter et bestemt antal frames. Dermed måler man tidsforbruget præcist i forhold til antallet af renderede frames, og man vil være i stand til at afgøre, præcis hvor lang tid applikationen har været om at rendere det bestemte antal frames.

9 Beskrivelse af test og Java3DTest

Dette afsnit er inddelt i syv underafsnit. De første tre beskriver testene, hvad vores benchmarkprogram gør og hvordan testforløbene foregår. Herefter følger underafsnit om benchmarkprogrammet, Java3DTest i forhold til Java3D, Testudstyret og hvad vores forventninger er til testresultaterne. Det sidste underafsnit beskriver kildekoden.

Vores benchmarkprogram rendere ved polygonmodelering, som omtalt i afsnit 4 "3D-rendering" side 7, hvilket vi har valgt, da polygonmodellering rendere hurtigere og performer bedre end f.eks. ray tracing.

9.1 Hvilke test skal afvikles?

Fra Sun Microsystems' hjemmeside downloader vi de to nyeste versioner, som begge er version 1.3.1, af Java3D's implementationer til DirectX og OpenGL. Som beskrevet i afsnit 7 "Java3D, DirectX og OpenGL" side 11 kan begge versioner ikke være installeret samtidig, så vi skal efter en test med den ene version afinstallere og installere den anden version. Det samme gælder for afvikling af de to JVM-versioner, som også skal testes. Her skal vi også afinstallere og installere. En anden vigtig ting, er, at vi skal sørge for, at benchmarkprogrammet bliver compilet med den JVM-version, som testen afvikles under, da den tilhørende kompiler også er i forskellige versioner. Det er enklere at afinstallere og installere implementationenerne DirectX og OpenGL end JVM-versionerne. Derfor bliver rækkefølgen af testene DirectX og OpenGL med JVM-version 1.4.2_06 og derefter køres OpenGL og DirectX med JVM-version 1.5.0_03.

Vi har også valgt at udføre testene i forskellige skærmindstillinger, fordi jo højere opløsning skærmen er sat til desto flere pixels skal der renderes. Det betyder at grafikkortet skal udføre flere beregninger og deraf presses mere end ved lavere opløsninger. Yderligere har det en renderingsmæssig belastning at benytte højere *farvedybde*, da der skal beregnes mere nøjagtige farver for hver pixel på skærmen. Hvilken indflydelse skærmindstillingerne reelt har illustreres i følgende eksempel:

> Med en skærmopløsning på 800×600 og med en farvedybde på 16-*bit*, hvilket svarer til 2 bytes, skal der allokeres (800×600)×2 bytes hukommelse eller 938KB. Dette er næsten én megabyte hukommelse der skal renderes for hver frame [Brackeen 2003, 58]. Har man derimod en farvedybde på 32-bit skal der allokeres (800×600)× 4 bytes hukommelse, 1876KB.

Med dette ønsker vi at undersøge performanceforskellen mellem DirectX og OpenGL når der renderes i lav skærmopløsning i forhold til høj skærmopløsning.

Vi tester på fire forskellige skærmopløsninger som er:

- 1280×1024 pixels
- 1024×768 pixels
- 800×600 pixels
- 640×480 pixels

Hver af de fire skærmopløsninger testes i farvedybde 32-bit og 16-bit. Det bliver til otte forskellige skærmindstillinger, som er valgt fordi skærm og grafikkort ikke understøttede højere skærmopløsninger eller andre farvedybde værdier.

Det forholder sig sådan at 32-bit farver er det samme som 24-bit farver, hvor der er ekstra 8-bit fyld (padding), så pixeldata stemmer overens med 32-bit. Derfor gør det ingen forskel om det billede vi benytter som texture til objekter er 24-bit eller 32-bit, idet de resterende 8-bit blot fyldes ud når der renderes i 32-bit med et 24-bit billede.

Vi er yderligere interesserede i at køre testene på to forskellige computere med to forskellige grafikkort. På den måde undersøges DirectX og OpenGL på en computer med høj processorkraft og et standard grafikkort og på en computer med lavere processorkraft, men med et hurtigere grafikkort. Sammensætningen af processorkraft og grafikkort er ikke det afgørende, men det er derimod sammenligningen af de to testsæt som hver opstilling producerer. Detaljerne omkring testudstyret beskrives nærmere i afsnit 9.5 "Testudstyr" side 23.

For at få nogle brugbare testresultater er det vigtigt at udelukke tilfældigheder. Derfor skal alle testene køres igennem to gange hver, hvilket hjælper os til at gennemskue evt. tilfældigheder i testresultaterne.

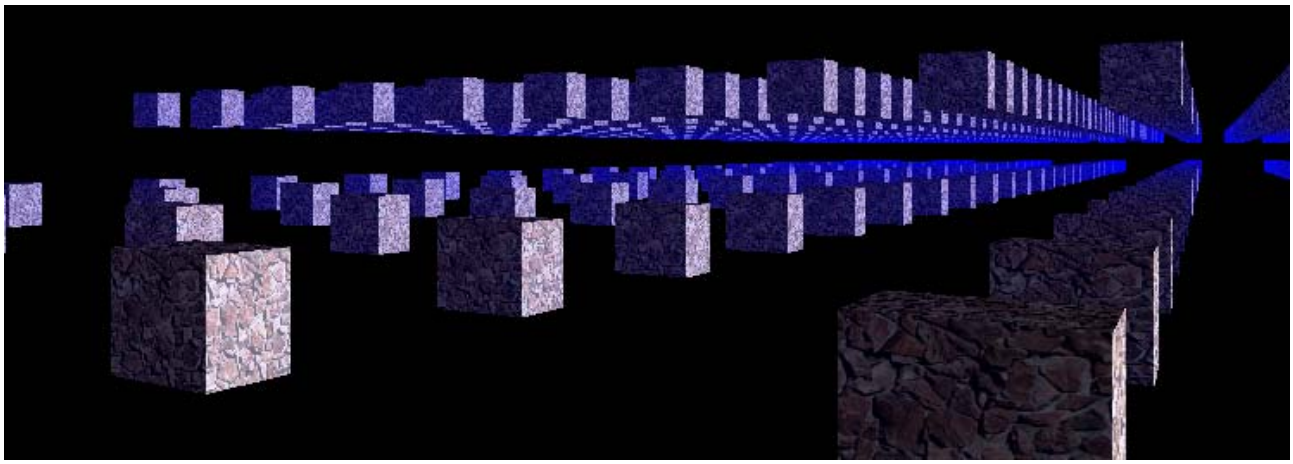
For at opsummere kører vores benchmarkprogram test på hvert af de to grafikkort. Der er otte forskellige skærmindstillinger, som hver giver et testresultat. Hver test køres to gange med en af de fire opstillinger:

- DirectX og JVM-version 1.4.2_06
- DirectX og JVM-version 1.5.0_03
- OpenGL og JVM-version 1.4.2_06
- OpenGL og JVM-version 1.5.0_03

I alt giver det 2 grafikkort × ((8 skærmindstillinger × 4 opstillinger) × 2 gange) = 128 testresultater. Da vi altså arbejder med to test af hver indstilling og benytter gennemsnittet for hver, ender vi med 64 testresultater, som bearbejdes i afsnit 10 "Analyse" side 32.

9.2 Hvad gør benchmarkprogrammet, Java3DTest?

Selve benchmarkprogrammet, Java3DTest, måler performance ved at rendere et 3D-univers bestående af 5000 kasser, som er opstillet på rækker omkring universets centrum, (0, 0, 0), og flytter synsfeltet (kameraet) rundt i universet. Dette skaber en bevægelse, som gør, at Java3D skal foretage udregninger for kassernes positioner i forhold til kameraet. Bevægelsen af kameraet er en cirkulær rotation omkring universets centrum med en radius på 20.
Kasserne er placeret i to lag – henholdsvis 2500 kasser over og 2500 kasser under kameraet. Kameraet bevæger sig således aldrig igennem kasserne, men mellem de to lag af kasser, se Figur 1.



Figur 1: Screenshot fra benchmarkprogrammet

Denne fastlagte bevægelse omkring centrum gør, at renderingsforløbet er statisk og ens for alle testene. Dette er vigtigt, da det udelukker tilfældigheder og uensartet renderingsbelastning. Hvis ikke alle testene udsættes for samme renderingsbelastning, vil resultaterne heller ikke være sammenlig-

nelige. Via det statiske forløb sikres undersøgelsens gyldighed, hvilket betyder, at de testresultater vi får, også svarer til det, vi gerne vil undersøge.

For at få et billede af, hvordan der performes under forskellige renderingsbelastninger, er kasserne placeret således, at der er kasser under kameraet i hele rotationsforløbet, men kun kasser over kameraet i halvdelen af rotationsforløbet. De kasser, der er placeret over kameraet, er placeret med halvt så stort mellemrum som de kasser, der er placeret under kameraet. Dermed skal Java3D rendere mange kasser i den del af rotationen, hvor der både er kasser over og under kameraet.

Forskellige renderingsbelastninger anvendes, da man måske kan forestille sig, at den ene implementering af Java3D, enten DirectX eller OpenGL, ville performe bedre end den anden under en bestemt renderingsbelastning. Således kunne man forestille sig, at den givne belastning ville vise et subjektivt billede af implementeringernes indbyrdes performanceforhold. Ved at anvende uensartet renderingsbelastning minimeres denne faktor og vi får et bredere billede af de forskellige implementeringers performance i flere sammenhænge.

Yderligere udføres testene i fuld skærm – også kaldet ”full screen exclusive mode”. Dette gøres for at sikre, at grafikkortet udelukkende skal beskæftige sig med vores benchmarkprogram og ikke holde Windows-skrivebordet eller andet opdateret sideløbende med testene. Dermed sikrer vi at der ikke i denne sammenhæng opstår fejlkilder i forbindelse med forskellige udefrakommende (andet end benchmarkprogrammet) belastninger for grafikkortet eller systemet i det hele taget. Ligeledes er full screen exclusive mode også den tilstand dagens førende computerspil typisk afvikles i.

For at anvende et bredere sæt af funktionerne i Java3D og i grafikkortet og for at øge den renderingmæssige belastning, anvendes texture på kasserne samt lys- og tåge-effekter. Filen, der bruges som texture, ligger lokalt på harddisken. Denne fil er et billede med en farvedybde på 24-bit, som forestiller murværk.

Af lys-effekter anvendes to forskellige effekter – ambient light og directional light. Ambient light er allestedsnærværende lys, som oplyser kasserne fra alle retninger. Directional light er retningsbestemt lys, som rammer kasserne i en bestemt vinkel. Det allestedsnærværende lys er sat til at være

et svagt lys, mens det retningsbestemte lys er sat til at være stærkt lys. Dermed bliver forskellen og effekten af lysene på kasserne synlig.

9.3 Testforløbet og testdata

I vores benchmarkprogram er rotationshastigheden tidsafhængig og ikke afhængig af renderingshastigheden. Var den afhængig af renderingshastigheden ville rotationshastigheden blive påvirket af renderingsbelastninger, såsom antallet af objekter eller skærmindstillinger, og derved ville testresultaterne blive usammenlignelige. Det er nødvendigt, med testresultater vi kan sammenligne, og derfor er der en anden vigtig faktor, vi skal forholde os til, nemlig vores valg af forskellige renderingsbelastninger i testforløbet, som beskrevet i afsnit 9.2 "Hvad gør benchmarkprogrammet, Java3DTest?" side 16. Det er nemlig altafgørende for at kunne sammenligne, at alle testene kører samme forløb.

Det har stor betydning, når der er forskellige renderingsbelastninger i testforløbet, om testvarigheden er et bestemt antal frames, eller om den er defineret ved tid, som beskrevet i afsnit 8 "Benchmark – test af performance" side 12. Forestiller man sig at testvarigheden er defineret i frames, kan én test rendere flere objekter end en anden, fordi renderingsbelastningerne gør at antallet af frames pr. sekund, varierer. Det kan illustreres i et lille eksempel.

> Testes eksempelvis med skærmindstillingerne 1280×1024×32 i test 1 og 800×600×32 i test 2, og testvarigheden er i frames, vil test 1 rendere færre frames pr. sekund end test 2 pga. af renderingsbelastningen og dermed nå længere i testforløbet. De to test i dette eksempel vil ikke rendere det samme antal objekter, fordi test 2 bliver hurtigere færdig, og der vil ikke være grundlag for sammenlignelige testresultaterne.

Derfor er vores testvarighed defineret i tid, og vi har valgt at testene køres i 10 minutter for hver skærmindstilling. Hvorfor vi lige har valgt, at testen skal forløbe i 10 minutter, er, ikke det afgørende, hvorimod det er vigtigt, at JVM'en får lov til at varme op [Marner 2002, 80]. Det er Jacob Marners erfaring, at Java mindst skal køre i 20 - 30 sekunder, før den virtuelle maskine performer optimalt [Marner 2002, 80], [Per. Kom Marner 2005], og det tager vi forbehold for ved at give benchmarkprogrammet en opvarningstid på ét minut, hvor testforløbet køres igennem, før testen startes. Vi udnytter opvarmningstiden ved at få skrevet frames pr. sekund, millisekund pr. frame og den totale rende-

ringstid ud på skærmen. Udskrivning af data på skærmen, før testen starter, anvendes som en indikation af, om applikationen starter korrekt op, og om frameraten beregnes korrekt. Når opvarmningstiden er forbi, fjerner benchmarkprogrammet selv den føromtalte information fra skærmen og starter testforløbet, alt imens renderingen af kasserne fortsætter.

For at spare tid har vi valgt, at benchmarkprogrammet selv skal kunne fortsætte testforløbet med næste skærmindstilling, når først vi har sat den i gang. Testene kører altså automatisk, indtil vi skal skifte mellem DirectX og OpenGL eller JVM-versionerne. Måden vi har valgt at gøre det på, er, ved at starte benchmarkprogrammet via kommandolinieargumenter i en batchfil, som beskrives nærmere i afsnit 9.7.2 "Batchfilen – opstart af Java3DTest" side 31.

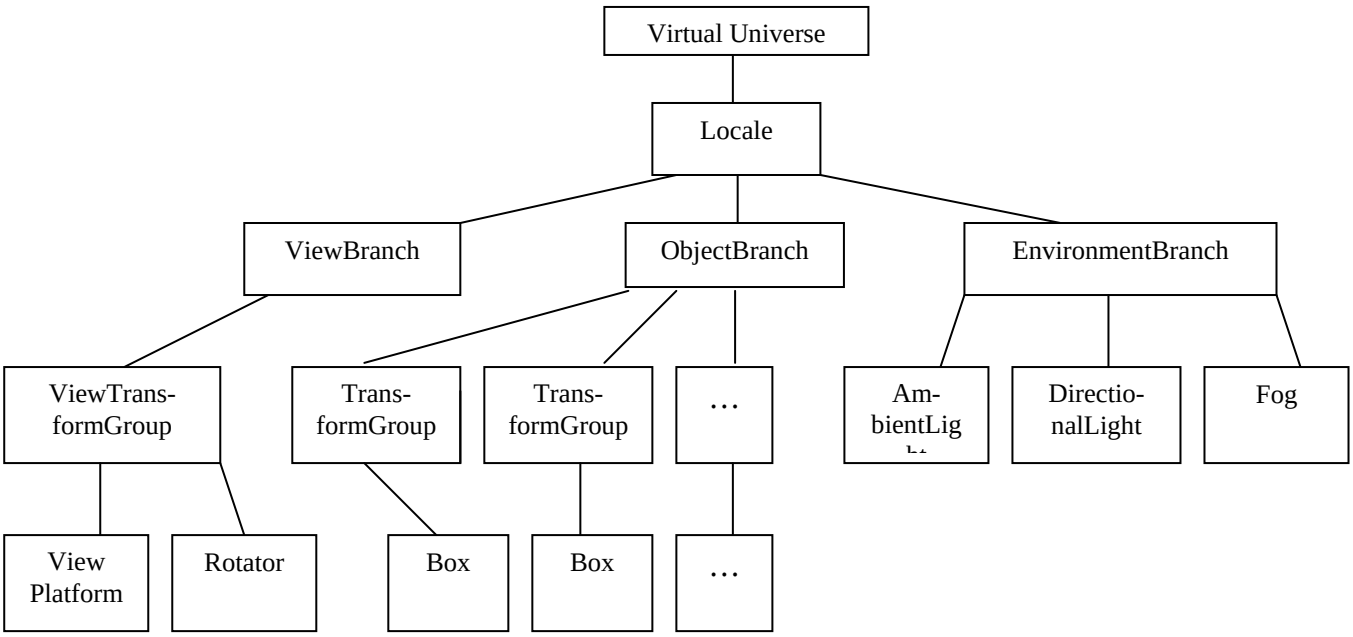
Som nævnt i afsnit 8 "Benchmark – test af performance" side 12 er måleenhenden for performance antallet af millisekunder pr. renderede frame. Disse testresultater udskrives i hver sin tekstfil, fordelt i fire mapper pga. de fire testopstillinger beskrevet tidligere. Der er altså otte tekstfiler i hver mappe. Mappens navn og placering gives også via kommandolinieargumenterne og tekstfilernes indhold beskrives sidst i afsnit 9.7.1.1 "Java3DTest" side 26.

9.4 Java3DTest og Java3D

I dette afsnit gennemgår vi, hvordan Java3DTest er programmeret i forhold til Java3D API'et samt de særlige programmeringsmæssige forhold, som man som programmør arbejder under, når man programmerer 3D-applikationer i Java3D. Helt konkret beskrives Java3D's scenegraf og hvordan scenegrafen ser ud for Java3DTest. Yderligere kommer vi ind på, hvordan vi har anvendt Java3D's særlige optimering i forbindelse med kompilering af denne scenegraf. Vi afgrænser os fra at give en beskrivelse af, hvordan denne optimering i Java3D foregår, men vil forklare, hvordan vi anvender optimeringsmuligheden i Java3DTest.

9.4.1 Scenegrafen

Et af de særlige programmeringsmæssige kendetegn ved Java3D er, at programmøren skal skabe en scenegraf, som afleveres til Java3D, som beregner og renderer scenegrafens elementer. Scenegrafen består eksempelvis af visuelle objekter, effekter, opførsler og lyde. Når Java3D modtager en scenegraf, bliver scenegrafen gennemgået og Java3D beregner diverse opførsler og effekter, renderer visuelle objekter og afspiller lyde. Selve processen i at foretage beregninger og rendere objekter skal programmøren ikke tage sig af. Programmøren skal udelukkende koncentrere sig om at definere selve universet i form af en scenegraf. Figur 2 illustrerer, hvordan vores Java3DTest's scenegraf ser ud.



Figur 2: Scenegrafen indeholder visuelle objekter, effekter og opførsler, som Java3D skal forholde sig til ved rendering af hver frame.

Der er et sæt regler for design af en scenegraf. Først og fremmest skal programmøren oprette et VirtuelUniverse-objekt, som definerer det tredimensionelle univers. Dertil skal der tilknyttes et Locale-objekt, som definerer en lokalitet i universet og dermed et lokalt koordinatsystem, hvor der kan placeres objekter. Dermed kan et univers også indeholde flere lokaliteter. Herefter kan der tilknyttes grene, BranchGroup's, til Locale-objektet. BranchGroup's kan indeholde de enkelte objekter, opførsler m.m. [Ref. 14].

Yderligere skal der tilknyttes et ViewPlatform-objekt, som definerer selve kameraet. Denne kan for overblikkets skyld tilknyttes sin egen BranchGroup.

Efter alt dette kan vi gå i gang med at designe selve universet ved at oprette grene og tilknytte objekter til disse.

Som Figur 2 illustrerer, skabes der i Java3DTest et virtuelt univers med et enkelt Locale-objekt, der har tilknyttet tre grene; ViewBranch, ObjectBranch og EnvironmentBranch.

ViewBranch indeholder et TransformGroup-objekt, ViewTransformGroup, som indeholder ViewPlatform-objektet, samt et RotationInterpolator, rotator, der sørger for kameraets bevægelse rundt i universet. TransformGroup-objektet, ViewTransformGroup, tager sig af kameraets placering i universet og gør det muligt at flytte kameraets position med vores rotator. ObjectBranch indeholder samtlige 5000 kasser i universet. Hver kasse knyttes til en TransformGroup, for at gøre det muligt at angive en specifik placering i universet. EnvironmentBranch indeholder de effekter, der skaber universets miljø. Her er der tale om ambient light, directional light og tåge.

9.4.2 Optimering af performance – kompilering

Som beskrevet i afsnit 6 "Java3D API" side 10 er der lagt vægt på optimering af performance i Java3D. Et eksempel på dette er muligheden for at kompilere sine BranchGroup's – altså de enkelte grene på scenegrafen. Kompileringen gør, at det ikke længere er muligt at få adgang til eksempelvis at læse eller skrive til en TransformGroup's egenskaber – dvs. ændre eller hente eksempelvis positionen af et objekt i universet. Hvis man ønsker at være i stand til at udføre sådanne handlinger på komponenter i en bestemt BranchGroup, skal det klargøres for Java inden kompileringen, at den pågældende TransformGroup kan læses eller skrives til. Ved kompilering foretager Java3D optimeringer på baggrund af de foruddefinerede oplysninger om, hvilke rettigheder der er til de enkelte komponenter i den kompilerede BranchGroup.

I vores eget benchmarkprogram, Java3DTest, sørger vi for at definere de rettigheder, der er nødvendige for at afvikle programmet som ønsket. Der er her kun tale om en enkelt rettighed – nemlig skrivning til ViewPlatform-objektets TransformGroup. Denne rettighed er den eneste, der er nødvendig at definere, da ViewPlatform'en er det eneste vi ønsker at flytte efter kompilering og under programafvikling. Herefter sørger vi for at kompilere alle tre grene på scenegrafen, så Java3D kan foretage så meget optimering, som det er muligt.

Som beskrevet i indledningen afsnit 2.1 "Forundersøgelse" side 4, fik Jacob Marner et resultat omkring Java3D, at det skulle være 2.5 gange langsommere end den tilsvarende C++-applikation. På Jacob Marners hjemmeside [Ref. 08] kan man i hans seneste opdateringer og rettelser til rapporten læse, at han er blevet kontaktet af Sun Microsystems, som forklarer at det dårlige resultat skyldes, at han i sin kode ikke har slået ”collidability” og ”pickability” fra på objekterne i scenegrafen. Collidability relaterer sig til, hvorvidt et givent objekt i universet kan kollidere med andre objekter. Ved at slå denne fra kunne man forestille sig, at Java3D bliver fritaget fra at undersøge om der opstår kollision mellem det pågældende objekt og andre objekter. Pickability relaterer sig til om et givent objekt kan ”tages” med musen af applikationens bruger. I denne sammenhæng kunne man ligeledes forestille sig, at Java3D fritages for beregninger i denne sammenhæng. Derfor giver det umiddelbart god mening, at performance kan forbedres ved at slå disse egenskaber fra. Jacob Marner skriver selv på sin hjemmeside [Ref. 08], at han accepterer Sun Microsystems’ forklaring, da han ikke har tid til at foretage en ny performancetest. Selve problemet med collidability og pickability skulle yderligere være relateret til en *bug* i den Java3D-version, som Jacob Marner anvendte, hvilken var version 1.3 beta 1 i OpenGL-udgaven.

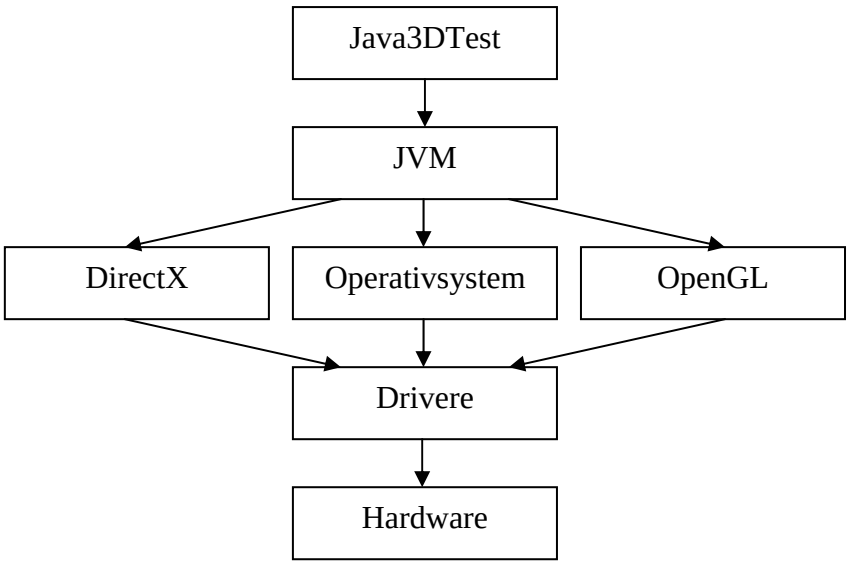
For at undersøge problematikken forud for vores egne test, har vi foretaget en prætest med den Java3D-version, som vi selv anvender, hvilken er version 1.3.1. Vi har foretaget prætesten i OpenGL-udgaven af denne version, da det var denne udgave, som Jacob Marner havde anvendt. Eneste forskel er dermed, at vores Java3D-version ikke er en beta-udgave.

I prætesten undersøger vi, hvilken performancemæssig betydning det har at slå collidability og pickability fra for alle objekter på scenegrafen i vores eget benchmarkprogram, Java3DTest. Dermed udføres to test – en, hvor vi slår collidability og pickability fra for alle objekter, og en, hvor vi slår dem til for alle objekter. Resultatet af testen blev, at tidsforbruget var 91.3141 ms. pr. frame i testen med collidability og pickability slået fra og 87.9692 ms. pr. frame i testen med collidability og pickability slået til. Dermed er der ikke nogen synlig performancegevinst ved at slå collidability og pickability fra. Faktisk viser testresultatet, at performance er en anelse højere, når collidability og pickability er slået til. Umiddelbart er resultaterne ret ens, hvorfor vi må antage, at der ikke er nogen særlig effekt ved at slå collidability og pickability fra. Spørgsmålet bliver i denne sammenhæng om Sun Microsystems’ påstand om, at Jacob Marners resultat ikke er brugbart, eller om der

har været en bug i Java3D, som senere er blevet rettet og derfor ikke har betydning i vores undersøgelse. Vi har undersøgt Sun Microsystems' bug-database og har ikke været i stand til at finde frem til en bug, der relaterer sig til collidability eller pickability og som har med performance at gøre. Uanset hvordan situationen forholder sig, kan vi konkludere, at performance i vores egen applikation ikke påvirkes af, om collidability og pickability er slået fra eller til. Derfor anvender vi Java3D's standardindstillinger for disse egenskaber, hvilke er, at de er slået til.

9.5 Testudstyr

Vores performancetest udføres på to forskellige systemer for at se, hvilken indvirkning forskellig hardwareplatforme har på testresultaterne. Dette har vi beskrevet i afsnit 9.1 "Hvilke test skal afvikles?" side 14. I dette afsnit beskriver vi testsystemernes arkitektur i form af, hvilke hardware- og software-elementer der har indflydelse på vores testresultater. Vi gennemgår en generel systemarkitektur for at beskrive den teoretiske testindvirkning fra hardware og software i systemet. Herefter præciserer vi de to testsystemers specifikationer.



Figur 3: Simplificeret illustration af systemarkitekturen

Figuren viser de elementer i systemarkitekturen, som har en afgørende indflydelse på resultatet af en performancetest. Hvis vi starter fra toppen, er det klart at implementeringen af vores benchmarkprogram, Java3DTest, har indflydelse på resultaterne. Hernæst behandles Java3DTest af den virtuelle maskine, JVM'en, som derfor har indflydelse på performance. Vi har tidligere nævnt, at der gen-

nem tiden har været stor forskel på JVM'ens hastighed, og man har efterhånden en del optimeringer i JVM'ens programafviklingen. Det næste trin i den lagdelte arkitektur er operativsystemniveauet, hvor vi også finder DirectX og OpenGL. JVM'en har sammen med Java3D adgang til funktioner i operativsystemet, DirectX og OpenGL. Det er gennem disse funktionskald, at Java3DTest bliver afviklet. Derfor har operativsystemet samt DirectX og OpenGL en klar indflydelse på systemets performance. For at opnå den højest mulige performance er det derfor en god idé at anskaffe de nyeste versioner af disse. Operativsystemniveauet har adgang til samtlige hardwaredrivere – herunder driveren til systemets grafikkort. Drivernes implementeringer er vigtige for hardwarens performance. I vores tilfælde er grafikkort-driveren af særlig stor betydning. Generelt anbefales det derfor også i performancesammenhæng at holde sine drivere opdaterede. I sidste ende har hardwaren naturligvis en meget afgørende betydning. Hardwarens hastighed – herunder særligt CPU'en og grafikkortet – har en afgørende betydning for systemets performance. Man kunne yderligere forestille sig, at et givent grafikkort ville performe bedre med DirectX end med OpenGL eller omvendt og derved favorisere den ene frem for den anden i vores test. For derfor at give et mere fyldestgørende billede af deres generelle indbyrdes performanceforhold er det nødvendigt at foretage testene på mange forskellige grafikkort. Vores test foretages på to forskellige hardwareplatforme, hvor grafikkortene er forskellige. De to testsystemer navngives på baggrund af de grafikkort, der anvendes i systemerne og specificeres i det følgende.

GeForce4-systemet

CPU: AMD Athlon XP 2000+ 1,67 GHz
RAM: 384 MB RAM
Grafikkort: Asus Geforce4 Ti4200 128 MB
Operativsystem: Windows XP professional

Intel-systemet

CPU: Intel Pentium 4 2,4 GHz
RAM: 512 MB RAM
Grafikkort: Intel 82845G Graphics Controller 64 MB
Operativsystem: Windows XP professional

Som det ses af ovenstående specifikation, anvender vi forskellige processorer og grafikkort i de to testsystemer. Operativsystemet er det samme for begge maskiner, og der er forskel på størrelsen af maskinernes hukommelse. I GeForce4-systemet anvendes et kraftigt grafikkort, som er beregnet til 3D-spil, hvor Intel-systemets grafikkort ikke er decideret beregnet på at skulle anvendes til at køre de mest krævende 3D applikationer, men dog understøtter hardwareacceleration, hvilket er nødvendigt for at kunne udføre vores test. Til gengæld anvendes en hurtigere CPU i Intel-systemet end i GeForce4-systemet, og det bliver derfor interessant at se, hvilket system der performer bedst i testene.

9.6 Forventninger

Vores forventninger til testresultaterne bygger bl.a. på informationer fra vores interview med Jacob Marner, hvor han hævdede at den gængse holdning blandt spiludviklere er, at der ikke er en mærkbar performanceforskel mellem de to implementationer DirectX og OpenGL, se Bilag 16.1. Der skal altså være en gennemgående tendens til at den ene, af implementationerne, performer bedre ved 3D-rendering, før der kan konkluderes på en egentlig performanceforskel.

Til testen mellem JVM-versionerne 1.5.0_03 og 1.4.2_06, forventer vi, at der vil vise sig en forbedring. Dette bygger vi på at, som beskrevet i afsnit 2.2 "Problemfelt" side 5, at Javas virtuelle maskine er blevet kraftigt forbedret fra version 1.0 til version 1.4.2, og derfor forventer vi en forbedring fra JVM-version 1.4.2_06 til JVM-version 1.5.0_03.

Med henblik på de varierende skærmopløsninger i testene, har vi en klar forventning om at benchmarkprogrammet vil bruge flere millisekunder til at rendere en frame i høje skærmopløsninger. Grundlaget for dette bygger på at der ved højere opløsninger skal renderes flere pixels pr. frame, som beskrevet i afsnit 9.1 "Hvilke test skal afvikles?" side 14. Yderligere vil farvedybden have en betydelig indflydelse på renderingshastigheden, da denne også har indflydelse på antallet af pixels pr. frame, se afsnit 9.1 "Hvilke test skal afvikles?" side 14.

Til benchmarktesten benytter vi, som beskrevet i afsnit 9.1 "Hvilke test skal afvikles?" side 14, to forskellige testcomputere, hvor den ene har et helt nyt grafikkort, GeForce4, og en lavere processorkraft end den anden, der derimod har et ældre grafikkort, Intel. Som vi beskriver i afsnit 9.5 "Testudstyr" side 23, har grafikkortet meget stor indflydelse på performance. Dette leder os frem til

en forventning om at testen med det nye grafikkort, GeForce4, vil have nogle lavere renderingstider, i de forskellige test, og deraf performer bedre.

Som det bliver beskrevet i afsnit 8 "Benchmark – test af performance" side 12 er det vigtigt at have et succeskriterium når der testes på performance. For at opsummere er vores succeskriterier:

- Der skal være en gennemgående tendens i testresultaterne til at enten DirectX eller OpenGL performer bedre end den anden. Denne performanceforskel skal yderligere række udover den største generelle afvigelse i testresultaterne. Den generelle afvigelse er forskellen mellem de to nøjagtig ens testsæt vi regner gennemsnittet ud på.

- JVM-version 1.5.0_03 skal performe bedre end JVM-version 1.4.2_06 i samtlige test, med samme betingelser. Dette for at kunne konkludere på om Javas performance er blevet forbedret i den, til dato, nyeste JVM-version 1.5.0_03.

9.7 Beskrivelse af kildekoden

En gennemgang af kildekoden præsenteres i dette afsnit. Vi beskriver de enkelte klasser og metoder samt den batch-fil, som anvendes til at starte applikationen op.

9.7.1 Beskrivelse af klasser

Benchmarkprogrammet, Java3DTest, består af tre klasser; Java3DTest, TestCanvas3D og ScreenManager. I dette afsnit præsenteres de klasser, som udgør programmet Java3DTest. Der gives en kort beskrivelse af hver klasse, samt klassens metoder.

9.7.1.1 Java3DTest

Java3DTest er en klasse, hvis formål er at teste Java3D. For at testene kan køre uden videre interaktion startes applikation med forskellige argumenter fra en batch-fil. En batch-fil kan udføre kommandoer, som er specificeret i tekstform, hvilket vi kommer nærmere ind på senere i afsnit 9.7.2 "Batchfilen – opstart af Java3DTest" side 31. På den måde kan der køres flere test automatisk efter hinanden med forskellige indstillinger. Klassen Java3DTest er ansvarlig for at skabe et virtuelt 3D-univers, starte testen op og afslutte testen ved at udskrive testresultaterne til en fil på harddisken.

```
public static void main(String[] args)
```

Main-metoden får fire argumenter fra en batchfil, som starter applikationen. De fire argumenter er antallet af millisekunder, applikationen skal køre, skærmindstillingen, som skrives med et tal, der refererer til et display mode array, antallet af sekunder applikationen skal varme op, før testen går i

gang og en mappeangivelse af, hvor testdatafilen skal gemmes. Startes applikationen uden argumenter, bruges defaultindstillinger, og mangler der et argument, kaldes metoden `printUse()`. Til sidst oprettes en ny instans af Java3DTest objektet, hvilket starter testen med de overførte testindstillinger.

`public static void printUse()`

Startes applikationen med andet end nul eller fire argumenter udskrives i terminalen en kort beskrivelse af de fire argumenter og deres variabeltype for at indikere, hvordan applikationen skal anvendes.

`public Java3DTest()`

Constuctoren skaber en ny instans af `Java3Dtest`, sætter full screen exclusive mode op med den valgte display mode og sørger for at skabe et virtuelt univers bestående af kasser. I Java3D skal hvert objekt, som skal præsenteres grafisk, have defineret et udseende via en klasse kaldet `Appearance`. I Java3DTest skal alle kasser have samme udseende, hvorfor der blot skabes et enkelt `Appearance`-objekt, som gives som argument, når der oprettes en kasse. Dette gøres i constructoren ved at kalde metoden, `createAppearance()`. Ligesom med `Appearance`, refererer alle kasser til samme Texture-objekt i hukommelsen. Derfor indlæses texture-billedet i constructoren.
Yderligere skabes en ny instans af `TestCanvas3D` med doublebuffering, som tilknyttes det virtuelle univers. Dermed fortæller vi Java3D, at det virtuelle univers, som vi har skabt, skal renderes på vores eget kanvas – nemlig `TestCanvas3D`.

`public void createAppearance()`

Metoden `createAppearance()` skaber et `Appearance`-objekt, som skal deles af alle kasser i det virtuelle univers. `Appearance`-objektet skabes således, at det reagerer på lys på en passende måde ved at angive præcis hvilke farver og med hvilken intensitet, der skal reageres, når en kasse rammes af lys. Herefter indstilles udseendet til at blande farvepåvirkningen udefra med texture-farverne, således at texture på kasserne kan påvirkes af eksempelvis tåge. Endelig tilføjes det fælles texture-object.

`public BranchGroup createSceneGraph()`

`createSceneGraph()`skaber den gren af den samlede graf, hvor de synlige objekter i universet er placeret. Hver enkel kasse skabes ved at kalde metoden `newBox(Vector3d pos)`. Endelig kompileres denne gren, for at Java3D kan foretage optimeringer på grafen.

`public TransformGroup newBox(Vector3d pos)`

Denne metode skaber et nyt `Box`-objekt med sin egen `TransformGroup` og returnerer til sidst denne `TransformGroup`. Det er nødvendigt, at hver kasse (`Box`) har sin egen `TransformGroup` for at kunne blive placeret et bestemt sted i universet, hvilket vi netop ønsker at gøre. Metoden tager et `Vector3d`-objekt som argument, hvilket angiver den nye kasses placering i universet. Kassen tildeles den fælles `Appearance` og placeres i det ønskede punkt.

`public void createUniverse(TestCanvas3D canvas3D, BranchGroup scene)`

Metoden skaber det samlede univers, ud fra de to argumenter, `canvas3D`, som er det kanvas, der skal renderes på, og `scene`, som er den gren af den samlede scenegraf, som indeholder kasserne.
Metoden skaber et univers med tåge, lys og et ekstra retningsbestemt lys. Ud over den gren, scene, som metoden får overført, skabes to nye grene til den samlede graf; `viewBranch` og `environmentBranch`. `viewBranch` er den gren, som holder styr på selve kameraet i universet, som kaldes for en `ViewPlatform`. Metoden sørger for at indstille denne `ViewPlatform`, således at kameraet har startposition i (20, 0, 0) og under testen roterer omkring centrum. Dette gøres ved at skabe en tidsstyret `RotationInterpolator` og koble denne til vores `ViewPlatform`.
`environmentBranch` er den gren, der holder styr på tåge, lys og det retningsbestemte lys i universet. Der skabes en tåge-effekt, et svagt lys, som er allestedsnærværende, og et kraftigt retningsbestemt lys, som har retningen (1, 0, 1). For at afgrænse lys -og tåge-effekterne, og dermed aflaste Java3D for unødvendigt arbejde, oprettes et kugle-objekt, som har centrum i (0, 0, 0) og en radius på 1000, som definerer det område, hvor lyset og tågen har effekt. Til sidst kompileres de to nye grene, så Java3D kan foretage optimeringer på hele grafen.

`public void finish(long testTimeUsed, int testFrames)`

Denne metode kaldes, når applikationen er færdig med at teste. Argumenterne `testTimeUsed` og `testFrames` definerer henholdsvis det tidsforbrug applikationen har haft i testen og det antal frames som blev renderet under testen. Ud fra disse værdier udregnes antal millisekunder pr. frame og antal

frames pr. sekund. Sammen med øvrige data om skærmindstillinger og opvarmningstid, skrives disse ud i en fil, som bliver navngivet i forhold til skærmindstilling, testtid og opvarmningstid, som alle er startindstillingerne til testen. Ved at navngive filen efter startindstillingerne bliver det nemt at differentiere de forskellige testfiler fra hinanden, når alle test er kørt færdigt.

9.7.1.2 ScreenManager

`ScreenManager` er en klasse som initialiserer fuld skærm og grafikindstillinger.

```
public ScreenManager()
```

Opretter et nyt `ScreenManager`-objekt og skaber en reference til systemets grafikkort. Denne reference anvendes senere til at initialisere full screen exclusive mode.

```
public void setFullScreen(DisplayMode displayMode)
```

Sætter systemet i fuld skærm og ændrer skærmindstillingerne. Hvis den specificerede indstilling er null eller systemet ikke er kompatibelt med indstillingen, udskrives en fejlmeddelelse og programmet afsluttes.

```
public JFrame getFullScreenWindow()
```

Returnerer den `JFrame` som anvendes til fuld skærm, eller `null` hvis systemet ikke er i fuld skærm.

9.7.1.3 TestCanvas3D

`TestCanvas3D` nedarver fra `Canvas3D`. Klassen opretter et kanvas til 3D-rendering på skærmen og udregner testdata i form af millisekunder pr. frame, frames pr. sekund, det totale tidsforbrug samt det totale antal renderede frames. I Java3D er det ikke påkrævet at skabe sin egen udgave af `Canvas3D`. I stedet kan man anvende den eksisterende implementering af `Canvas3D`. Grunden til at vi skaber vores egen udgave af `Canvas3D` er, at vi dermed får adgang til de metoder, der bliver kaldt i forbindelse med renderingen af hver enkelt frame. Dette skal vi bruge til at optælle antallet af renderede frames samt afgøre når testen skal stoppes.

```
public TestCanvas3D(GraphicsConfiguration graphicsConfiguration,
long testTime, long warmupTime, Java3DTest java3DTest)
```

Contructoren `TestCanvas3D()` opretter en ny instans af et `TestCanvas3D` med fire argumenter. Det første argument er en grafikkonfigurationen, som sendes videre til superklassen, `Canvas3D`, der opretter og initialiserer et nyt `Canvas3D`-objekt, som Java3D kan rendere på. Det andet argument er antallet millisekunder, som testen skal vare. `warmupTime` er den tid i millisekunder som appikationen skal bruge til at varme op. Det fjerde argument er en reference til `Java3DTest`-objektet, som vi senere skal fortælle at testen er slut.

`public void preRender()`

Metoden overskriver metoden i super-klassen, `Canvas3D`, og kaldes af Java3D renderingsløkken for pågældende frame efter at kanvasset er ryddet og før rendering af framen. Denne metode anvender vi til at måle tidsforbrug og til at sætte testen i gang, hvis opvarmningstiden er slut. Så længe applikationen kører opvarmning, dvs. testen endnu ikke er startet, beregnes løbende antal millisekunder pr. frame, antal frames pr. sekund samt det totale tidsforbrug, ved at kalde metoden `calcStats()`.

`public void postSwap()`

Metoden overskriver metoden i super-klassen `Canvas3D` og kaldes af Java3D renderingsløkken for pågældende frame efter rendering af kanvasset, og da `setDoubleBufferEnable` er sat til `true` bliver det efterfulgt af et buffer swap. I denne sammenhæng anvender vi metoden til lige at tilføje lidt grafik på det ellers færdigrenderede kanvas. Vi udskriver testdata på skærmen, så længe testen ikke er startet ved at kalde metoden `displayStats()`.

`public void postRender()`

Metoden overskriver metoden i super-klassen `Canvas3D` og kaldes af Java3D renderingsløkken efter rendering af kanvas for pågældende frame og før et buffer swap. Denne metode anvender vi til at tælle antallet af frames op.

`public void finish()`

`finish()` kaldes når en test er slut og udregner tiden, applikationen har kørt, før den kalder `finish()`-metoden i `Java3DTest` klassen.

`public void calcStats()`

`calcStats()` benyttes kun før testen startes til at udregne frames pr. sekund, millisekunder pr. frame og samlet tidsforbrug, som skal skrives på skærmen.

`public void displayStats()`

Denne metode udskriver frames pr. sekund, millisekunder pr. frame og samlet tid i sekunder på skærmen. Udskrivningen stopper, når testen går i gang, for at udregningerne ikke får nogen indflydelse på testresultatet.

9.7.2 Batchfilen – opstart af Java3DTest

Programmet startes med en batchfil, for at give Java argumenterne: `-Dsun.java2d.noddraw=true` som er en Java fix og `-Xms128m -Xmx128m`, som er henholdsvis størrelsen af megabytes i hukommelsen, som skal allokeres fra start og max størrelsen. `-Dsun.java2d.noddraw=true` retter en fejl i Java3D, som gør, at grafikkortet ikke kan bruges i full screen exclusive mode. Argumentet slår en særlig DirectDraw-funktion fra i JVM, som skaber dette problem. Denne funktion er dog ikke nødvendig for vores applikation. Med dette argument opnår vi at kunne afvikle Java3DTest i full screen exclusive mode med hardwareacceleration via testsystemets grafikkort. Det sidste som står i batchfilen er Java3DTest argumenterne, som allerede er beskrevet i main-metoden: `numFrames(int) displayModeNr(int) warmupTime(int) path(folder)`.

10 Analyse

De forskellige test er blevet udført på to forskellige computere, med forskelligt hardware, se afsnit 9.5 "Testudstyr" side 23 og i dette afsnit præsenteres testresultaterne. Performance bliver målt i millisekunder pr. frame, jo færre millisekunder der benyttes til at rendere hver frame desto bedre er performance for testkørslen. Der er blevet testet på DirectX med JVM-version 1.4.2_06 og 1.5.0_03 og OpenGL med samme JVM-versioner, der er altså udført fire test på hver testmaskine. Disse test er blevet udført i otte forskellige skærmindstillinger.

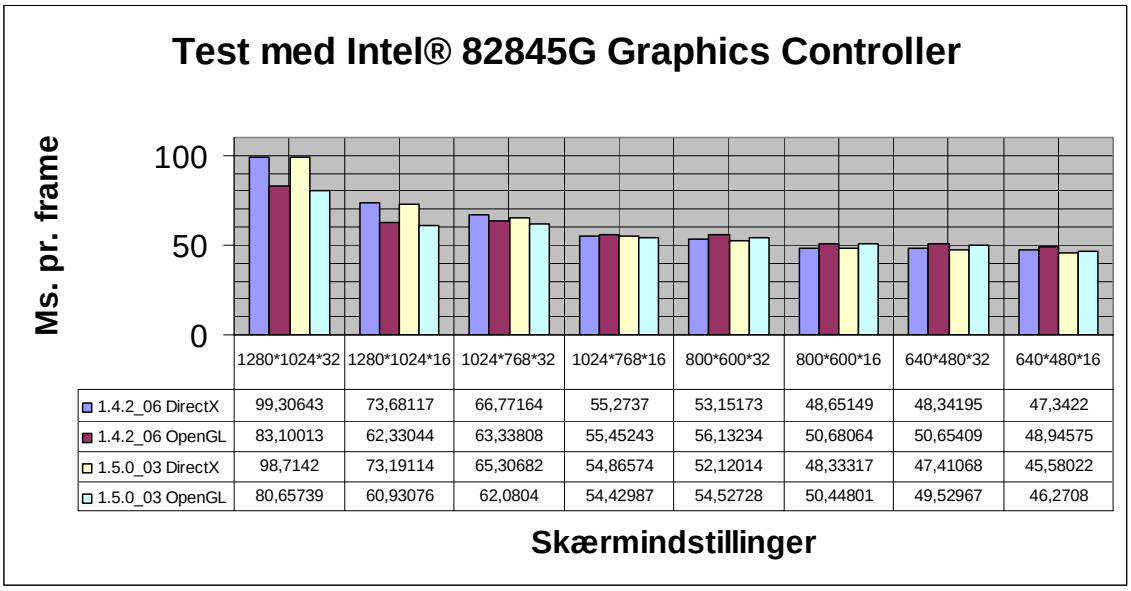
Som beskrevet i afsnit 9.1 "Hvilke test skal afvikles?" side 14 er hver test blevet kørt to gange, hvorefter gennemsnittet af renderingstiden er taget. Generelt var afvigelsen på de to test meget minimal, men den højeste afvigelse var på 0,5 millisekund pr. frame, i testen med Geforce4-grafikkortet med JVM-version 1.5.0 i OpenGL. Dette svarer til en procentvis afvigelse på,

$$\frac{28{,}8204 - 28{,}3385}{28{,}3385} \times 100 \; = 1{,}7\%$$

Som vi skrev i afsnit 9.6 "Forventninger" side 25 anvendes denne værdi som indikator for om de målte performanceforskelle blot er resultater af en generel afvigelse i systemerne.

10.1 Test med Intel-grafikkort

Resultaterne fra testen med Intel-grafikkortet er illustreret på følgende graf.



	1280*1024*32	1280*1024*16	1024*768*32	1024*768*16	800*600*32	800*600*16	640*480*32	640*480*16
1.4.2_06 DirectX	99,30643	73,68117	66,77164	55,2737	53,15173	48,65149	48,34195	47,3422
1.4.2_06 OpenGL	83,10013	62,33044	63,33808	55,45243	56,13234	50,68064	50,65409	48,94575
1.5.0_03 DirectX	98,7142	73,19114	65,30682	54,86574	52,12014	48,33317	47,41068	45,58022
1.5.0_03 OpenGL	80,65739	60,93076	62,0804	54,42987	54,52728	50,44801	49,52967	46,2708

Figur 4: Viser resultaterne fra testmaskinen med Intel-grafikkortet

Resultaterne viser det forventede billede af, at der bruges flere millisekunder til at rendere en frame jo højere skærmindstillingerne er sat.

10.1.1 Analyse af performanceforholdet mellem DirectX og OpenGL

Ser man på forholdet mellem DirectX og OpenGL viser resultaterne tydeligt at OpenGL performer bedre end DirectX i de højeste skærmindstillinger, (1280×1024×32). Med JVM-version 1.4.2_06 er der en forskel på,

$$\frac{(99{,}3064 - 83{,}1001)}{83{,}1001} \times 100 = 19{,}5\%$$

eller ca. 16,2 millisekunder pr. frame, som OpenGL performer bedre end DirectX. En interessant observation er dog at OpenGL kun performer væsentligt bedre end DirectX med skærmindstillingerne (1280×1024×32) og (1280×1024×16). Ser vi på de laveste skærmindstillinger, (640×480×16), er DirectX rent faktisk minimalt hurtigere end OpenGL til at rendere hver frame. I testen med de laveste skærmindstillinger er der en forskel på,

$$\frac{(48{,}9457 - 47{,}3422)}{47{,}3422} \times 100 = 3{,}4\%$$

eller 1,6 millisekund som DirectX performer bedre end OpenGL.

For at give et samlet billede af performanceforskellene tages gennemsnittet af de fire forskellige opsætningers renderingstid. Den gennemsnitlige renderingstid beregnes ved at addere opsætningens testresultater og dividere med antallet af test. Med opsætningen JVM-version 1.4.2_06 med DirectX er gennemsnittet 61,565 millisekunder pr. frame. Samme JVM-version i OpenGL bruger 58,829 millisekunder pr. frame. Dette giver en samlet performanceforbedring ved brug af OpenGL i JVM-version 1.4.2 på,

$$\frac{(61{,}5650 - 58{,}8290)}{58{,}8290} \times 100 = 4{,}6\%$$

Ved brug af JVM-version 1.5.0_03 er den gennemsnitlige renderingstid, med DirectX, 60,6903 millisekunder pr. frame, og med OpenGL 57,359 millisekunder pr. frame. Hvilket giver en procentvis performanceforbedring ved benyttelse af OpenGL og JVM-version 1.5.0_03 på,

$$\frac{(60{,}6903 - 57{,}3590)}{57{,}3590} \times 100 = 5{,}8\%$$

10.1.2 Analyse af performanceforholdet mellem JVM-version 1.4.2_06 og 1.5.0_03

Ud fra testene viser det sig at 1.5.0_03 er lidt hurtigere til at rendere end den tidligere, 1.4.2_06. Med skærmindstillingerne (1280×1024×32) og ved brug af DirectX, er 1.5.0_03,

$$\frac{(99{,}3064 - 98{,}7142)}{98{,}7142} \times 100 = 0{,}6\%$$

hurtigere til at rendere end JVM-version 1.4.2_0, dette svarer til 0,6 millisekunder. Med skærmindstillingerne (640×480×16) er performanceforskellen ca. 3,9%, 1,8 millisekunder, som 1.5.0_03 er hurtigere til at rendere. Samme billede tegner sig når der benyttes OpenGL i testen, også her performer 1.5.0_03 en smule bedre end 1.4.2_06. Forskellen, med skærmindstillingerne (1280×1024×32), er på 0,5%, 2,5 millisekunder, og med (640×480×16) er forskellen 5,8%, 2,7 millisekunder.

Den samlede performanceforskel mellem 1.4.2_06 og 1.5.0_03 ved brug af DirectX udregnes til at være

$$\frac{(61{,}5650 - 60{,}6903)}{60{,}6903} \times 100 = 1{,}4\%$$

Performanceforskellen ved brug af OpenGL er,

$$\frac{(58{,}8290 - 57{,}3590)}{57{,}3590} \times 100 = 2{,}6\%$$

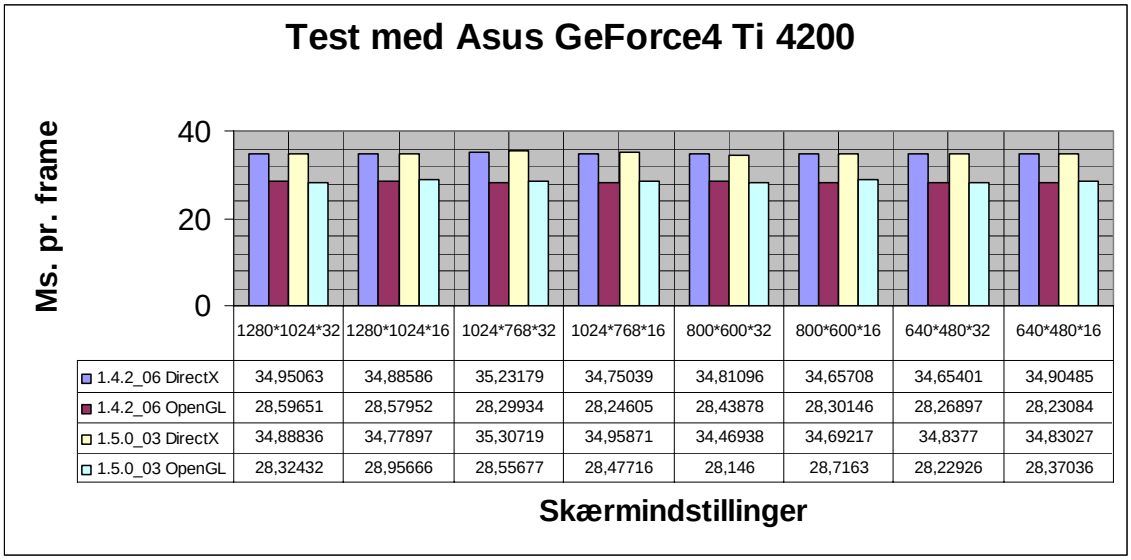
10.1.3 Opsummering af test med Intel-grafikkort

Den opsætning der renderede hurtigst i de højeste skærmindstillinger er altså via OpenGL og JVM-version 1.5.0_03, men derimod performer DirectX med JVM-version 1.5.0_03 minimalt bedre i de laveste skærmindstillinger.

Yderligere er det ved at benytte OpenGL med JVM-version 1.5.0_03, at man generelt opnår den hurtigste renderingstid, og med DirectX med JVM-version 1.4.0_06 man opnår den langsomste.

10.2 Test med GeForce4-grafikkort

I den anden testmaskine har vi brugt en noget langsommere processor, men derimod et nyere og bedre grafikkort, GeForce4, se afsnit 9.5 "Testudstyr" side 23. Resultaterne har dog vist sig at differentiere bemærkelsesværdigt meget i forhold til resultaterne fra testmaskinen med Intel-grafikkortet. Resultaterne er illustreret på Figur 5.



Figur 5: Viser resultaterne fra testmaskinen med GeForce4-grafikkortet

Grafen viser en tydelig performancemæssig forbedring i forhold til resultaterne fra Figur 4. Der er dog ikke samme performanceforskel mellem høje og lave skærmindstillinger, som der blev illustreret med Intel-grafikkortet.

10.2.1 Analyse af performanceforholdet mellem DirectX og OpenGL

Ses der på performanceforskellene mellem DirectX og OpenGL illustrerer Figur 5 at OpenGL renderer hurtigere end DirectX. Forskellen i højeste skærmindstillinger, (1280×1024×32), med JVM-version 1.4.2_06, er,

$$\frac{(34{,}9506 - 28{,}5965)}{28{,}5965} \times 100 \; = 22{,}2\%$$

hvilket svarer til ca. 6,4 millisekunder pr. frame som OpenGL er hurtigere til at rendere hver frame. Med samme JVM-version og med skærmindstillingen (640×480×16) er OpenGL,

$$\frac{(34{,}9048 - 28{,}2308)}{28{,}2308} \times 100 \; = 23{,}6\%$$

hurtigere til at rendere, hvilket er 6,7 millisekunder hurtigere pr. frame. I modsætning til testen med Intel-grafikkortet forbliver OpenGL altså betydeligt hurtigere end DirectX.

Den samlede forskel i renderingshastighed mellem DirectX og OpenGL med JVM-version 1.5.0_03 udregnes til at være,

$$\frac{(34{,}8453 - 28{,}4721)}{28{,}4721} \times 100 \ = 22{,}3\%$$

eller 6,3 millisekunder. Performanceforskellen ved brug af 1.4.2_06 bliver målt til at være 22,4%.

10.2.2 Analyse af performanceforholdet mellem JVM 1.4.2_06 og 1.5.0_03

I forhold til problemstillingen med brug af forskellige JVM-versioner, synes det ikke at have indflydelse på resultaterne, i testen med GeForce4-grafikkortet. Med DirectX og skærmindstillingerne (1280×1024×32) er JVM-version 1.5.0_03,

$$\frac{(34{,}9048 - 34{,}8888)}{34{,}8888} \times 100 \ = 0{,}04\%$$

hurtigere til at rendere en frame. For at give et samlet billede af den minimale performanceforskel mellem 1.4.2_06 og 1.5.0_03, i testen med GeForce4, tages gennemsnittet af resultaterne til sammenligning. Med DirectX er JVM-version 1.5.0_03,

$$\frac{(34{,}8556 - 34{,}8453)}{34{,}8453} \times 100 \ = 0{,}03\%$$

hurtigere til at rendere en frame. Ved brug af OpenGL er 1.4.0_06,

$$\frac{(28{,}4721 - 28{,}3072)}{28{,}3072} \times 100 \ = 0{,}58\%$$

hurtigere til at rendere en frame. Der er altså en begrænset renderingsmæssig forskel mellem brugen af 1.4.2_06 og 1.5.0_03.

10.2.3 Opsummering af test med GeForce4-grafikkort

Testen viste tydeligt at OpenGL performer bedre end DirectX i samtlige test, den samlede afvigelse var på hele 22,3% med JVM-version 1.5.0_03 og 22,4% med JVM-version 1.4.2_03. Der er dog ikke en opsætning der tydeligt performer bedst. Det viste sig yderligere i testen, at der stort set ikke var forskel på performance uanset hvilken JVM-version, der blev benyttet.

I dette afsnit har vi udelukkende præsenteret testresultaterne, og i næste afsnit diskuteres testresultaterne som også sættes op imod vores forventninger til testene.

11 Diskussion

Diskussionen anvendes til at diskutere analyseresultaterne i forbindelse med Java3D-udgaverne - DirectX og OpenGL. Derefter diskuteres resultaterne omkring JVM-versionernes performance. Yderligere bringer vi vores indskud til den eksisterende diskussion om Java som spiludviklingssprog på baggrund af vores undersøgelsesresultater.

11.1 Diskussion af analyseresultater

I dette afsnit forholder vi os til resultaterne i analysen ved at diskutere resultaternes signifikans og gyldighed. Vi vil yderligere betragte resultaterne i forhold til vores forventninger og diskutere hvorfor resultaterne enten stemmer overens med vores forventninger eller ikke gør. Vi diskuterer først resultaterne omkring performanceforskellen mellem de to Java3D-udgaver; DirectX og OpenGL. I denne del kommer vi også ind på skærmindstillingernes betydning for resultaterne. Dernæst diskuteres performanceforskellen mellem JVM-version 1.4.2_06 og 1.5.0_03.

Performanceforskel mellem DirectX-udgaven og OpenGL-udgaven af Java3D

Ud fra analyseresultaterne fandt vi, fra testen med GeForce4-grafikkortet, at der var en væsentlig performanceforbedring ved brug af Java3D med OpenGL frem for DirectX. Der var en klar gennemgående tendens til at OpenGL-udgaven renderede hurtigere end DirectX i samtlige skærmindstillinger. Gennemsnitligt renderede Java3D med OpenGL 22,4% og 22,3% hurtigere, i henholdsvis JVM-version 1.4.2_06 og JVM-version 1.5.0_03, end Java3D med DirectX. Dette indikerer, at OpenGL-udgaven af Java3D er implementeret bedre end DirectX-udgaven. Det viste sig dog også i denne test, at skærmindstillingerne ikke havde nogen indflydelse på testresultaterne, hvilket ikke stemmer overens med vores forventninger. Forklaringen skal højst sandsynligt findes i, at den faktiske performance af en 3D-applikation kan afhænge af flere både hardwaremæssige og softwaremæssige faktorer, som vi også har illustreret i afsnit 9.5 "Testudstyr" side 23. Testsystemet med Geforce4-grafikkortet har højst sandsynligt en hardwarerelateret flaskehals, der gør, at det ikke har nogen indflydelse på grafikkortet, at der foretages flere pixel-beregninger ved at øge skærmopløsningen og farvedybden. Dvs. grafikkortets processor kommer aldrig til at arbejde maksimalt, men vil have perioder, hvor den ganske enkelt venter på at få overført data fra CPU'en, som den skal beregne. Hvis der er en flaskehals et sted i systemet før grafikkortet, vil grafikkortet aldrig blive udnyttet optimalt, og performance kommer ikke til at afhænge af grafikkortets arbejdsbyrde. Derfor

kan man forestille sig, at testsystemet med det hurtige GeForce4-grafikkort har en flaskehals i systemet før grafikkortet. Det er i denne forbindelse oplagt at pege på CPU'en eller AGP-porten som mulige flaskehalse i dette testsystem, da særligt CPU'en er langsom i forhold til grafikkortet. En flaskehals i et system er dog ikke unormalt, hvorfor dette testsystem stadigvæk er en udmærket repræsentant for et brugersystem. Derfor er resultaterne i testene med dette system gyldige i forhold til at repræsentere performanceforholdet i et system med en flaskehals, der ligger forud for grafikkortet.

I testen med Intel-grafikkortet var resultatet for sammenligning mellem Java3D med DirectX og OpenGL ikke ligeså markant som ved testen med GeForce4-grafikkortet. Samlet set performer OpenGL-udgaven 4,6% og 5,8% bedre, i henholdsvis JVM-version 1.4.2_06 og JVM-version 1.5.0_03, end DirectX-udgaven i dette testsystem. Dog er dette resultat ikke gennemgående for alle skærmindstillinger. Tendensen er, at OpenGL-udgaven performer bedre end DirectX-udgaven i de høje skærmindstillinger, mens DirectX-udgaven performer minimalt bedre end OpenGL-udgaven i de lave skærmindstillinger. Grafens tendens er, at performanceforskellen bliver større jo højere skærmindstillingen er, hvilket giver grundlag for at forestille sig, at performanceforskellen ville blive endnu større, hvis vi havde testet med endnu højere skærmindstillinger – eksempelvis 1600×1200 pixels med en farvedybde på 32-bit. De højere skærmindstillinger fra testen er de mest anvendte nu til dags, hvorfor resultaterne fra disse test vægtes særligt højt. Det er interessant, at performanceforskellen bliver større, når antallet af pixelberegninger øges. Dette indikerer, at OpenGL-udgaven er mere effektiv end DirectX-udgaven, når det kommer til de helt basale pixelberegninger. Dermed er der yderligere et grundlag for at sige, at OpenGL-udgaven generelt set renderer hurtigere end DirectX-udgaven.

Yderligere fremgår det af Intel-testen, at performance sænkes, når skærmindstillingerne højnes. Dette stemmer overens med vores forventninger om, at flere pixeldata sætter grafikkortet under større pres og deraf øger renderingstiden pr. frame. Dette faktum indikerer også, at der ikke er en flaskehals i systemet forud for grafikkortet, som gør, at grafikkortet ikke modtager data nok til, at det kan yde maksimalt under testen.

Vores kriterium for, at resultaterne er signifikante, var, at performanceforskellen mellem OpenGL-udgaven og DirectX-udgaven skulle være større end den højeste afvigelse i resultatet mellem to

identiske test, hvilken er 1,7%. Dermed er vores resultat omkring performanceforskellen i denne sammenhæng ikke blot en del af afvigelsen, men et signifikant udslag af, at der anvendes forskellige Java3D-versioner – DirectX og OpenGL.

Performanceforskel mellem JVM-versionerne 1.4.2_06 og 1.5.0_03

Generelt set var der ikke den store forskel i performance mellem JVM-versionerne 1.4.2_06 og 1.5.0_03. I testen med GeForce4-grafikkortet var der ingen signifikant forskel i performance mellem JVM-versionerne 1.4.2_06 og 1.5.0_03. Dette kan skyldes den føromtalte flaskehals, hvor systemet, ganske enkelt ikke er i stand til at levere data til grafikkortet i en tilstrækkelig hastighed, hvorfor performance ikke ændrer sig.

I Intel-testen var forskellen i performance 1,4% i testene med DirectX, mens performanceforskellen var 2,6% i testene med OpenGL – begge resultater i JVM-version 1.5.0_03's favør. Da disse procentsatser er tydeligt højere end den højeste testafvigelse, er der belæg for at sige, at JVM-version 1.5.0_03 performer bedre end JVM-version 1.4.2_06 i denne test.

Dermed må vi konstatere, at der ikke afsløres en klar forbedring i performance i den nye udgave af JVM, selv om JVM-version 1.5.0_03 i Intel-testen generelt performer signifikant bedre end 1.4.2_06. Dette modsvarer umiddelbart vores forventning til testen, hvor vi netop forventede en større performancemæssig forskel.

I forbindelse med vores undersøgelse af JVM-versionernes performance er det vigtigt at sige, at vores test ikke er en typisk test af en JVM's hastighed. For at give et fyldestgørende billede af en JVM's performance er det relevant at teste denne med mange forskellige instruktioner og algoritmer – eksempelvis sorteringsalgoritmer. Vores test afslører udelukkende, hvordan JVM performer i forbindelse med 3D-rendering, da vi ønsker at vurdere JVM i forhold til 3D-spilafvikling. Dog må vi her konstatere, at vores test ikke er en god repræsentant for et computerspil, da et typisk computerspil vil have mange beregninger og funktionskald, som ikke relaterer sig til grafisk rendering. Eksempelvis vil et computerspil typisk foretage en masse fysikberegninger. Det er derfor muligt, at et rigtigt computerspil, udviklet i Java, ville kunne afsløre et anderledes performanceforhold mellem JVM-version 1.5.0_03 og JVM-version 1.4.2_06.

Selve JVM'ernes opstartstider undersøger vi ikke. En forbedring i opstartstiden ville også være at betragte som en forbedring af JVM'ens performance. Dette aspekt af performance er dog ikke afgørende i computerspilsammenhæng, da en computerspilbruger naturligvis hellere vil leve med langsom opstart frem for generel langsom programafvikling.

11.2 Java som spiludviklingssprog

Dette afsnit omhandler, hvad vi, på baggrund af vores undersøgelsesresultater, kan føje til overvejelserne omkring Java som spiludviklingssprog.

Vores undersøgelse tager som nævnt udgangspunkt i Jacob Marners undersøgelse af Java som spiludviklingssprog. Jacob Marner nævner i sin rapport, at Java muligvis har en fremtid indenfor spiludvikling, da det er forventeligt, at JVM'en bliver yderligere forbedret i fremtiden, og at Java på denne baggrund muligvis endda vil blive hurtigere end C++ [Marner 2002, 37 og 38]. Vores test af JVM-version 1.5.0_03 sammenlignet med 1.4.2_06 viste dog, at JVM-teknologien ikke er blevet betydeligt bedre mellem disse to versioner. Denne påstand skal naturligvis tages med de forbehold, som vi har præsenteret i forrige afsnit. Ud fra vores undersøgelse kan vi dermed tilføje til diskussionen af Java som spiludviklingssprog, at performance ikke er forbedret i en grad, som skal få spiludviklere til at revidere deres syn på Java. Hvis performance af JVM løbende kan forbedres, vil Java muligvis nå en performance, som gør, at sproget kan konkurrere med C++ indenfor spiludvikling. Vi kan dog ikke spå om fremtiden for JVM.

Et argument, som taler for Java som spiludviklingssprog, er som nævnt, at produktiviteten i Java er højere end i C++. I Jacob Marners rapport kan man læse, at udviklingen for programmeringssprog indenfor spiludvikling er gået mod et højere abstractionsniveau. Spiludvikling har foregået i assembler, hvor spiludviklerne ikke kunne lide ideen om C, da de mente, at det var for langsomt. Først i 1993, da computerspillet "Doom" udkom, gik det op for udviklere, at det sagtens kunne lade sig gøre at udvikle spil i C, og derved mindske produktionstiden betydelig [Marner 2002, 75]. Man kunne således forestille sig, at udviklingen ville fortsætte i denne retning, og at Java eller et lignende sprog med lige så højt abstraktionsniveau ville kunne gøre sin entré i spiludviklingsbranchen.

I forhold til valg af Java3D-udgave kan vi sige, at OpenGL-versionen er at foretrække. Med Jacob Marners resultater i baghovedet[1], er det svært at sige om Java3D kan anvendes til spiludvikling. Måske er Java3D's forsøg på at være et generelt API over DirectX og OpenGL med et meget højt abstractionsniveau et problem i forhold til at opnå en høj performance. Vores resultater fortæller dog ikke noget om Java3D i forhold til andre måder at rendere 3D på, vi har undersøgt hvilken af Java3D's udgaver, der performer bedst.

[1] Jacob Marner når, som nævnt i afsnit 2.1 "Forundersøgelse" side 4, frem til, at Java3D med OpenGL er 2,5 gange langsommere end C++ med OpenGL.

12 Konklusion

Vores første mål med projektet var at afdække hvilken af Java3D API'ets implementeringer DirectX og OpenGL, der renderer hurtigst, ud fra de testsystemer vi har arbejdet med i rapporten. Ud fra resultaterne fra de to testsystemer kan det konkluderes, at OpenGL samlet set har været den implementering, der har renderet hurtigst i vores benchmarkprogram. I testen med Intel-grafikkortet performede OpenGL henholdsvis, med JVM 1.4.2_06 og 1.5.0_03, 4,6% og 5,8% bedre end ved brug af DirectX. Det mest bemærkelsesværdige resultat var, at OpenGL i skærmindstillingen (1280×1024×32) performede hele 19,5% bedre end DirectX, begge med JVM 1.4.2_06.

I testen med GeForce4-grafikkortet viste det sig, at OpenGL også her renderede hurtigere end DirectX. Den samlede forskel i performance var endda endnu tydeligere end ved testen med Intel-grafikkortet. I GeForce4-testen var OpenGL, med JVM 1.4.2_06, 22,2% hurtigere til at rendere en frame og med JVM 1.5.0_03, 23,6% hurtigere.

Samlet kan vi konkludere, at OpenGL-udgaven af Java3D performer bedre end DirectX-udgaven i vores benchmarkprogram.

Med henblik på undersøgelsen af performanceforbedringer ved rendering af 3D-grafik ved at anvende Javas virtuelle maskine 1.5.0_03 frem for 1.4.2_06, viser resultaterne, fra begge testsystemer, at der kan opnås en mindre forbedring i performance. I testen med Intel-grafikkortet performer JVM-versionen 1.5.0_03 2,6% bedre end version 1.4.2_06 når OpenGL-udgaven benyttes. Med DirectX performer JVM-versionen 1.5.0_03 1,4% bedre end version 1.4.2_06.

I testen med GeForce4-grafikkortet er performanceforskellen næsten ikke eksisterende. Den samlede performanceforskel ved brug af OpenGL er 0,03% som JVM-version 1.5.0_03 performer bedre end JVM-version 1.4.2_06. Derimod viser resultaterne at JVM 1.4.2_06 rent faktisk performer bedre ved brug af DirectX. Det samlede resultat blev udregnet til 0,58%. Disse resultater afskrives dog med henvisning til den generelle afvigelse, som er at finde i resultaterne.

Samlet kan der konkluderes, at der er en mindre performanceforbedring ved at benytte JVM-version 1.5.0_03 frem for JVM-version 1.4.2_06.

Set i lyset af Java som spiludviklingssprog til performancekrævende spil, er der dermed ikke den store forbedring at spore i vores benchmarktest, hvorfor det på denne baggrund endnu ikke er relevant at revidere opfattelsen af Java som spiludviklingssprog. Vi kan dog konkludere, at det bedst betaler sig, at anvende OpenGL-udgaven af Java3D, da denne performer bedre end DirectX-udgaven.

13 Litteraturliste

[Brackeen 2003] Brackeen, David (2003) *"Developing Games in Java"* Copyright 2004 by New Riders Publishing.

[Marner 2002] Marner, Jacob (2002) *"Evaluating Java For Game Development"* Rapport by Jacob Marner, B.Sc. Department of Computer Science, University of Copenhagen, March 4th 2002. http://rolemaker.dk/articles/evaljava/index.htm

[Pers. kom. Marner 2005] Personlig samtale med Jacob Marner 2005.

14 Referenceliste

[Ref. 01] Om DirectX fra Microsofts hjemmeside:
http://www.microsoft.com/windows/directx/default.aspx?url=/windows/directx/productinfo/overview/default.htm

[Ref. 02] Windows XP's egen beskrivelse af DirectX i Hjælp og support.

[Ref. 03] Om OpenGL fra den officielle hjemmeside:
http://www.opengl.org/about/overview.html#1

[Ref. 04] Officiel PDF om specifikationen af OpenGL version 2.0:
http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[Ref. 05] Om OpenGL fra Microsofts hjemmeside:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/opengl/openglstart_9uw5.asp

[Ref. 06] Deadline Games' hjemmeside:
http://www.deadlinegames.com/

[Ref. 07] Information om Jacob Marner:
http://www.rolemaker.dk

[Ref. 08] Hjemmeside for "Evaluating Java For Game Development":
http://rolemaker.dk/articles/evaljava/index.htm

[Ref. 09] Java3D API Specifikation fra Sun microsystems:
http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dguide/Introduction.html

[Ref. 10] Javadesktop.org:
http://www.javadesktop.org/forums/thread.jspa?threadID=10338&tstart=90

[Ref. 11] Java3D Performance Guide:
http://java.sun.com/products/java-media/3D/collateral/1.2.1.perfguide.html

[Ref. 12] Om OpenGL fra Microsofts hjemmeside:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/opengl/oglport_3oqg.asp

[Ref. 13] Definition:
http://www.english-test.net/toeic/vocabulary/words/037/toeic-definitions.php

[Ref. 14] Java3D API Specifikation fra Sun microsystems:
http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dguide/Introduction.html#50910

[Ref. 15] Hjemmesiden mywiseowl.com om profilers:
http://www.mywiseowl.com/articles/Profiler

[Ref. 16] PDF om rendering fra http://www.architecture.yale.edu:
http://www.architecture.yale.edu/dmonline/tutorials/Maya/Complete/Rendering.pdf

15 Ordliste

3D	Når noget har eller virker som om det har bredde, højde og dybde. Forkortelse af tredimensionel.
3D-rendering	Processen i at producere et billede på skærmen ud fra polygoner, texture, lys, skygge og andre grafiske informationer.
Abstraktions-niveau	Abstraktionsniveauet i et programmeringssprog er sigende for, hvor mange instruktioner, der skal skrives for at opnå den ønskede funktion. Et højt abstraktionsniveau kræver få instruktioner, mens et lavt abstraktionsniveau kræver mange instruktioner.
API	Står for Application Programming Interface og er et standardiseret programmeringsinterface. Det muliggør, for udvikleren, at skrive applikationer til en standard uden specifik viden omkring hardwareimplementeringen.
Assembler	Et lav-niveau program, som oversætter instruktioner fra andre programmer til maskinkode, så de kan benyttes af den rette hardware.
Benchmark-program	Et program der er egnet til at måle performance af et givent system. Vores banchmarkprogram kaldes Java3DTest.
Bit	Er forkortelsen af binary digit, hvilket er den mindste enhed, der kan lagres i computeren.

Bug	I en computer er en bug en fejl, der f.eks. forhindrer et program i at fungere korrekt.
C/C++	C eller C++ er programmeringssprog med et relativt højt abstraktionsniveau dog lavere end Java. Disse sprog er meget anvendt til spiludvikling, da de generelt har høj performance.
Constuctor	En metode, der opretter et objekt. I Java har constructoren samme navn som dens klasse.
DirectX	Er API'er for bl.a. 3D-grafik, lyd, input og netværk fra Microsoft der indeholder programpakkerne; Direct3D, DirectSound, DirectDraw, DirectVideo, DirectPlay og DirectInput. DirectX er integreret i Microsoft Windows.
Frame	En frame er et enkelt grafisk billede i en række af grafiske billeder.
Framebuffer	Et område af RAM der benyttes til at lagre pixeldata for et enkelt skærmbillede, eller frame.
Fravedybde	Antal af farver per pixel, som skærm og grafikkort understøtter.
GL4Java	Et API til Java som giver adgang til OpenGL API'et. Kaldes også for en OpenGL-binding.
Java3D	Er et scenegrafsbaseret 3D API til Java-platformen, udviklet af Sun Microsystems. Det ligger oven på DirectX eller OpenGL.
JNI	Er forkortelsen for Java Native Interface og definerer vejen som et Java-program skal kalde programmer skrevet i sprog som C.
JVM	Står for Java virtuelle maskine. Er software implementation på computeren som kører kompileret Java-kode, såsom appletter og applikationer.
Multiprocessing	Refererer til et computersystems evne til at supportere mere end et program på samme tid.
Multithreading	En teknik der sørger for at der startes en tråd i stedet for en ny proces. Tråde bruger signifikant mindre hukommelse end processer.

OpenGL	Et grafik-API udviklet af Silicon Graphics.
Performer/ Performance	Et overordnet mål for produktiviteten af et givent system.
Pixel	Forkortelse af picture element. En pixel er det mindste element af et grafisk display eller det mindste element af et renderet billede.
Realtime	Når noget genereres mens det bruges, modsat f.eks. optagelser, der kan tage længere tid at generere end de tager at vise.
SDK	Står for Software development kit, og er en samling af værktøjer, API'er og dokumentation.
Texture	Er en imagefil, som bruges til at tilføre komplekse mønstre til et objekts overflade, i en 3D-scene.

16 Bilag

16.1 Interview med Jacob Marner – Deadline games

Hvilke programmeringssprog anvendes i firmaet til udvikling af de enkelte dele af 3D-spil?

- Hos Deadline games bruger de hovedsageligt C++ til udvikling af spil. Enkelte ting er programmeret i assembler. Dette pga., at der udvikles til PS2, hvilket gør objektorienteringen i C++ for dyr – rent hukommelses- og performancemæssigt.
-

Anvender I Java? – Hvorfor?

- Problemet med Java til konsolspil er at memoryforbruget er alt for højt i forhold til at PS2 og Xbox har 16MB memory.
- Et andet argument for ikke at bruge Java til konsoller er at konsollerne ikke har nogen virtuel hukommelse.
- Grunden til der ikke er så mange RAM i konsollerne er, at de på den måde bliver mere robuste.
- Next-generation-konsoller, PS3 og xbox2/360, har mere hukommelse, hvorfor Java måske ville kunne komme på tale, hvis der vel at mærke bliver udviklet en virtuel maskine til disse konsoller. Samtidig har de nye konsoller flere processorer, hvilket åbner op for parallel programmering. Dette gøres lettere i Java end i C++, hvorfor Java her har en stor fordel i forbindelse med produktiviteten.
- Java statisk kompileret kode er ikke lige så hurtig som C++ statisk kompileret kode, da Java ikke giver mulighed for at komme ned og ”pille” og optimere de enkelte elementer. Java har dog et højere abstraktionsniveau.

Benytter i jer af DirectX, OpenGL eller en ren sotfware implementering?

- Software er for langsomt, og det er ikke længere nødvendigt at tilbyde softwaremulighed, da alle gamere og selv low-end PC’ere er i besiddelse af grafikkort der understøtter både DirectX og OpenGL
- Deadline Games benytter sig af DirectX.
- Jacob Marner tror ikke vi vil finde den store performanceforskel mellem brugen af DirectX og OpenGL. Der er dog ikke lavet nogen egentlige undersøgelser om performanceforskellene. Om folk benytter DirectX eller OpenGL synes mere at være afhængigt af folks forudindtagede oplevelse af programmerne.

Hvordan Tester i på performance?

- Deadline games performancetester via frameprofiler, der laver nogle grafer over hvilke komponenter, der bruger mest tid og hvilket scriptsprog.
- Memoryprofil der holder øje med heapen.

- Man skal generelt altid performanceteste i millisekunder og ikke frames pr. sekund da man på den måde for en meget mere nøjagtig måling.
- Deadline games benytter sig også af performanceanalyse, og laver grafer over performance.
- Deadline games forsøger sig i øjeblikket med en form for automatisk test af programmerne i løbet af spiludviklingen.
- Det er vigtigt at man lader sit Javaprogram køre ca. 20-30 sekunder inden man tester, da Java bruger lidt længere tid første gang man kører koden igennem.

Diverse ting der kom frem under interviewet

- Java3D er altid lidt efter i forhold til de nyeste funktioner i DirextX og OpenGL.
- Interessant at undersøge Javas performanceforbedring fra JVM-version 1.3.1 til 1.5.0_01
- Med en virtuel maskine er det muligt at vurdere koden og dens brug i runtime. Derved kan man optimere kodekompileringen efter dette og opnå forbedret performance. Noget lignende er kommet med GCC3-kompileren, som er en C++-kompiler. Den gør det, at den tester programmet først og på baggrund af statistik fra denne test, kompileres koden med optimeringer. Derved har ikke blot Java denne fordel.

16.2 DirectX

Samlet udgør nedenstående komponenter DirectX [Ref. 02]:

- DirectDraw API'et understøtter direkte adgang til acceleratorfunktionerne i skærmkortet og giver f.eks. spil og digitale videocodecs adgang til funktionerne uden oplysninger fra brugeren om enhedens funktioner.

- Direct3D er et 3D-API på lav-niveau, der giver en grænseflade til de 3D-gengivelsesfunktioner, der er indbygget i moderne skærmkort og understøtter specialiserede CPU-instruktionssæt.

- DirectSound API'et kæder et program og lydkortets funktioner sammen til hurtig lydmixing, lydafspilning og lydindspilning.

- DirectMusic API'et er den interaktive lydkomponent i DirectX og understøtter såvel komprimerede som ikke-komprimerede digitale lydformater. Derved kan filmlyd reagere på forskellige ændringer i softwaremiljøet, fra f.eks. brugeren.

- DirectInput API'et giver avanceret input til bl.a. spil og behandler input fra f.eks. joysticks, tastaturer og mus og spilenheder med feedback.

- DirectPlay API'et understøtter spilforbindelser over et modem, Internettet eller LAN. I nyere versioner af DirectPlay er der understøttelse af tale via netværket.

- DirectShow API'et indeholder ind- og afspilning af multimediefiler på computeren og på internetservere, såsom DVD-afspilning, videoredigering og -mixing. DirectShow understøtter en lang række lyd- og videoformater, såsom AVI (Audio-Video Interleaved), og MP3 (MPEG Audio Layer-3).

17 Kildekode

```
1    import java.awt. *;
2    import com.sun.j3d.utils.universe.  *;
3    import javax.media.j3d. *;
4    import javax.vecmath. *;
5    import javax.swing. JFrame ;
6    import com.sun.j3d.utils.geometry.  *;
7    import com.sun.j3d.utils.image.TextureLoader;
8    import java.io. *;
9
10   /**
11   * The application renders box objects using Java3D with DirectX or OpenGL
12   */
13   public class Java3DTest
14   {
15       private static final DisplayMode  POSSIBLE_MODES [] = {
16           new DisplayMode (1600, 1200, 32, 60),    //0
17           new DisplayMode (1600, 1200, 16, 60),    //1
18           new DisplayMode (1280, 1024, 32, 60),    //2
19           new DisplayMode (1280, 1024, 16, 60),    //3
20           new DisplayMode (1024, 768, 32, 60),     //4
21           new DisplayMode (1024, 768, 16, 60),     //5
22           new DisplayMode (800, 600, 32, 60),      //6
23           new DisplayMode (800, 600, 16, 60),      //7
24           new DisplayMode (640, 480, 32, 60),      //8
25           new DisplayMode (640, 480, 16, 60)       //9
26       };
27
28       /**
29       * Constructs a new instance of Java3DTest, sets up full screen mode and
30       * creates the virtual universe
31       */
32       public Java3DTest ()
33       {
34           //set up full screen mode
35           screenManager   = new ScreenManager ();
36           displayMode   = POSSIBLE_MODES [displayModeNr ];
37           screenManager.  setFullScreen (displayMode );
38
39           //load texture
```

```
40            java.net. URL  texImage  = null;
41            try {
42                texImage   = new java.net. URL("file:stone.jpg" );
43            }
44            catch (java.net. MalformedURLException  ex) {
45                System .out. println (ex. getMessage ());
46            System .exit (1);
47            }
48            texture  = new TextureLoader (texImage, screenManager.  getFullScreenWindow ()).getTexture ();
49
50            //get the GraphicsConfiguration to be used with the Canvas3D
51            GraphicsConfiguration   config  = SimpleUniverse. getPreferredConfiguration  ();
52
53            //create Canvas3D
54            TestCanvas3D canvas3D   = new TestCanvas3D (config, testTime, warmupTime,   this );
55            canvas3D. setDoubleBufferEnable (true );
56
57            //add Canvas3D to the full screen window
58            screenManager.  getFullScreenWindow ().getContentPane ().setLayout (new BorderLayout ());
59            screenManager.  getFullScreenWindow ().getContentPane ().add("Center" , canvas3D );
60            screenManager.  getFullScreenWindow ().setVisible (true );
61
62            createAppearance ();
63
64            // Create scene and virtual universe
65            BranchGroup scene   = createSceneGraph ();
66            createUniverse (canvas3D, scene );
67        }
68
69        /**
70        * Creates the scene graph, and adds 5000 boxes
71        */
72        public  BranchGroup  createSceneGraph ()
73        {
74            // Create the root of the branch graph
75            BranchGroup objectBranch   = new BranchGroup ();
76
77            // Create the upper boxes
78            for(int i=0; i<50; i++)
```

```
79              {
80                  for(int j=50; j>0; j--)
81                  {
82                     objectBranch.  addChild(newBox(new Vector3d(-25+i, 0.5, 25-j)));
83                  }
84              }
85
86              // Create the lower boxes
87              for(int i=0; i<50; i++)
88              {
89                  for(int j=50; j>0; j--)
90                  {
91                     objectBranch.  addChild(newBox(new Vector3d(-25+i*2, -0.5, 25-j*2)));
92                  }
93              }
94
95              // Have Java 3D perform optimizations on this scene graph.
96              objectBranch. compile();
97
98              return objectBranch;
99          }
100
101         /**
102         * Creates the universe and sets up the view. Adds fog, directional light
103         * and ambient light
104         */
105         public void createUniverse(TestCanvas3D canvas3D, BranchGroup scene   )
106         {
107             VirtualUniverse universe   = new VirtualUniverse();
108
109             Locale locale = new Locale(universe);
110
111             // Create viewing branch
112             BranchGroup viewBranch   = new BranchGroup();
113             viewTransformGroup   = new TransformGroup();
114             viewTransformGroup.  setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE   );
115             viewBranch. addChild(viewTransformGroup );
116
117             ViewPlatform viewPlatform   = new ViewPlatform();
```

```
118             viewTransformGroup.  addChild (viewPlatform );
119
120              View view  = new  View();
121             view. attachViewPlatform (viewPlatform );
122             view. addCanvas3D (canvas3D );
123
124             PhysicalBody physicalBody    = new  PhysicalBody ();
125             view. setPhysicalBody (physicalBody );
126
127             PhysicalEnvironment physicalEnvironment     = new  PhysicalEnvironment ();
128             view. setPhysicalEnvironment  (physicalEnvironment );
129
130              // Set viewing parameters to ensure correct view fustrum
131              double  frustumfar  = 40.0;
132             view. setFieldOfView (Math.PI  / 2.0);
133             view. setFrontClipPolicy (View.VIRTUAL_EYE );
134             view. setBackClipPolicy (View.VIRTUAL_EYE );
135             view. setFrontClipDistance  (0.1);
136             view. setBackClipDistance  (frustumfar );
137             view. setScreenScalePolicy (View.SCALE_EXPLICIT );
138             view. setScreenScale (1.0);
139
140
141             Transform3D t   = new  Transform3D ();
142             t. set(1, new  Vector3d(20,  0,  0));
143             viewTransformGroup.  setTransform (t);
144
145              //Boundings Sphere for the universe
146             BoundingSphere worldSphere    = new  BoundingSphere (new  Point3d(0.0,0.0,0.0),  1000.0);
147
148              // Create rotator for rotating the view through the universe
149             Alpha rotationAlpha    = new  Alpha(-1,  100000 );
150             RotationInterpolator rotator    = new  RotationInterpolator (rotationAlpha, viewTransformGroup, t,
      (float )Math.PI*2.0f,  0.0f);
151             BoundingSphere bounds    = worldSphere;
152             rotator. setSchedulingBounds (bounds );
153              //add the rotator to the viewTransformGroup
154             viewTransformGroup.  addChild (rotator );
155
```

```
156             // Add fog
157             BranchGroup environmentBranch    = new BranchGroup ();
158             LinearFog fog   = new LinearFog (new Color3f (0.0f, 0.0f, 1.0f), 0.0, frustumfar );
159             fog. setInfluencingBounds (worldSphere );
160             environmentBranch.  addChild (fog);
161
162             // Add strong global directional light
163             DirectionalLight light    = new DirectionalLight ();
164             light. setDirection (1.0f, 0.0f, 1.0f);
165             light. setColor (new Color3f (1.0f, 1.0f, 1.0f));
166             light. setEnable (true );
167             light. setInfluencingBounds (
168               new BoundingSphere (worldSphere ));
169             environmentBranch.  addChild (light );
170
171             // Add vague global ambient light
172             AmbientLight alight    = new AmbientLight (true, new Color3f (0.4f, 0.4f, 0.4f));
173             alight. setInfluencingBounds (worldSphere );
174             environmentBranch.  addChild (alight );
175
176             // Have Java 3D perform optimizations on the environment branch
177             environmentBranch.  compile ();
178
179             // Add the environment branch
180             locale. addBranchGraph (environmentBranch );
181
182             // Add the scene graf branch
183             // (the scene graf is already compiled)
184             locale. addBranchGraph (scene );
185
186             // Have Java 3D perform optimizations on the view branch
187             viewBranch. compile ();
188
189             // Add the view branch
190             locale. addBranchGraph (viewBranch );
191         }
192
193         /**
194         * Creates a new box object with a transformgroup using the default
```

```
195         * appearance and texture
196         */
197         public  TransformGroup  newBox(Vector3d pos )
198         {
199             // Create the TransformGroup node
200             TransformGroup objTrans   = new TransformGroup ();
201 
202             // Create textured cube and add it to the scene graph.
203             Box textureCube  = new Box(0.2f, 0.2f, 0.2f,
204                         Primitive.GENERATE_TEXTURE_COORDS     | Primitive.GENERATE_NORMALS, app  );
205             objTrans. addChild(textureCube );
206 
207             Transform3D t   = new Transform3D ();
208             t. set(pos);
209 
210             objTrans. setTransform (t);
211 
212             return objTrans;
213         }
214 
215         /**
216         * Creates a common, shared appearance for all boxes
217         */
218         public  void createAppearance ()
219         {
220             // Create appearance object for textured cube
221             app  = new Appearance ();
222 
223             // Create material object for appearance
224             Material material   = new Material(
225                 new Color3f(1.0f,1.0f,1.0f), // Ambient
226                 new Color3f(0.0f,0.0f,0.0f), // Emissive
227                 new Color3f(1.0f,1.0f,1.0f), // Diffuse
228                 new Color3f(0.5f,0.5f,0.5f), // Specular
229                 128.0f);
230 
231             //set this flag to make sure that the box will respond to lights
232             material. setLightingEnable (true);
233 
```

```
234             app. setMaterial (material );
235
236              //create TextureAttributes
237             TextureAttributes texattrib   = new TextureAttributes ();
238
239             texattrib. setPerspectiveCorrectionMode  (TextureAttributes.NICEST  );
240             texattrib. setTextureMode (TextureAttributes.MODULATE  );
241
242             app. setTextureAttributes (texattrib );
243
244              //set the texture of the appearance
245             app. setTexture (texture );
246         }
247
248         /**
249         * Prints final test data to file
250         */
251         public  void  finish (long  testTimeUsed,  int  testFrames )
252         {
253              //write test data to file
254              try {
255                  PrintWriter  out  = new  PrintWriter (new  BufferedWriter (
256                      new
      FileWriter (path+"3Dtest_disp" +displayModeNr +"_testTime" +testTime +"_warmupTime" +warmupTime +".txt" )));
257                  out. println ("DisplayMode: " +displayMode. getWidth ()+" * "+displayMode. getHeight ()+" *
      "+displayMode.getBitDepth());
258                  out.println("  Frames:  "+testFrames);
259                  out. println ("WarmupTime: " +warmupTime );
260                  out. println ("Time used (ms): " +testTimeUsed );
261                  out. println ();
262                  out. println ("*** MS PR. FRAME: " +(float )testTimeUsed /testFrames +" ***" );
263                  out. println ();
264                  out. println ("Frames pr. second: " +((float )testFrames /testTimeUsed )*1000 );
265
266                  out. flush ();
267                  out. close ();
268              } catch (IOException  e){
269                  System .out. println ("Exception: " +e.getMessage ());
270              }
```

```
271
272             //exit
273             System.exit(0);
274         }
275
276         /**
277         * Prints a short description of the four arguments and how to use the application
278         */
279         public static void printUse()
280         {
281             System.out.println("Use: java Java3DTest numFrames(int) displayModeNr(int) warmupTime(int)
    path(folder)");
282
283             System.exit(0);
284         }
285
286         /**
287         * Main checks arguments and creates a new Java3DTest
288         */
289         public static void main(String[] args)
290         {
291             if(args.length == 4)
292             {
293                 testTime  = Integer.parseInt(args[0]);
294                 displayModeNr  = Integer.parseInt(args[1]);
295                 warmupTime  = Integer.parseInt(args[2]);
296                 path  = args[3];
297             }
298             else if(args.length == 0)
299             {
300                 testTime  = 3000;
301                 displayModeNr  = 0;
302                 warmupTime  = 30000;
303                 path  = "Test";
304             }
305             else
306             {
307                 printUse();
308             }
```

```
309             Java3DTest java3DTest   = new Java3DTest ();
310         }
311
312         private Texture texture;
313         private static int displayModeNr;
314         private static long warmupTime, testTime;
315         private DisplayMode displayMode;
316         private TransformGroup viewTransformGroup;
317         private ScreenManager screenManager;
318         private Appearance app;
319         private static String path;
320     }
321
322
```

```
1    import javax.media.j3d.*;
2    import java.awt.*;
3    import javax.vecmath.*;
4
5    /**
6    * The TestCanvas3D calculates and displays test data (fps, ms pr. frame etc.)
7    */
8    public class TestCanvas3D extends Canvas3D
9    {
10       /**
11       * Constructs and initializes a new Canvas3D object that Java 3D can render onto.
12       */
13       public TestCanvas3D(GraphicsConfiguration graphicsConfiguration, long testTime,
14           long warmupTime, Java3DTest java3DTest )
15       {
16           super(graphicsConfiguration);
17           this.testTime = testTime;
18           this.warmupTime = warmupTime;
19           this.java3DTest = java3DTest;
20           frame = 0;
21           fpsI = 0;
22           testStarted = false;
23       }
24
25       /**
26       * This routine is called by the Java 3D rendering loop after clearing the
27       * canvas and before any rendering has been done for this frame. We call this
28       * method to be able to calculate the amount to time used to render the frames
29       */
30       public void preRender()
31       {
32           if(frame == 0)
33               startTime = System.currentTimeMillis();
34
35           currentTime = System.currentTimeMillis();
36
37           //calculate time in milliseconds pr. frame
38           if(frame != 0)
39               elapsedTime = currentTime - prevTime;
```

```
40            prevTime  = currentTime;
41
42            //if the test is started add the elapsed time to the testTimeUsed
43            if(testStarted )
44                testTimeUsed  += elapsedTime;
45
46            if(testStarted  && testTimeUsed  >= testTime )
47                finish();
48
49            //start test if warupTime is passed
50            if(!testStarted  && currentTime  - startTime  >= warmupTime )
51            {
52                testFrames  = 0;
53                testStartTime  = System.currentTimeMillis ();
54                testTimeUsed  = 0;
55                testStarted  = true;
56            }
57
58            //calculate stats if test is not started
59            if(!testStarted )
60                calcStats();
61
62            super.preRender();
63        }
64
65        /**
66        * This routine is called by the Java 3D rendering loop after completing all
67        * rendering to the canvas, and all other canvases associated with this view,
68        * for this frame following the buffer swap. Therefore we display the test data
69        * at this time
70        */
71        public void postSwap()
72        {
73            //display stats if test is not started
74            if(!testStarted )
75                displayStats();
76
77            super.postSwap();
78        }
```

```
79
80          /**
81          * This routine is called by the Java 3D rendering loop after completing all
82          * rendering to the canvas for this frame and before the buffer swap. After
83          * each frame we increment the frame attr. and the testFrames attr. if the
84          * test is started
85          */
86          public void postRender()
87          {
88              super.postRender();
89
90              frame++;
91              if(testStarted)
92                  testFrames++;
93          }
94
95          /**
96          * Called when test is finished and calculates testTimeUsed before calling
97          * finish(long testTimeUsed, int testFrames) in Java3DTest class
98          */
99          public void finish()
100         {
101             testEndTime  = System.currentTimeMillis();
102             testTimeUsed  = testEndTime - testStartTime;
103             java3DTest.finish(testTimeUsed, testFrames);
104         }
105
106         /**
107         * Is only used before test starts and calculates data to display on the
108         * screen. In order to get usefull information it waits 10 frames before
109         * calculating an average
110         */
111         public void calcStats()
112         {
113             //calculate fps
114             //fps = (float)(1000.0f/elapsedTime);
115             fpsI++;
116             fpsTime += elapsedTime;
117             if(fpsI==10)
```

```
118             {
119                 if(fpsTime  != 0)
120                     fps   = (float )(10000 .0f/fpsTime );
121                 fpsI   = 0;
122                 fpsTime   = 0;
123             }
124 
125             //calculate total time
126             totalTime   = (currentTime  - startTime )/1000;
127         }
128 
129         /**
130         * Prints statistics FPS, ms pr. frame and total time out on the screen
131         * before the test starts
132         */
133         public  void  displayStats ()
134         {
135             Graphics  g  = getGraphics ();
136             g. setColor (Color .WHITE );
137             //fps
138             g. drawString ("FPS: " +fps,  getWidth () - 200, getHeight () - 100);
139             //ms pr. frame
140             g. drawString ("ms pr. frame: " +elapsedTime,  getWidth () - 200, getHeight () - 75);
141             //total time
142             g. drawString ("Total time: " +totalTime,  getWidth () - 200, getHeight () - 50);
143         }
144 
145         private  long currentTime, prevTime, elapsedTime, fpsTime, startTime,
146             testStartTime, totalTime, testEndTime, testTimeUsed, warmupTime, testTime;
147         private  float  fps;
148         private  int fpsI, frame, testFrames;
149         private  boolean  testStarted;
150         private  Java3DTest java3DTest;
151     }
```

ScreenManager.java

```
1    import java.awt.*;
2    import javax.swing.JFrame;
3
4    /**
5    * The ScreenManager class manages initializing
6    * full screen graphics modes.
7    */
8    public class ScreenManager
9    {
10       /**
11       * Creates a new ScreenManager object.
12       */
13       public ScreenManager()
14       {
15           GraphicsEnvironment environment =
16               GraphicsEnvironment.getLocalGraphicsEnvironment();
17           device = environment.getDefaultScreenDevice();
18       }
19
20       /**
21       * Enters full screen mode and changes the display mode.
22       * If the specified display mode is null or not compatible
23       * with this device, or if the display mode cannot be
24       * changed on this system, the application exits.
25       */
26       public void setFullScreen(DisplayMode displayMode)
27       {
28           final JFrame frame = new JFrame();
29           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30           frame.setUndecorated(true);
31
32           device.setFullScreenWindow(frame);
33
34           if (displayMode != null && device.isDisplayChangeSupported())
35           {
36               try {
37                   device.setDisplayMode(displayMode);
38               }
39               catch (IllegalArgumentException ex) {
```

```
40                      System.out.println(ex);
41                      System.exit(0);
42                  }
43                  // fix for Java3D
44                  frame. setSize(displayMode. getWidth(), displayMode. getHeight());
45              }
46              else
47              {
48                  System.out.println("Display change not supported"  );
49                  System.exit(0);
50              }
51          }
52
53          /**
54          * Returns the window currently used in full screen mode.
55          * Returns null if the device is not in full screen mode.
56          */
57          public JFrame getFullScreenWindow()
58          {
59              return (JFrame)device.getFullScreenWindow();
60          }
61
62          private GraphicsDevice device;
63      }
```