Faculteit Toegepaste Wetenschappen
Vakgroep ELIS
Voorzitter: Prof. dr. ir. J. Van Campenhout

Ontwikkeling van een virtuele robot in Java3D

door Dieter Van Uytvanck

Promotor: Prof. dr. ir. K. De Bosschere
Thesisbegeleiders: dr. ir. M. Ronsse, ir. M. Christiaens

Afstudeerwerk ingediend tot het behalen van de graad van licentiaat in de informatica, optie: softwareontwikkeling

Academiejaar 2001-2002

Toelating

De auteur geeft de toelating dit afstudeerwerk voor consultatie beschikbaar te stellen en delen van het afstudeerwerk te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit dit afstudeerwerk.

Dieter Van Uytvanck 28 mei 2002

Dankwoord

Graag zou ik iedereen willen bedanken die heeft bijgedragen tot de verwezenlijking van dit eindwerk, in het bijzonder dank ik:

- mijn promotor, Prof. dr. ir. K. De Bosschere, voor het geven van nuttige wenken tijdens de vergaderingen en bij het schrijven van dit document;
- mijn begeleiders, dr. ir. M. Ronsse en ir. M. Christiaens, voor de bijstand bij de implementatie van de virtuele robot en voor de hulp bij het oplossen van de praktische problemen die daarbij voorkwamen;
- de vrije softwaregemeenschap, voor het leveren van het grootste deel van de software die bij het realiseren van deze thesis gebruikt werd;
- mijn ouders, zonder wiens steun ik onmogelijk deze studies gevolgd zou kunnen hebben.

Ontwikkeling van een virtuele robot in Java3D

door
Dieter Van Uytvanck

Afstudeerwerk ingediend tot het behalen van de graad van licentiaat in de informatica, optie: softwareontwikkeling

Academiejaar 2001-2002

Universiteit Gent
Faculteit Toegepaste Wetenschappen
Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout

Promotor: Prof. dr. ir. K. De Bosschere
Begeleiders: ir. M. Christiaens, dr. ir. M. Ronsse

Samenvatting
Het doel van deze scriptie bestaat erin een virtuele robot te creëren die verbonden is met een Java-aangestuurde Scorbot ER III. Met dit driedimensionaal model is het mogelijk om de bewegingen van de echte robot te volgen en hem op een intuïtieve manier aan te sturen. Aangezien de communicatie tussen beiden gebeurt via het TCP/IP-protocol is het mogelijk om de robot op afstand te besturen en te controleren.

Trefwoorden: 3D-simulatie, virtuele robot, robotbesturing, Java3D

Inhoudsopgave

1 Probleemstelling **1**

1.1 Situatieschets 1

1.2 De beschikbare technologie: een overzicht 2

1.3 Vereisten voor de virtuele robot 6

2 Platformkeuze **8**

2.1 Noden 8

2.2 VRML 10

2.3 Java3D 12

2.4 Conclusie 16

3 Implementatie **17**

3.1 Anatomie van de robot 17

3.2 Beschrijving in VRML 18

3.3 De opbouw in Java3D 19

3.4 De camerapositionering 25

3.5 Opbouw klassestructuur 28

3.6 Constructie van de interface 31

3.7 Prestatiemeting 33

4 Interactie met de simulator **35**

4.1 Druktoetsen 35

4.2 Slepen-en-plaatsen 35

4.3 Aanraakscherm 37

4.4 Joystick 37

5 Simulator-robot verbinding **43**

5.1 Doel van de communicatie 43

5.2 Keuze van communicatieprogrammatuur 43

5.3 Concrete implementatie 44

5.4 Bedenkingen bij vertraging 47

5.5 Bedenkingen bij beveiliging 47

6 Toekomstperspectieven **48**
6.1 Botsingen 48
6.2 Beeldverwerking 50

A Broncode op CD-ROM **52**

B Metingen van processorgebruik **54**

Lijst van figuren

1.1 De Scorbot ER III 2
1.2 Een 2D-model 3
1.3 Een draadmodel 4
1.4 Een volwaardig 3D-model 4

2.1 mogelijke lagen bij het renderen 9
2.2 VRML codefragment 10
2.3 Het resultaat van de code uit fig. 2.2 en de bijhorende DAG 11
2.4 Scènegraaf en resultaat van de code uit figuur 2.5 14
2.5 Codefragment van een programma in Java3D 15

3.1 De verschillende lichaamsdelen van de robot 18
3.2 De hoeken tussen de verschillende beweegbare onderdelen 19
3.3 De opbouw van de inhoud-scènegraaf van de virtuele robot 21
3.4 Overzicht van rotatiemethodes 24
3.5 De ViewBranchGraph in detail uitgewerkt 25
3.6 De camera cirkelt rondom de robot heen 27
3.7 Schermafdruk van de GUI bij de virtuele robot 32
3.8 Grafiek voor processorbelasting bij de torens van Hanoi 34

4.1 De Behavior-klasse voorgesteld als deterministische automaat 36
4.2 Scènegraaf met gedragingen voor slepen-en-plaatsen 38
4.3 Schematische doorstroming van gegevens bij invoer via de joystick 41

5.1 Beginpositie van de robot 45
5.2 Uitwisseling van gewrichtscoördinaten via sockets 45

6.1 Botsingsdetectie bij een bewegende kubus 49
6.2 Perceptiecyclus van een visueel gestuurde robot 51

Lijst van tabellen

3.1 Rotatie-assen van de robotonderdelen 22

4.1 Invoerverwerking van de joystick 42

B.1 Metingen van processorgebruik met en zonder optimalisaties 55

Hoofdstuk 1

Probleemstelling

1.1 Situatieschets

Binnen de vakgroep ELIS werd vorig jaar het idee opgevat om een modern bewakingssysteem te bouwen. Onder de noemer "Project Cycloop" zouden de daarvoor bedoelde robot en camera volledig aangestuurd worden in Java. Na de nodige aanpassingen is het dus de bedoeling om met dit project aan te tonen dat, in tegenstelling tot wat algemeen aangenomen wordt, het wel degelijk mogelijk is om een applicatie die waretijdsvereisten stelt te implementeren in een taal als Java.

1.1.1 De robot

Als ik het verder over "de robot" zal hebben, gaat het over de Scorbot ER III, een robot met 5 vrijheidsgraden en een grijper die open en dicht kan gaan. Een foto ervan is te vinden in figuur 1.1. De basis voor veel van de onderdelen van dit bewakingssysteem werd reeds gelegd in vorige jaren. Zo werd de robot tot voor kort aangestuurd door 3 i8086-processoren en gebeurde de communicatie tussen de robot en deze processoren via een seriële of parallelle kabel. Deze constructie werd vervangen door één enkele pc met een USB-interface ([CM01]). Ook de programmatuur die de robot aanstuurde werd geactualiseerd: de originele Modula 2-code werd omgezet naar de Java-programmeertaal ([VDV01, Ven02]).

1.1.2 Doelstelling

Tenslotte werd vorig jaar een scriptie uitgeschreven met het doel een 3D-simulatie van de robot tot stand te brengen ([Pel01]). Mijn scriptie is dan ook het logische vervolg op de al geleverde inspanningen hieromtrent. Aangezien het de bedoeling is dat het bewakingssysteem kan gebruikt worden zonder al te veel voorkennis van computers, werd het duidelijk dat er nood was aan moderne en gebruiksvriendelijke programmatuur voor de aansturing ervan. Het ontwerpen van die programmatuur werd mijn taak.

Figuur 1.1: De Scorbot ER III

1.2 De beschikbare technologie: een overzicht

De robotica heeft de afgelopen decennia een grote weg afgelegd. Ze is geëvolueerd van een marginale zijtak van de wetenschap -voor buitenstaanders badend in een science-fictionsfeer - tot een onmisbaar hulpmiddel voor de industrie en wetenschap van vandaag. Gedurende deze evolutie is er echter steeds nood gebleken aan twee zaken:

1. Simulatie; een model dat precies de bewegingen die de robot maakt (of zou maken) kan weergeven, dikwijls op een computerscherm.

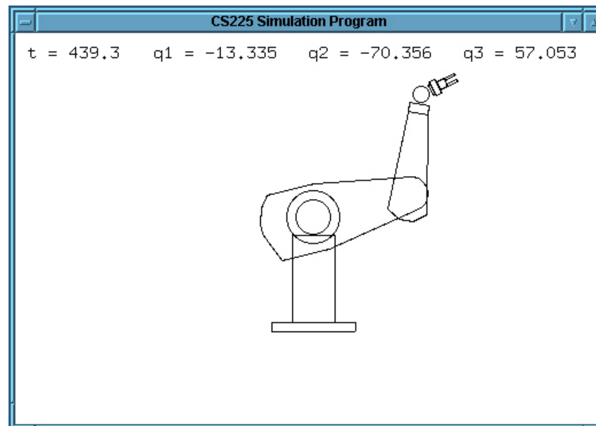
2. Een invoer- en/of leermethode; uiteraard moet het mogelijk zijn om op een gestructureerde wijze te bepalen welke taken een robot dient uit te voeren. Dikwijls is het wenselijk dat men hierbij in zekere mate kan experimenteren of de robotaansturder een zekere handeling interactief aan kan leren.

Bij het maken van een geschikte keuze wat deze criteria betreft, is het dan ook noodzakelijk om de bestaande technieken te onderzoeken en tegen elkaar af te wegen.

1.2.1 Simulatie

Een robot is geen eenvoudig werktuig als een ander. Enerzijds is er de technische complexiteit die erachter schuil gaat en die ervoor zorgt dat het niet eenvoudig is om een precies beeld te krijgen van de de positie ervan op een bepaald moment. Anderzijds schuilt er ook een potentieel gevaar in: de robot kan bij foutieve sturing zichzelf schade toebrengen of, nog erger, personen verwonden (en in extreme gevallen zelfs doden). Tenslotte is er in de industrie bij het invoeren van een nieuw programma niet altijd de mogelijkheid om een productielijn stil te leggen, testen uit te voeren en dan pas weer verder te gaan met de productie.

Het spreekt voor zich dat het daarom aangewezen is over een simulatiemodel te beschikken. Dat kan er dan voor zorgen dat op voorhand, zonder de echte robot zelf te



Figuur 1.2: Een 2D-model

activeren, bepaalde bewegingen getest kunnen worden en indien nodig bijgestuurd worden. Als de robot zich niet op dezelfde plaats bevindt als diegene die de robot bestuurt bewijst een grafisch model uiteraard ook zijn diensten. Vraag is: welke soorten modellen kan men gebruiken?

1.2.1.1 Tweedimensioneel model

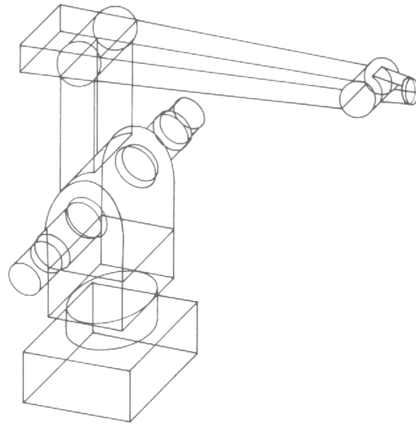
Zoals de naam al aangeeft, wordt er slechts met twee dimensies gewerkt binnen dit model. Dit model was vroeger populair omwille van de geringe rekenkracht die ervoor vereist is. Het volstaat voor zeer eenvoudige toepassingen, maar het verlies aan dieptezicht is moeilijk te verantwoorden voor meer ingewikkelde machines. Figuur 1.2 geeft een dergelijk model weer.

1.2.1.2 Draadmodel

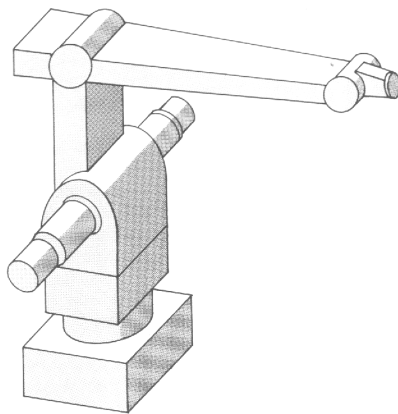
Voor toepassingen die nood hebben aan dieptezicht was het draadmodel (figuur 1.3) lange tijd bijzonder populair. Het geeft de gebruiker een goed beeld van de positie van de robot, terwijl het niet erg rekenintensief is. Een variant die veeleisender is, maar nog steeds op hetzelfde principe gebaseerd is, verwijdert alle (voor het oog) onzichtbare lijnen, met een realistischer beeld tot gevolg.

1.2.1.3 Volwaardig 3D-model

Met het verbeteren van de rekencapaciteit en de intrede van speciale 3D-processors werd het mogelijk om nog realistischer te werk te gaan. In plaats van een driedimensioneel beeld te gaan voorstellen door alleen lijnen, kon ook aandacht besteed worden aan de "vulling" van de vlakken en werd het zelfs mogelijk om schaduwen toe te voegen. Uiteraard zorgt deze manier van voorstellen voor een bijzonder natuurgetrouwe weergave, zoals blijkt uit het voorbeeld uit figuur 1.4. Het lag dan ook voor de hand om voor dit type simulatie te kiezen bij het ontwerpen van een interactief model voor het project Cycloop.



Figuur 1.3: Een draadmodel



Figuur 1.4: Een volwaardig 3D-model

1.2.2 Aansturing

Een goede grafische weergave is echter niet genoeg, de robot moet ook op een vlotte manier bestuurd kunnen worden. Qua invoer bestaan ook hier weer verschillende mogelijke methodes. Een onderscheid kan gemaakt worden op basis van de manier waarop de robot geïnstrueerd wordt. Is er een rechtstreekse invoer van een mens die de robot op één of andere manier laat bewegen, dan is er sprake van on-line operaties. In het andere geval, waarbij een eerder ingevoerd programma eenvoudigweg uitgevoerd wordt zonder verdere interactie spreekt men over off-line besturing. Wat volgt is een beknopt overzicht van de mogelijke invoerprocessen die vandaag gebruikt kunnen worden.

1.2.2.1 Doorleidmodus (Lead Thru)

Een persoon neemt de robot, die voorzien is van een servomotor, beet en voert de beweging uit die de robot later zelfstandig moet uitvoeren. Het voordeel van deze methode is dat men de aansturing niet hoeft te abstraheren en direct kan aangeven wat er gebeuren moet. Dit is terzelfdertijd de zwakte ervan, dikwijls is er nood aan een soort normalisatie die de kleine fouten die de aanstuurder maakt neutraliseert. Bovendien zijn niet alle robots gemakkelijk manipuleerbaar, vooral de elektrisch gestuurde niet, wat de toepassing ervan voor de Scorbot ER III onmogelijk maakt.

1.2.2.2 Leermodus (Teach mode)

Om tegemoet te komen aan de beperkingen van de doorleidmodus zijn er verschillend afstandsbedieningen ontworpen om een robot te programmeren. Daarbij beweegt men de verschillende onderdelen stap voor stap en slaat men de stand tussen de verschillende posities op. Hoewel dit systeem veelvuldig gebruikt wordt is het bijvoorbeeld niet mogelijk om er een ingevoerd programma met over te zetten naar een andere robot.

1.2.2.3 Programmering

Offline programmeren Een totaal andere aanpak bestaat erin om de robot op voorhand te gaan programmeren zonder echte interactie met de machine zelf. Net zoals men een computerprogramma algoritmisch opbouwt, zal men de commando's invoeren via bijvoorbeeld een tekst-gebaseerde taal. Een voorbeeld van invoer is te vinden in het volgende codefragment (afkomstig uit het torens van Hanoi-voorbeeld uit [Ven02]).

```
MoveCom.MoveGripperHorizontal("Cart", transl, 0f);
currentdisc = towers[from][position];
System.out.println("Disc : " + currentdisc);
MoveCom.OpenGripper ((float)(36 + 3*currentdisc));
transl.Z = (float) (168 + 30*position);
MoveCom.MoveGripperHorizontal("Cart", transl, 0f);
MoveCom.CloseGripper();
transl.Z = topPosition;
MoveCom.MoveGripperHorizontal("Cart", transl, 0f);
```

Deze methode is ongetwijfeld de meest veelzijdige omwille van de mogelijkheid tot integratie met bestaande algoritmen. Voor toepassingen zonder menselijke bestuurder (het merendeel binnen industrieel gebruik dus) is het ook de enige mogelijke invoermodus.

Online programmeren Als men dezelfde principes als hierboven toepast op een interactieve waretijdsomgeving, combineert men de voordelen van een afstandsbedieninggestuurde robot met een machine-onafhankelijke werking van de besturing. Als men een aangepaste GUI voorziet en de vertraging voor het verzenden van een commando naar de robot zo klein mogelijk houdt, is het resultaat een zeer handig invoermodel. Het is in dit opzicht dan ook logisch dat de keuze voor het besturen van de Cycloop-robot op deze methode viel.

1.2.2.4 Projectieve VR

Onder impuls van onder meer de ruimtevaart wordt voortdurend onderzoek gedaan naar nieuwe manieren van robotgebruik. De in [FR99] beschreven Projectieve Virtuele Realiteit is daar één van de vruchten van. Deze techniek maakt het mogelijk om – zonder de robot rechtstreeks te besturen – bepaalde opdrachten door te geven. Verplaatst men in een computergegenereerde scène bijvoorbeeld een doos van punt A naar punt B, dan zal het bovenliggende systeem zelf de robot(s) opdracht geven om de doos vast te grijpen op punt A , te verplaatsen naar punt B en tenslotte de doos ook weer neer te zetten. Op die manier hoeft de gebruiker niets af te weten van het gebruikte robotsysteem. Hoewel dit een veelbelovende evolutie is, is het voorlopig hoofdzakelijk geschikt voor opdrachten die een zekere routine vertonen.

1.2.2.5 Hybride technieken

Tenslotte zijn in de loop van de tijd ook verschillende technieken ontworpen die de eigenschappen van de vorige methodes bundelen. Zo is er bijvoorbeeld in [ASZ$^+$02] een systeem gebouwd voor medische doeleinden waarbij de gebruiker via een GUI aangeeft welk deel van het lichaam onderzocht moet worden en waarbij de robot via beeldverwerkende software de meer precieze positie gaat bepalen. Op die manier wordt een samenwerking bekomen tussen de menselijke operator en het aansturende programma.

Een dergelijk opzet leek me bijzonder interessant om toe te passen op het project Cyloop: de bewaker geeft de camera aan welk deel van een ruimte er in het oog gehouden moet worden en het bewakingssysteem zorgt er zelf voor dat bewegende zaken automatisch gevolgd worden. Hierbij is het dan op elk moment mogelijk om handmatig de camera te gaan bijsturen.

1.3 Vereisten voor de virtuele robot

Op basis van de voorgaande vergelijking werd het uiteindelijk duidelijk op welke basis de uitwerking van de virtuele robot gestoeld zou zijn. De streefdoelen werden:

- Een driedimensionale robot, omwille van de hoge graad van natuurgetrouwheid van een dergelijk model.
- De mogelijkheid om de robot aan te sturen vanuit op voorhand opgestelde programma's. Dit is zeker met het oog op beeldverwerkende algoritmes als stuurkracht achter de bewegingen onontbeerlijk.
- De aanwezigheid van een GUI om op een natuurlijke wijze de robot on-line te besturen.
- Aangezien de virtuele robot ook als zuiver simulatiemodel moet gebruikt kunnen worden, is er ook nood aan het verwerken van invoer afkomstig van het Java-gebaseerde stuurprogramma. Op die manier kan op het scherm steeds gezien worden in welke positie de robot zich zou bevinden bij een echte uitvoering van een programma.

Hoofdstuk 2

Platformkeuze

Nu de doelstellingen duidelijk afgelijnd zijn, is de volgende stap het kiezen van een geschikt platform waarop de programmatuur zal gerealiseerd worden. Dankzij de explosieve groei van 3D-toepassingen de laatste jaren is er een vrij grote keuze wat programmeertalen en omgevingen betreft. Het is bijgevolg belangrijk om van bij de start een goed gefundeerde keuze te maken, teneinde achteraf niet voor onaangename verrassingen komen te staan.

2.1 Noden

Het ideale implementatieplatform voor een project als dit onderscheidt zich door volgende kenmerken:

2.1.1 Een meerlagenmodel voor de 3D-rendering

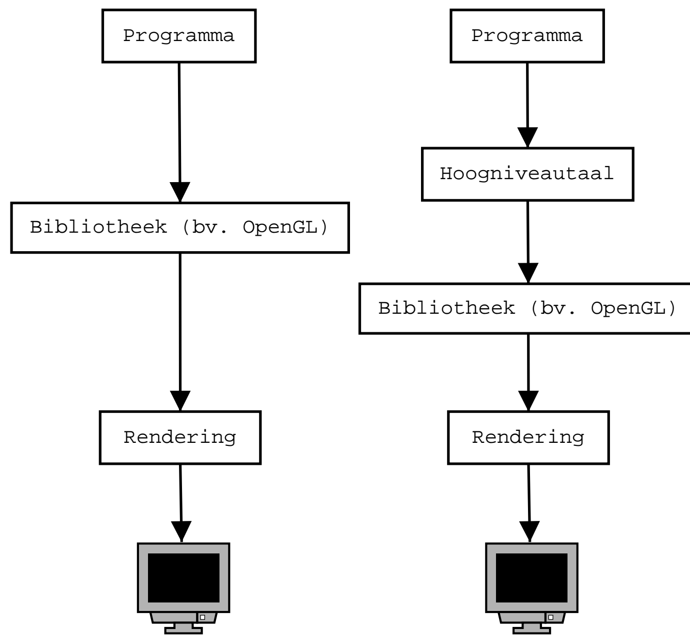
Voor het implementeren van toepassingen die gebruik maken van driedimensionale scènes, doet men traditioneel een beroep op een daarvoor geschikte 3D-bibliotheek. Men roept dan vanuit het eigen programma een functie op, de bibliotheek zorgt op zijn beurt voor de correcte rendering (i.e. de weergave op een Tweedimensionaal scherm).

Door een beroep te doen op deze onderliggende laag zorgt men voor een gemakkelijke overdraagbaarheid naar andere hard- en/of software. Zo wordt het mogelijk één zelfde programma met een minimum over te zetten naar een ander besturingssysteem. Analoog kan men een beroep doen op om het even welke grafische processor ("hardware rendering"), of – bij gebrek hieraan – deze laten emuleren door de CVE (het zogenaamde "software renderen"). De momenteel meest gebruikte 3D-bibliotheken zijn ongetwijfeld OpenGL[1] en DirectX[2], hoewel de laatste meer op spelletjesontwikkeling gericht is.

Met het toevoegen van een extra laag boven de eerdergenoemde lagen (figuur 2.1.1) kan men de laatste afhankelijkheid, die van de renderbibliotheek, vermijden. Daarenboven bieden talen op dit niveau de luxe dat de gebruiker ervan zich kan toespitsen op

[1] http://www.opengl.org

[2] http://www.microsoft.com/directx



(a) model met 2 lagen (b) model met 3 lagen

Figuur 2.1: mogelijke lagen bij het renderen

het ontwerpen van 3D-toepassingen, zonder dat een uitgebreide kennis nodig is over de onderliggende bibliotheek.

Tenslotte is het interessant erop te wijzen dat door deze manier van werken men de prestatie verkrijgt van een laagniveauoplossing, met de mogelijkheid alles toch te implementeren op een hoog niveau. De kost van de bijhorende vertaling is zo goed als verwaarloosbaar, zoals zal blijken uit verdere metingen.

2.1.2 Ondersteuning voor communicatie

Omdat het de bedoeling is dat er een verbinding tot stand komt tussen de virtuele en de echte robot, moet er uiteraard voorzien zijn in een mogelijkheid om via het Internet te communiceren. Daar zowat alle platformen tegenwoordig deze mogelijkheid bezitten, gaf dit weinig problemen.

2.1.3 Een generische oplossing

In een ideale implementatie moet de code gescheiden zijn van de specifieke gegevens van één robot. Als er overgeschakeld wordt op een andere robot met dezelfde eigenschappen, zou de code dus herbruikbaar moeten zijn.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 0.0 0.5 1.0
      }
   }
   geometry Cylinder {
      height 2.0
      radius 1.5
   }
}
```

Figuur 2.2: VRML codefragment

2.2 VRML

2.2.1 Herkomst

Bij de opkomst van het Internet werd het duidelijk dat een standaard nodig was om driedimensionale scènes weer te geven in een browser. Deze standaard werd VRML (Virtual Reality Modelling Language), een tekstgebaseerd bestandsformaat dat een sterke gelijkenis vertoont met HTML. VRML is ontstaan in 1995 en is ondertussen aan versie 2.0 toe, waarbij vooral interactie en animatie verbeterd zijn ten opzichte van vroegere versies. Voor de grafische verwerking doet het een beroep op een meerlagenmodel zoal beschreven in paragraaf 2.1.1

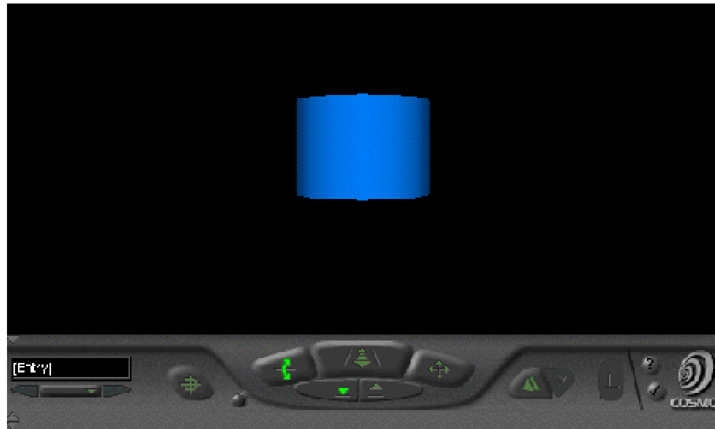
2.2.2 Paradigma

Het principe erachter is eenvoudig: maak een beschrijving van de objecten die zich binnen een virtuele wereld bevinden, samen met hun gedrag en andere gegevens. Als men dan met een programma (alleenstaand of een plug-in voor een browser) het bestand opent krijgt men een kijk op de gedefinieerde wereld. Het volgende codefragment uit figuur 2.2 toont aan hoe een VRML-bestand er uit ziet. In figuur 2.3 wordt het grafische resultaat weergegeven.

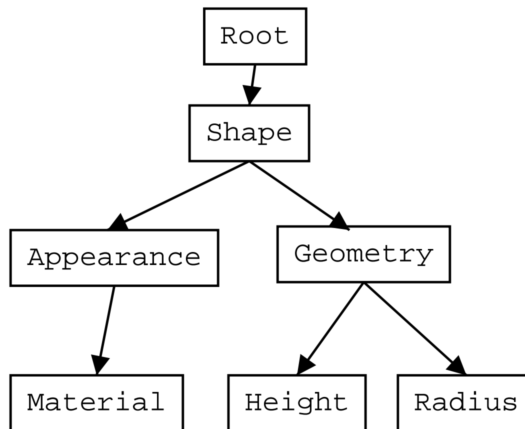
De essentie van de VRML-specificatie[3] bestaat uit het concept van de scènegraaf. Dit is een gerichte acyclische graaf waarin de toppen de rol spelen van een driedimensionale vorm of van een beschrijving van een transformatie die de kinderen van die top ondergaan.

Om deze omgeving met een programma aan te sturen heeft men de Java External Authoring Interface (EAI) nodig. Deze is oorspronkelijk ontwikkeld als een aanvulling bij de VRML-standaard, en dient als extra plug-in geïnstalleerd te worden. Als dit ge-

[3]http://www.vrml.org/technicalinfo/specifications/vrml97/



(a) resultaat



(b) DAG

Figuur 2.3: Het resultaat van de code uit fig. 2.2 en de bijhorende DAG

beurd is, kan men in twee richtingen communiceren tussen de virtuele wereld en het Java-hoofdprogramma.

2.2.3 Kwalitatieve analyse

Zoals al eerder vermeld heeft Stéphane Peltot vorig jaar een simulatieomgeving ontwikkeld voor de Scorbot ER III. Daarvoor heeft hij een beroep gedaan op VRML om de robot te beschrijven. Voor het programmeren van invoerverwerking werd initieel een beroep gedaan op EAI, maar toen dit enkele praktische problemen gaf met de invoer van tekstbestanden werd er overgeschakeld naar een zuiver Java-programma. Dat zorgde ervoor dat verschillende tekstbestanden werden ingelezen en tot één VRML-bestand werden herwerkt. Voor de details hier rond verwijs ik naar [Pel01], maar het mag duidelijk zijn dat dit geen ideale situatie was.

Gegeven de inspanningen die al geleverd zijn om de robotsimulatie met VRML te ontwerpen, zou men ervoor kunnen pleiten om op de weg van EAI verder te gaan. Bij een grondige evaluatie kwamen nog verscheidene nadelen van VRML voor meer omvangrijke projecten naar boven, onder meer in [SECW99] en [Bou97]:

- EAI is pas ontworpen nadat de VRML-standaard er was en daar lijdt de samenhang tussen beide delen nogal onder. Voor kleine projecten hoeft dat geen bezwaar te zijn, maar wanneer men een meer generische oplossing probeert te construeren met een bepaalde complexiteit is de EAI niet meteen de gemakkelijkste manier van werken.

- Automatische botsingsdetectie (tussen verschillende lichamen) bestaat bij VRML (huidige versie, 2.0) nog niet.

- Er bestaat niet zoiets als een standaard VRML plug-in. Als men meer geavanceerde functies zoals EAI wil gebruiken, wordt men automatisch beperkt tot enkele grote namen die niet voor alle platformen beschikbaar zijn.

- Zo goed als onbestaande ondersteuning voor speciale in- en uitvoerapparaten zoals hoofdgemonteerde schermen, 3D-muizen en joysticks.

- Er enkele bijzonder vervelende tekortkomingen aan de VRML-taal zelf, zoals problemen met het opsplitsen van ingewikkelde scènes over meerdere bestanden.

- Tot slot is een uitgebreid VRML-bestand, in vergelijking met de gangbare programmeertalen, bijzonder moeilijk leesbaar.

2.3 Java3D

2.3.1 Herkomst

Een ander platform voor het ontwikkelen van 3D-applicaties is Java3D. Zoals de naam al laat vermoeden, is dit een uitbreidingsbibliotheek voor de Java-programmeertaal. Ze

is in 1997 in het leven geroepen door enkele instanties binnen de grafische wereld, met name Intel, Silicon Graphics, Apple en Sun. Het doel was een API te ontwerpen die platformonafhankelijk was maar toch de prestaties leverde die nodig zijn voor hedendaagse complexe 3D-toepassingen. Na een ontwikkelingsfase die 4 jaar duurde werd in 2001 de eerste versie van het Java3D-platform vrijgegeven, eveneens gebaseerd op een grafisch meerlagenmodel (zie § 2.1.1). Tegenwoordig zijn implementaties voorhanden voor Linux (x86/ppc), Windows, HP UX, Solaris, Irix en (binnen afzienbare tijd) Mac OS X.

2.3.2 Paradigma

Wat de datastructuren voor de opslag van scènes betreft, bestaat er een grote gelijkenis tussen VRML en Java3D. Bij elk van de twee wordt een zogeheten scènegraaf (scenegraph) opgesteld, waarbij de verschillende onderdelen aan de takken van deze DAG hangen. Deze gelijkenissen zijn niet toevallig, maar zijn gebaseerd op onderling overleg tussen de ontwerpers van Java3D en het VRML-consortium[4].

Het grote verschil met VRML zit hem in de mogelijkheden van Java als programmeertaal. Waar VRML een beroep moet doen op uitbreidingen die nooit een erg robuuste indruk maken, heeft Java3D al van bij het ontstaan een complete objectgeoriënteerde omgeving aan boord. Daarmee wordt een bijzonder stevige basis gelegd voor goed gestructureerde toepassingen die volledig voldoen aan de principes van moderne objectgeoriënteerde softwareontwikkeling.

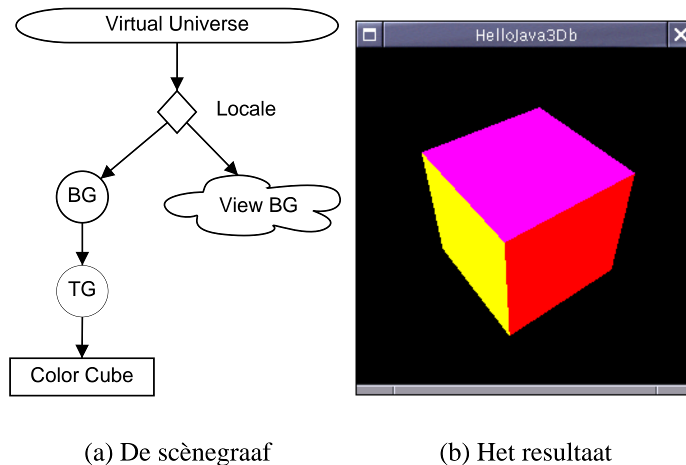
2.3.2.1 De taxonomie van de scènegraaf

Om iets dieper in te gaan op de manier van werken binnen het Java3D-kader is het interessant om het kleine voorbeeldprogramma te bekijken dat een geroteerde kubus op het scherm weergeeft in figuur 2.5. Als men deze code gaat compileren en uitvoeren, construeert Java3D – analoog met VRML – een scènegraaf. Deze datastructuur wordt opgebouwd door de programmeur en vormt de ruggengraat van de grafische verwerkingsalgoritmen waarvoor de bibliotheek instaat. Een voorbeeld ervan is te zien in figuur 2.4.

We onderscheiden volgende onderdelen:

- Het *virtual universe*, dit is het hele universum als het ware, alle weer te geven objecten bevinden zich binnen dit universum.
- De *locale*, een object dat uniek verbonden is met een coördinatenstelsel.
- De *view branch graph*, een deel van de graaf dat verband houdt met de manier van weergeven (bevat o.a. het camerastandpunt).
- Een *branch group (BG)*, de top uit de graaf die de container is waarin de eigenlijke lichamen zich bevinden.

[4] http://www.vrml.org/consort/sun98.html



(a) De scènegraaf (b) Het resultaat

Figuur 2.4: Scènegraaf en resultaat van de code uit figuur 2.5

- Een *transform group (TG)*, deze top bevat een ruimtelijke transformatie die toegepast zal worden op zijn kinderen. Het is mogelijk om TG's te nestelen: bij een rotatie na een translatie bijvoorbeeld heeft een TG een TG als kind.

In het codevoorbeeld wordt een BG aangemaakt, die als kind een TG heeft waar 2 rotaties mee geassocieerd zijn, één rond de X-as en één rond de Y-as. Deze TG heeft op zijn beurt een gekleurde kubus als kind, waardoor op het scherm een geroteerde kubus terecht komt. Hoewel dit fragment langer is dan een gelijkaardig VRML-bestand dat dezelfde scène voorstelt, komt de hoeveelheid code vooral voort uit de eenmalige constructie van de scènegraaf. Bij grotere programma's is deze hoeveelheid extra code dus zeker verwaarloosbaar. Bovendien laat Java toe om de overbodige details te verbergen door gebruik van inkapseling.

Eens de scènegraaf bij het uitvoeren opgesteld is, zal Java3D zonder tussenkomen van de programmeur zelf de nodige optimalisatiealgoritmen erop toepassen, teneinde een zo groot mogelijke prestatie te bekomen.

2.3.2.2 Loaders

Anders dan men op het eerste idee zou verwachten, ondersteunt Java3D geen serializatie van scènegrafen of onderdelen ervan. Men heeft bij de ontwikkeling ervan niet opnieuw het wiel willen uitvinden – een naar mijn mening zeer lovenswaardig principe. Er bestaan immers reeds vele ingeburgerde formaten voor het opslaan van driedimensionale objecten, dikwijls afkomstig uit de CAD- of raytracingwereld (respectievelijke voorbeelden: AutoCAD en PovRay). Door een methode te voorzien om deze formaten in te lezen werd het mogelijk om met bijzonder complexe lichamen te werken zonder dat men deze volledig hoeft op te bouwen met behulp van Java-commando's.

Om deze manier van werken zo eenvormig als mogelijk te houden, biedt Java3D een interface, *Loader*, aan. Deze bevat alle standaardmethodes om een ingelezen bestand

```
// HelloJava3Db geeft een geroteerde kubus weer.

public class HelloJava3Db extends Applet {
  public BranchGroup createSceneGraph() {
    // Maak de wortel van de branch graph
    BranchGroup objRoot = new BranchGroup();
    // rotate-object heeft een
    // samengestelde transformatiematrix
    Transform3D rotate = new Transform3D();
    Transform3D tempRotate = new Transform3D();
    rotate.rotX(Math.PI/4.0d);
    tempRotate.rotY(Math.PI/5.0d);
    rotate.mul(tempRotate);
    TransformGroup objRotate = new TransformGroup(rotate);
    objRoot.addChild(objRotate);
    objRotate.addChild(new ColorCube(0.4));
    // Laat Java 3D deze scènegraaf optimaliseren.
    objRoot.compile();
    return objRoot;
  } // einde van CreateSceneGraph

// Maak een eenvoudige scène en voeg die toe
// aan het virtuele universum
  public HelloJava3Db() {
    setLayout(new BorderLayout());
    GraphicsConfiguration config =
    SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas3D = new Canvas3D(config);
    add("Center", canvas3D);
    BaranchGroup scene = createSceneGraph();
    SimpleUniverse sU = new SimpleUniverse(canvas3D);
    // Verplaats het ViewPlatform wat om alles
    // in beeld te brengen
    sU.getViewingPlatform().setNominalViewingTransform();
    sU.addBranchGraph(scene);
  } // einde van HelloJava3Db (constructor)

  public static void main(String[] args) {
    Frame frame =
    new MainFrame(new HelloJava3Db(), 256, 256);
  } // einde van main

} // end van HelloJava3Db
```

Figuur 2.5: Codefragment van een programma in Java3D

te integreren in een scènegraaf. Om nu een bepaald bestandsformaat te ondersteunen, volstaat het een object te creëren dat deze interface implementeert. Deze manier van werken brengt met zich mee dat de applicatie-ontwerper zich niet hoeft te bekommeren over het formaat waarin de 3D-objecten zijn opgeslagen. Die details worden immers afgehandeld door het gebruikte Loader-object.

Een loader die bijzonder goed van pas kan komen is de VRML-loader. Hiermee wordt het mogelijk om de grafische beschrijving van de robot die vorig jaar gemaakt werd in te lezen in een Java3D-programma; dit spaart het mogelijk dubbele werk uit om de robot opnieuw te tekenen.

2.3.2.3 Gedragingen (Behaviors)

Waar nodig wordt binnen de Java3D API gebruik gemaakt van draden. Dit is het geval voor het renderen en ook de zogenaamde gedragingen. Dit zijn acties die gebeuren wanneer er zich een op voorhand gedefinieerde actie, een stimulus, voordoet. Het kan bijvoorbeeld gaan om een bepaalde tijd die verstrijkt, invoer door een invoerapparaat, of een botsing van twee lichamen.

2.3.3 Kwalitatieve analyse

Globaal gezien biedt dit platform een stevige basis voor de opbouw van vrij complexe programma's. Het grote pluspunt is de perfecte integratie met het hele Java-platform, waardoor iemand met ervaring met deze taal zich vrij gemakkelijk kan inwerken in de Java3D-bibliotheek. Als nadeel kan men aanhalen dat voor sommige projecten er nood zal zijn aan een programmeertaal die op een lager niveau binnen het renderproces werkt (meer precies de nood aan het handmatig aanpassen van de rendering-pijplijn). Het is echter duidelijk dat Java3D niet bedoeld is voor dergelijke laagniveauprogrammering.

2.4 Conclusie

Na een vergelijking tussen VRML en Java3D voor het implementeren van de virtuele robot kwam ik tot de conclusie dat de ideale oplossing erin zou bestaan om van beide technieken gebruik te maken.

VRML is zeer geschikt voor zijn oorspronkelijke doel: een beschrijving van een driedimensioneel lichaam in een standaardformaat. Voor het eigenlijke programmeren van 3D-applicaties is Java3D dan weer een geschikte keuze. Door gebruik te maken van de Loader-interface haalt men het beste van de twee werelden in huis en is het bovendien mogelijk om gebruik te maken van de reeds geleverde inspanningen voor het weergeven van de robot in VRML.

Hoofdstuk 3

Implementatie

Nu er een duidelijke keuze gemaakt is wat de programmeeromgeving betreft, is het tijd om de details van de virtuele robot van dichterbij te bekijken. Er wordt ingegaan op de anatomie van de robot, hoe die voorgesteld wordt in de virtuele wereld, hoe de verschillende onderdelen tot beweging komen. Verder wordt de code gedocumenteerd aan de hand van verschillende schema's.

3.1 Anatomie van de robot

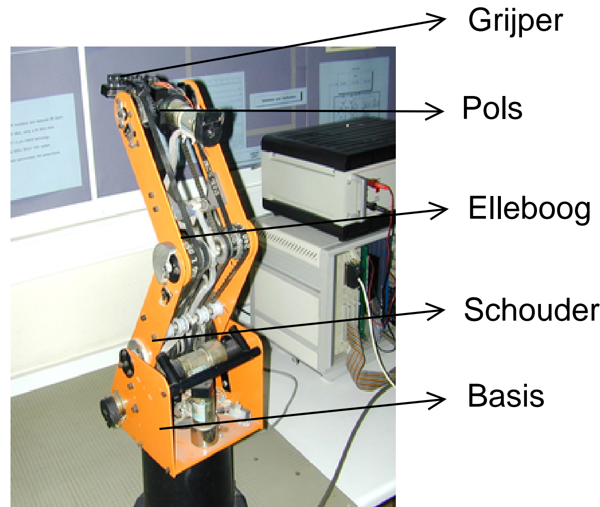
De Scorbot ER III is een robot met 5 vrijheidsgraden en een grijper die open en dicht kan gaan. De verschillende onderdelen van een robot worden traditioneel benoemd met hun menselijke equivalenten en dat is in dit geval niet anders. We onderscheiden volgende delen:

- De *basis* (Engels[1]: *base*), het laagste beweegbare onderdeel, waar de robotarm op rust.
- De *schouder (shoulder)* vormt de onderste hoek van de arm met de basis.
- De *elleboog (elbow)* is het deel dat op de schouder volgt en als het ware het "midden" van de arm vormt.
- De *pols (wrist)* volgt op zijn beurt op de elleboog.
- De *grijper (gripper)* is als het ware de hand van de robot, deze kan open en dicht schuiven.

De precieze positie van bovengenoemde delen is te vinden op figuur 3.1.

Elk van deze onderdelen kan onafhankelijk bewegen. De gewrichten (basis tot en met pols) kunnen roteren, de grijper kan zich openen en sluiten. De pols is in dat opzicht

[1]Om de programmeerstijl zo eenduidig mogelijk te houden, is ervoor gekozen om in de javacode te werken met de Engelstalige benamingen van de robotonderdelen. Deze werden immers al gebruikt in de vroegere code om de robot aan te sturen, cfr. [VDV01].



Figuur 3.1: De verschillende lichaamsdelen van de robot

een speciaal geval: hij kan zowel rond zijn eigen as draaien (*twist*) als kantelen *(tilt*). De hoeken die overeenkomen met de vrijheidsgraden van de robot zijn aangeduid op figuur 3.2.

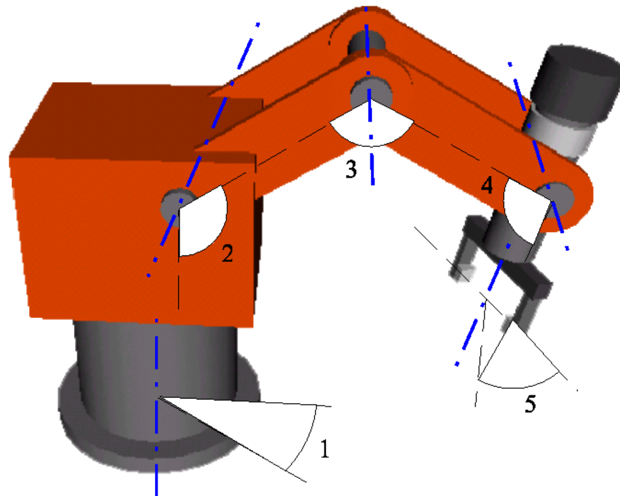
3.2 Beschrijving in VRML

3.2.1 De bestaande code

Gezien er weinig specifieke literatuur beschikbaar is in verband met de Scorbot ER III, was Stéphane Peltot vorig jaar reeds verplicht om de afmetingen van robot handmatig op te meten. De resultaten daarvan verwerkte hij in verschillende versies van VRML-bestanden. Deze bestaan uit een opeenvolging van constructies van geometrische vormen met de daarop uitgevoerde rotaties en translaties. Verder zijn ook zaken als kleur en het materiaal van de lichamen erin verwerkt. In de latere versies verschijnen ook meer specifieke gegevens, zoals virtuele knoppen om de robot aan te sturen. Aangezien we in casu VRML alleen wensen te gebruiken als formaat om het robotmodel in op te slaan, volstond de eerste versie van de reeks VRML-bestanden voor onze doeleinden.

3.2.2 De aanpassingen

Omdat we de verschillende delen willen laten bewegen ten opzichte van elkaar zou de gemakkelijkste oplossing erin bestaan om elk deel van de robot op te slaan in een aparte BranchGroup, die dan als ouder een rotatie-TransformGroup kan krijgen. Gezien een loader steeds een volledige BranchGroup inleest per VRML-bestand, bestond de eerste taak erin om het originele VRML-bestand zo te gaan opsplitsen dat er zich in elk bestand juist één beweegbaar onderdeel bevindt.



Figuur 3.2: De hoeken tussen de verschillende beweegbare onderdelen

Na wat experimenteren met de VRML-code is dit inderdaad gelukt, zij het met enkele kleine aanpassingen. Zo moest in elk van de zes bestanden apart de kleur van de onderdelen gespecificeerd worden. Er waren nu zes VRML-bestanden met de naam "deel$n$.wrl", waarbij n voor een nummer staat van 1 tot en met 6. In dit stadium was het mogelijk om de onderdelen apart weer te geven op het scherm, als ze alle tegelijk werden getoond leek de robot echter te bestaan uit verschillende zwevende onderdelen, elk op een willekeurige plaats.

De oplossing hiervoor leek nogmaals te bestaan in het manueel aanpassen van de VRML-omschrijvingen, ditmaal met de bedoeling dat de stukken die met elkaar verbonden zijn nu ook bij het virtuele model in elkaar pasten. Na wat trial-and-error zoekwerk was er eindelijk een consistent beeld van de robot te zien.

3.3 De opbouw in Java3D

3.3.1 De definitieve scènegraaf

De robotonderdelen worden nu voorlopig opgeslagen in de zesdelige array "parts". Volgende stap is de rotaties van de beweegbare onderdelen mogelijk maken. Daarvoor moeten de ledematen als ouder een TransformGroup toegewezen krijgen die hun bewegingsmogelijkheden specificeert. Onder de wortel van de scène bevindt zich nog eens een TransformGroup die ervoor zorgt dat robot herschaald wordt, dit met de bedoeling dat alles op het scherm kan. Een grafische voorstelling van de scènegraaf, op de ViewBranch na, is te vinden in figuur 3.3.

In deze constructie is het sluiten van de grijper nog niet opgenomen; wegens een mechanisch defect was het immers onmogelijk om deze aan te sturen. Het mag echter duidelijk zijn dat slechts een kleine aanpassing vereist is om deze mogelijkheid toe te voegen. Het zal volstaan nog een TransformGroup onderaan de graaf toe te voegen, als

kind van de onderste TransformGroup, die voor de translaties kan zorgen die hiervoor nodig zouden zijn.

3.3.2 Driedimensionale transformaties

Volgende stap binnen het ontwerpproces was het correct laten roteren van de gewrichten. Dit houdt een sterk verband met de wijze waarop driedimensionale transformaties door Java3D afgehandeld worden.

Java3D maakt voor het voorstellen van punten in de driedimensionale wereld gebruik van de zogeheten *genormaliseerde homogene coördinaten*. Homogene coördinaten houden in dat, in tegenstelling tot het klassieke cartesiaanse coördinatenstelsel dat slechts drie elementen vereist om een plaats binnen de 3D-wereld ondubbelzinnig vast te leggen, er nog een vierde element wordt toegevoegd aan de coördinaten. Om het cartesische equivalent te bekomen van een homogeen coördinaat, deelt men eenvoudigweg de eerste 3 coördinaten door het vierde coördinaat (dat i.c. uiteraard dient te verschillen van 0). Kort samengevat, als $w \neq 0$ dan geldt er:

$$(x, y, z, w)_{homogeen} \equiv (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})_{cartesisch}$$

Aangezien bij *genormaliseerde* homogene coördinaten de w-component steeds gelijk is aan 1, geldt dus in de praktijk:

$$(x, y, z, 1)_{homogeen} \equiv (x, y, z)_{cartesisch}$$

Op het eerste zicht kan deze aanpak misschien nodeloos ingewikkeld lijken, door een dergelijk coördinatenstelsel wordt het wel mogelijk om de belangrijkste ruimtelijke transformaties te laten gebeuren door een eenvoudige vermenigvuldiging met een 4x4-matrix. Bovendien wordt de toepassing van verschillende opeenvolgende transformaties herleid tot een eenmalige vermenigvuldiging met het product van de verschillende transformatiematrices. Het is bijgevolg logisch dat de klasse binnen Java3D die voor ruimtelijke transformaties instaat, de Transform3D-klasse, bestaat uit een dergelijke 4x4-matrix.

3.3.3 Rotaties in de praktijk

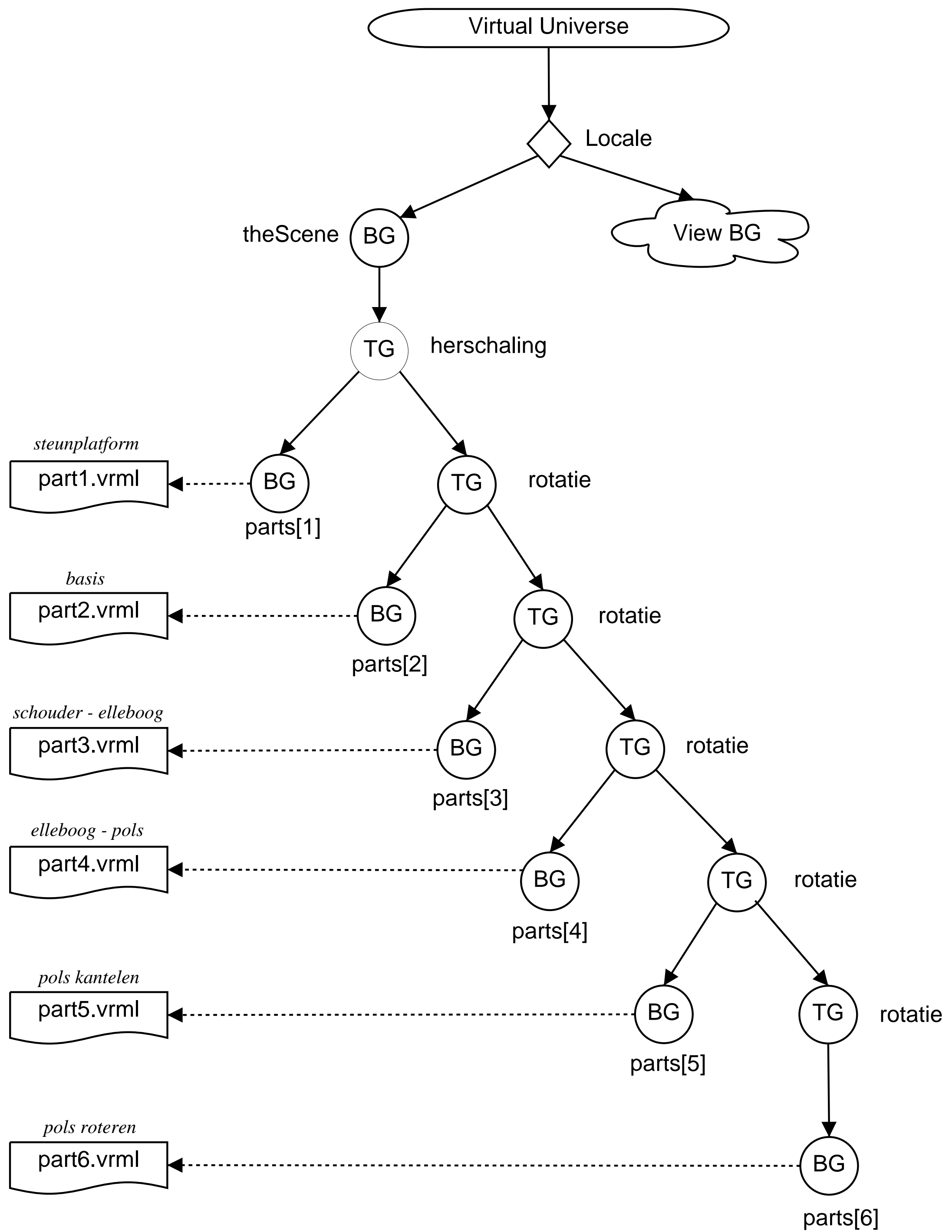
Als men nu een deel van de robot wil laten roteren zoals in de realiteit is er nood aan een rotatie om een as die evenwijdig is met één van de hoofdassen. Een overzicht hiervan is te vinden in tabel 3.1.

Aangezien het de bedoeling is dat een robotdeel om zijn eigen as roteert en niet om de X-, Y- of Z-as zal er nood zijn aan aangepaste translaties van en naar de oorsprong van het coördinatenstelsel. We kunnen de transformatie dan als volgt beschrijven:

1. Voer een translatie uit van het te roteren deel naar de oorsprong.

2. Voer de gewenste rotatie uit rond de gewenste as.

3. Voer opnieuw een translatie uit, dit keer naar het beginpunt.



Figuur 3.3: De opbouw van de inhoud-scènegraaf van de virtuele robot

Onderdeel	rotatie rond
basis	Y-as
shouder	as ‖ Z-as
elleboog	as ‖ Z-as
pols (kantelen)	as ‖ Z-as
pols (draaien)	as ‖ Y-as

Tabel 3.1: Rotatie-assen van de robotonderdelen

Een uitzondering op deze werkwijze is het basisgedeelte, daarvan bevindt het midden zich namelijk op de Y-as, wat de translaties overbodig maakt.

Voorbeeld

Het volgende voorbeeld illustreert deze werkwijze. Stel dat de we de schouder over een hoek van 0.1 radialen willen roteren. Dan is daar de volgende Java3D-code voor nodig, gesteld dat het aangrijpingspunt van de schouder zich op $(0.05, 0.35, 0, 1)$ bevindt.

```
double xtrans, ytrans, ztrans;
p1 = 0.05; p2 = 0.35; p3 = 0;
// translatie naar oorsprong
Transform3D trans = new Transform3D();
trans.setTranslation(new Vector3d(-p1,-p2,-p3));
// rotatie rond Z-as
Transform3D shoulderRotation = new Transform3D();
shoulderRotation.rotZ(alfa);
shoulderRotation.mul(trans);
// translatie weg van oorsprong
Transform3D backtrans = new Transform3D();
backtrans.setTranslation(new Vector3d(p1,p2,p3));
backtrans.mul(shoulderRotation);
// voer de transformaties door op het scherm
shoulderTransformGroup.setTransform(backtrans);
```

Deze code zal volgende matrices genereren:

$$T_p = \begin{bmatrix} 1 & 0 & 0 & p_1 \\ 0 & 1 & 0 & p_2 \\ 0 & 0 & 0 & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{-p} = \begin{bmatrix} 1 & 0 & 0 & -p_1 \\ 0 & 1 & 0 & -p_2 \\ 0 & 0 & 0 & -p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_\alpha = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

En tenslotte de volgende transformatiematrix genereren:

$$M_{schouder Rotatie} = T_p R_\alpha T_{-p}$$

Om nu de nieuwe punten (x', y', z', w') van de schouder te berekenen voert men voor alle punten (x, y, z, w) van de schouder volgende vermenigvuldiging uit:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = M_{schouder Rotatie} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Omdat de transformatiematrix pas op het einde van de rotatie-methode wordt gebruikt, ziet de gebruiker niet dat het onderdeel (i.c. de schouder) tussentijds verplaatst wordt naar de oorsprong om daar geroteerd te worden. In tegendeel, voor hem lijkt het alsof de schouder mooi ter plaatse blijft en daar roteert rond de Z-as.

Het enige wat verder nog nodig is, is de juiste methodes voor het invoeren van een rotatie uitwerken en het bijhouden van de huidige grootte van een hoek, wat te vinden is in appendix A.

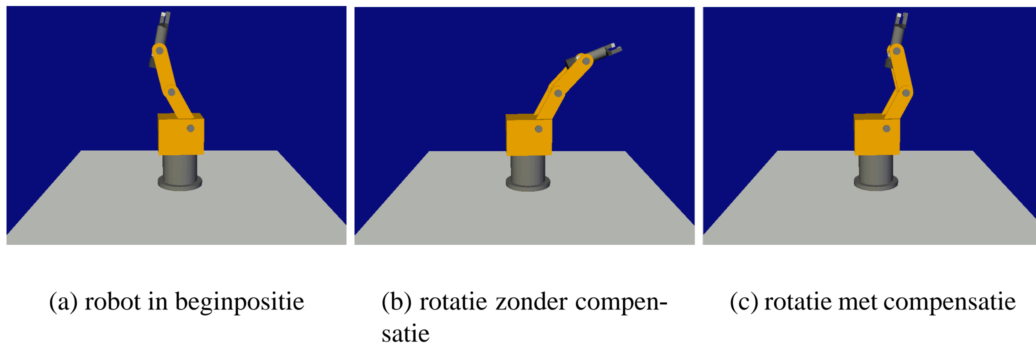
3.3.4 Bedenkingen bij interpolatie

Men zou zich kunnen afvragen of deze manier van roteren wel volstaat. Als de hoek van de draaiing immers een te groot verschil vertoont met de huidige positie waar het te roteren onderdeel zich in bevindt, komt men tot een schokkend beeld. Een dergelijk probleem zou kunnen opgelost worden door het invoeren van zogenaamde Interpolator-objecten. Men geeft bij een Interpolator 3 argumenten mee:

1. een beginpositie P_b van een object
2. een eindpositie P_e van een object
3. een tijdspanne T

Een interpolator zal dan gedurende de tijdspanne T voortdurend een tijdelijke positie van het object tussen P_b en P_e berekenen door interpolatie van de 3 parameters. Het gevolg is een vloeiende animatie.

Toch is het gebruiken van positie-interpolatie verre van ideaal, vooral dan wanneer men de virtuele robot laat reageren op invoer van de echte robot. Het gaat hier immers om een (quasi) waretijdstoepassing, waarbij men zich niet – vooral bij ware-tijd simulaties



(a) robot in beginpositie (b) rotatie zonder compensatie (c) rotatie met compensatie

Figuur 3.4: Overzicht van rotatiemethodes

– kan veroorloven om een bepaalde tijd te wachten tot een interpolator zijn werk verricht heeft vooraleer tot een volgende actie over te gaan. Bovendien zit men in dat geval steeds met een vertraging wat de invoer betreft – het is dus onmogelijk om een toekomstige positie te voorspellen. Dit maakt het gebruik van een interpolator zo goed als onmogelijk.

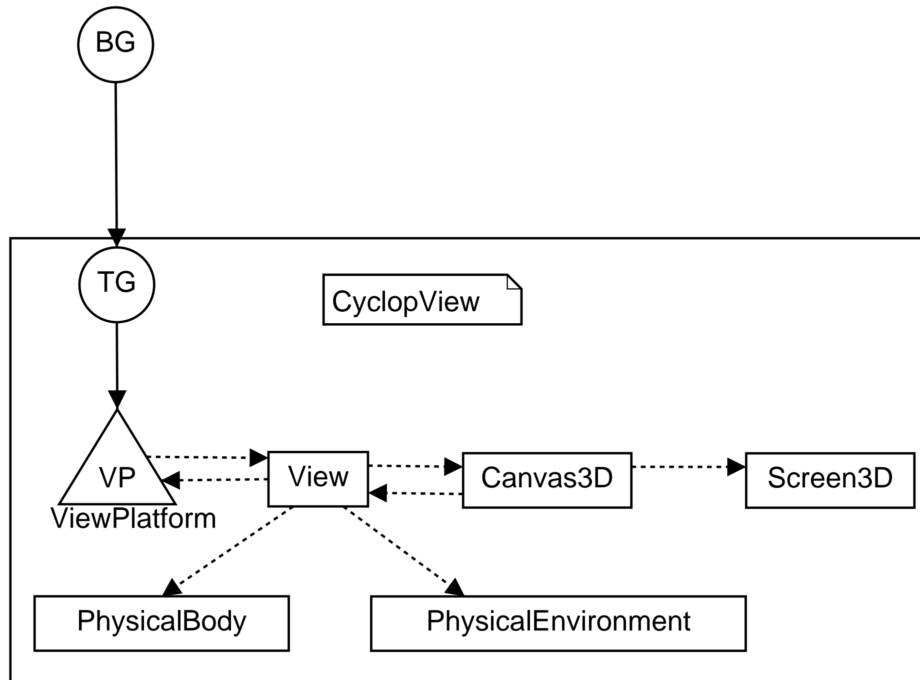
Wat de systeembelasting betreft (nog een belangrijke reden om voor interpolatie te kiezen) is er ook geen probleem. Bij alle testen bleek de opeenvolging van rotaties geen aanzienlijke processorbelasting met zich mee te brengen. Zolang de hoeken klein genoeg blijven is er eveneens geen sprake van schokkende beelden. Door het stroboscoopeffect[2] zijn opeenvolgende rotaties als één vloeiende beweging zichtbaar.

3.3.5 Intelligente rotaties

Met de hierboven beschreven technieken is het nu mogelijk om de robotledematen vrij natuurgetrouw te doen roteren. Toch schort er nog iets aan de manier waarop de rotaties de stand van de robot beïnvloeden. Als gevolg van de structuur van de scènegraaf zoals beschreven in figuur 3.3 op pagina 21 is er een ongewenst neveneffect. Als men bijvoorbeeld de schouder een bepaalde hoek laat draaien, zullen alle lager gelegen delen in de scènegraaf (i.e. de elleboog, de pols) eveneens over die bepaalde hoek gedraaid worden. Dit nevenverschijnsel is er verantwoordelijk voor dat de robot veel moeilijker bestuurbaar wordt via handmatige invoer. Een duidelijke illustratie van dit principe wordt gegeven in figuur 3.4, waar de schouder voor en na rotatie over een hoek van 1 radiaal getoond wordt.

Het is mogelijk om het rotatiegedrag "intelligenter" te maken, door deze nevenverschijnselen weg te nemen. Men compenseert dan de rotaties die teveel gemaakt zijn bij de kinderen van het draaiende gewricht door in de tegengestelde zin eenzelfde rotatie door te voeren. Zo compenseert men als het ware de te bruuske beweging. Er dient wel op gelet te worden dat er niet overgecompenseerd wordt: hiervoor is er beroep gedaan op een systeem met compensatievlaggen binnen het simulatieprogramma. De effecten van de gecompenseerde rotaties zijn eveneens zichtbaar in figuur 3.4.

[2]Het effect dat optreedt wanneer stilstaande maar enigszins veranderende beelden snel na elkaar worden aangeboden, zie ook [RDCP+96].



Figuur 3.5: De ViewBranchGraph in detail uitgewerkt

3.4 De camerapositionering

3.4.1 De ViewBranch-graaf

3.4.1.1 Structuur

De inhoudscènegraaf uit figuur 3.3 op pagina 21 verbergt nog een deel van de volledige scènegraaf onder de noemer “ViewBranch Graph”. Deze tak is verantwoordelijk voor wat in de 3D-grafiekwereld de camera (of het oogpunt) genoemd wordt. De precieze structuur ervan is terug te vinden in figuur 3.5.

3.4.1.2 Transformaties

Om het gezichtspunt van de gebruiker te laten veranderen zijn er twee mogelijkheden:

- Alle objecten laten bewegen, door een transformatie toe te passen op de wortel van de inhoudscènegraaf.
- Alleen de camera laten bewegen, door dezelfde transformatie op de wortel van de ViewBranch-graaf toe te passen.

Hoewel voor de eindgebruiker het eindresultaat hetzelfde blijft, is het aan te raden om de tweede manier te kiezen, en dit omwille van twee redenen:

- Het is veel minder rekenintensief om één object (in casu de ViewBranch-graaf) te transformeren dan alle objecten uit het virtuele universum te transformeren.

- Als men met een tweede camera die op het universum gericht is werkt, heeft het transformeren van alle objecten ongewenste neveneffecten op het tweede zichtpunt.

3.4.2 Vereenvoudigde universumopbouw

Bij eenvoudige toepassingen is het dikwijls niet nodig om zelf handmatig de ViewBranch-graaf op te stellen. De Java3D API biedt voor deze gevallen een voorgedefinieerd eenvoudig universum aan met de veelzeggende naam SimpleUniverse. Aangezien bij de Cycloopsimulatie echter nood is aan twee verschillende visies op het virtuele universum, was het onmogelijk hiervan gebruik te maken.

Om de hoge graad van complexiteit die gebonden is aan het handmatig opstellen van een eigen ViewBranch voor de gebruiker te verbergen, is het deel onder de ouder BranchGraph volledig opgenomen in de CyclopView-klasse. Zo wordt het veel doorzichtiger om twee verschillende visies aan te bieden zonder dat de eindgebruiker zich zorgen hoeft te maken over geavanceerde opties zoals het feit of de gebruikte zichtmethode gebaseerd is op dominantie van het linker- of rechteroog. Men kan een CyclopView-object informeel definiëren als een camera die binnen het virtuele universum geplaatst wordt en waarvan de oriëntering kan aangepast worden.

De belangrijkste velden van de CyclopView-klasse zijn de TransformGroup en het Canvas3D.

Op de TransformGroup kunnen transformaties uitgevoerd worden met als gevolg dat de camera zal geherpositioneerd worden (het gaat dan in de praktijk vooral om translaties).

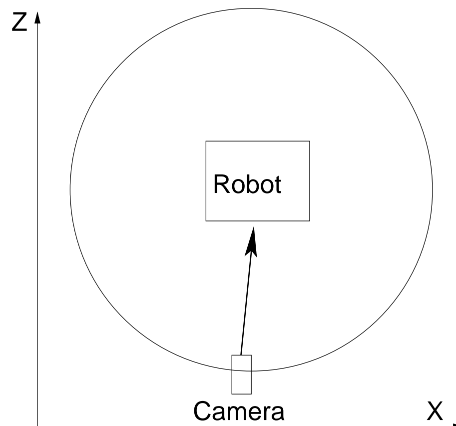
Het Canvas3D houdt een referentie bij naar het object dat voor de eigenlijke weergave van het driedimensionale tafereel zorgt. Dat dit object een kind is van het Canvas-object uit de AWT-bibliotheek gaf nogal wat problemen met de rest van de GUI, die met Swing gemaakt is. Het door elkaar gebruiken van lichtgewicht componenten (Swing) en zwaargewicht componenten (AWT) wordt door Sun sterk afgeraden, omwille van de mogelijke verkeerde overlappingen die dit kan veroorzaken. Toch is het mogelijk (en i.c. zelfs noodzakelijk) om beiden te combineren, zolang men er maar op let om de AWT-componenten steeds in een aparte container (bijvoorbeeld JPanel) te plaatsen.[3]

3.4.3 Twee camera's voor de cycloop

Bij het opstellen van de scènegraaf worden er twee CyclopView-objecten geïnstantieerd, één voor het hoofdzicht en één voor een miniatuurversie van een bovenaanzicht. Uiteraard hoeft men zich hiertoe niet te beperken, er kunnen in principe oneindig veel camera's gebruikt worden. Om de rekenkrachtvereisten niet te hoog te laten oplopen, is er echter voor gekozen dit aantal te beperken tot 2.

Er bestaan vele manieren om een camera bewegingen te laten maken, maar deze zijn zeker niet allemaal geschikt in dit geval. Standaard is er een soort aansturing die onder de naam "KeyNavigatorBehavior" een soort modus aanbiedt waarin de toeschouwer als het ware door het virtuele universum wandelt.

[3] Meer informatie over deze combinatie is te vinden op `http://java.sun.com/products/jfc/tsc/articles/mixing/`



Figuur 3.6: De camera cirkelt rondom de robot heen

Wie echter niet aan een dergelijk systeem gewend is, loopt makkelijk verloren op die manier, zoals uit enkele experimenten bleek. Door een onverwachtse beweging draait men het zichtveld op zo een wijze dat men niets meer van de robot ziet. Elke voorwaartse beweging die daarop volgt is voldoende om de gebruiker in verwarring te brengen en de weg naar de robot definitief kwijt te laten raken. (Toch is deze camerabesturingsmethode ter illustratie behouden in het programma. Bewegen is mogelijk met de pijltjestoetsen, de camera roteren kan men doen met de pagedown- en pageup-toets[4]).

Om de besturing zo intuïtief mogelijk te houden, is het aantal vrijheidsgraden voor de camera tot drie beperkt:

- om de robot heen cirkelen op constante afstand van de robot (in het XZ-vlak, zie ook figuur 3.6)
- de camera omhoog en omlaag laten gaan op een lijn die evenwijdig is met de Y-as
- de camera bewegen naar de robot toe (een soort van zoomfunctie, in het YZ-vlak)

Deze eenvoudige manier van camerawerk is een gevolg van de *LookAt()*-methode als onderdeel van de *Transform3D*-klasse, die toelaat om de positie van de camera te specificeren, samen met het te bekijken punt. Als men daarna deze transformatie toepast op de camera, wordt de camera automatisch van op zijn huidige positie op het opgegeven punt gericht.

[4]De code is behouden maar voorlopig uitgecommentarieerd. Ten gevolge van een bug in de huidige versie van de Blackdown Java3D-API zorgt het gebruik van KeyNavigatorBehavior voor een overmatig CVE-gebruik (tot 100%). Meer informatie hierover is te vinden in `http://www.blackdown.org/java-linux/java2-status/README-3D121_03`, Utility Bug 4376368.

3.5 Opbouw klassestructuur

3.5.1 De verschillende klassen

3.5.1.1 Cyclop

Dit is de hoofdklasse van het hele programma. De aansturing van de virtuele robot gebeurt op dit niveau. Zoals te zien is in 3.5.2 beschikt deze klasse over de nodige methodes om de verschillende onderdelen van de robot te besturen:

- checkBaseRotate
- checkShoulderRotate
- checkElbowRotate
- checkWristTiltRotate
- checkWristTwistRotate

Al deze methodes hebben als argument een vlottend kommagetal (type: double), dat de hoek aangeeft van de rotatie (in radialen) die doorgevoerd dient te worden. CheckBase-Rotate(Math.PI/2) zal bijvoorbeeld de basis van de robot over een hoek van 90° tegenwijzerszin roteren.

Voor al deze methodes bestaat ook een equivalent zonder het “check”-prefix dat echter niet publiek toegankelijk is. In laatstgenoemde versies wordt immers niet getest of de gewenste hoek zich wel binnen de toegelaten limieten van de robotposities bevindt. Voor een discussie over deze hoeken verwijzen we naar 6.1.

3.5.1.2 CyclopView

Dit is een vrij kleine klasse, die zoals al eerder beschreven, de complexiteit bij het eigenhandig opbouwen van een virtueel universum inkapselt.

3.5.1.3 CyclopGui

Binnen deze klasse bevinden zich alle noodzakelijke gegevens met betrekking tot de opbouw van de grafische gebruikersinterface (Graphical User Interface, GUI).

3.5.1.4 CyclopCommunicationSender

Deze klasse staat in voor het doorzenden van de invoer via de virtuele robot naar de reële robot.

3.5.1.5 CyclopCommunicationReceiver

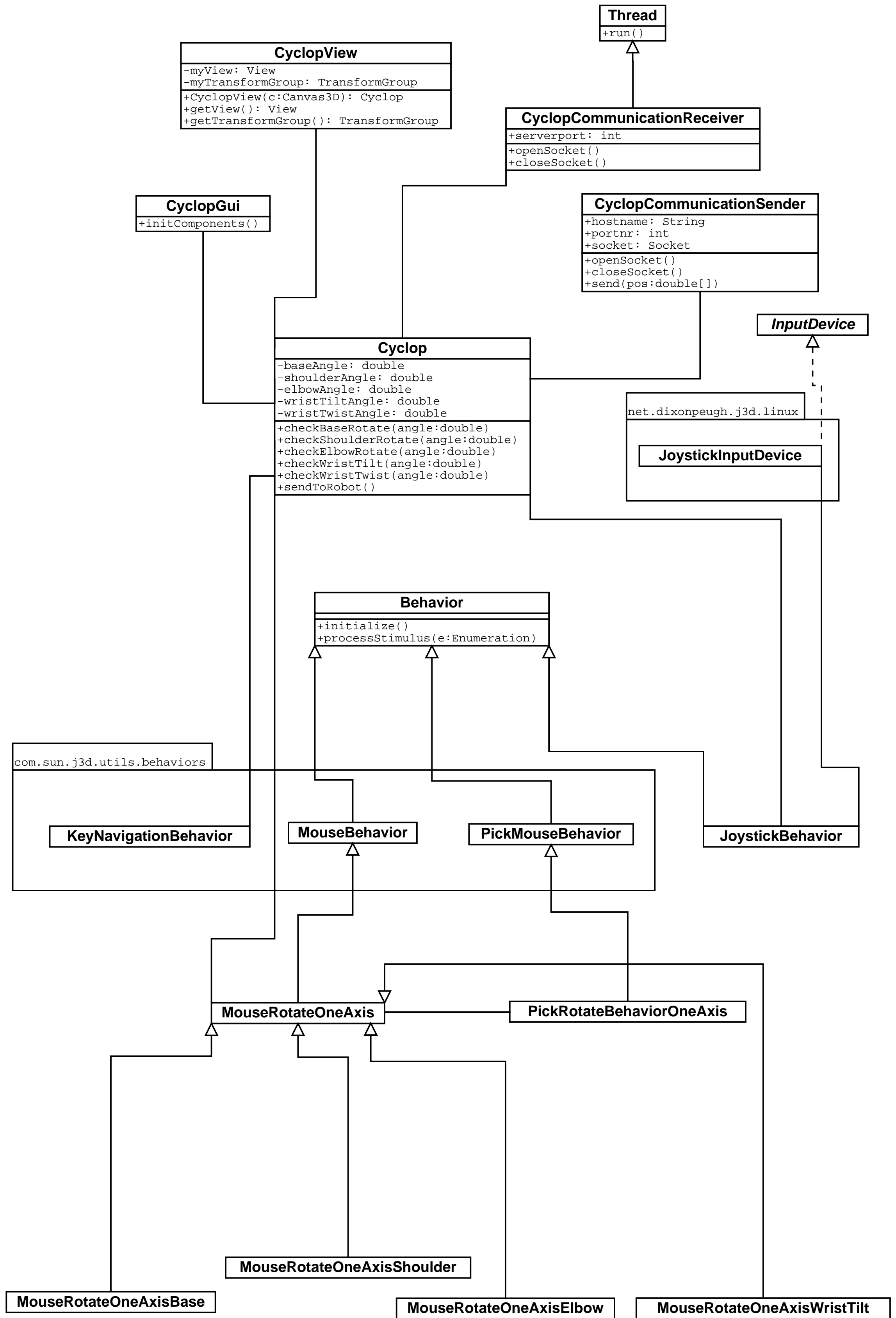
Een object van de CyclopCommunicationReceiver-klasse is voortdurend in leven en wacht op invoer van de echte robot wanneer de virtuele robot geen invoer verwerkt.

3.5.1.6 Gedragklassen

Alle klassen met het affix “Behavior” of “MouseRotate” , voeren een voorgedefinieerd gedrag uit bij bepaalde invoer. Omdat dit nauwer aanleunt bij het onderdeel interactie, worden deze behandeld in het volgende hoofdstuk.

3.5.2 UML-schema

Omwille van de overzichtelijkheid zijn alleen de belangrijkste velden en methodes weergegeven. Aangezien alle verbindingen 1-op-1 relaties zijn, zijn ook de daarvoor bestemde aanduidingen niet opgenomen.


Thread
+run()
CyclopView
-myView: View
-myTransformGroup: TransformGroup
+CyclopView(c:Canvas3D): Cyclop
+getView(): View
+getTransformGroup(): TransformGroup
CyclopCommunicationReceiver
+serverport: int
+openSocket()
+closeSocket()
CyclopGui
+initComponents()
CyclopCommunicationSender
+hostname: String
+portnr: int
+socket: Socket
+openSocket()
+closeSocket()
+send(pos:double[])
InputDevice
Cyclop
-baseAngle: double
-shoulderAngle: double
-elbowAngle: double
-wristTiltAngle: double
-wristTwistAngle: double
+checkBaseRotate(angle:double)
+checkShoulderRotate(angle:double)
+checkElbowRotate(angle:double)
+checkWristTilt(angle:double)
+checkWristTwist(angle:double)
+sendToRobot()
net.dixonpeugh.j3d.linux
JoystickInputDevice
Behavior
+initialize()
+processStimulus(e:Enumeration)
com.sun.j3d.utils.behaviors
KeyNavigationBehavior
MouseBehavior
PickMouseBehavior
JoystickBehavior
MouseRotateOneAxis
PickRotateBehaviorOneAxis
MouseRotateOneAxisShoulder
MouseRotateOneAxisBase
MouseRotateOneAxisElbow
MouseRotateOneAxisWristTilt

3.6 Constructie van de interface

Om de grafische weergave van de virtuele robot en zijn aansturing zo gebruiksvriendelijk mogelijk te maken is het van belang een aangepaste GUI te ontwerpen. Daarbij kwam de Swing API bijzonder goed van pas; deze is op dezelfde filosofie als de rest van de Java API gestoeld en maakt het dus mogelijk om op hoog niveau snel en robuust een grafische gebruiksomgeving te ontwerpen. Snel omdat men zich niet hoeft bezig te houden met de achterliggende code om alle grafische elementen weer te geven, robuust omdat veel weerkerende ontwerppatronen in herbruikbare en makkelijk toegankelijke objecten gegoten zijn.

Met behulp van Swing werden volgende elementen in de GUI geïntegreerd (ook weergegeven in figuur 3.7):

- een Canvas3D met daarop de robot uiteraard
- een kleiner Canvas3D met een bovenaanzicht van de robot
- knoppen om de verschillende robotdelen te bewegen
- 3 schuifknoppen om de camerapositie aan te passen
- een statusbalk met daarop de huidige stand van de hoeken in radialen
- een uitklapmenu

Binnen het laatstgenoemde menu bevinden zich de volgende opties:

3.6.1 Gevoeligheid instellen

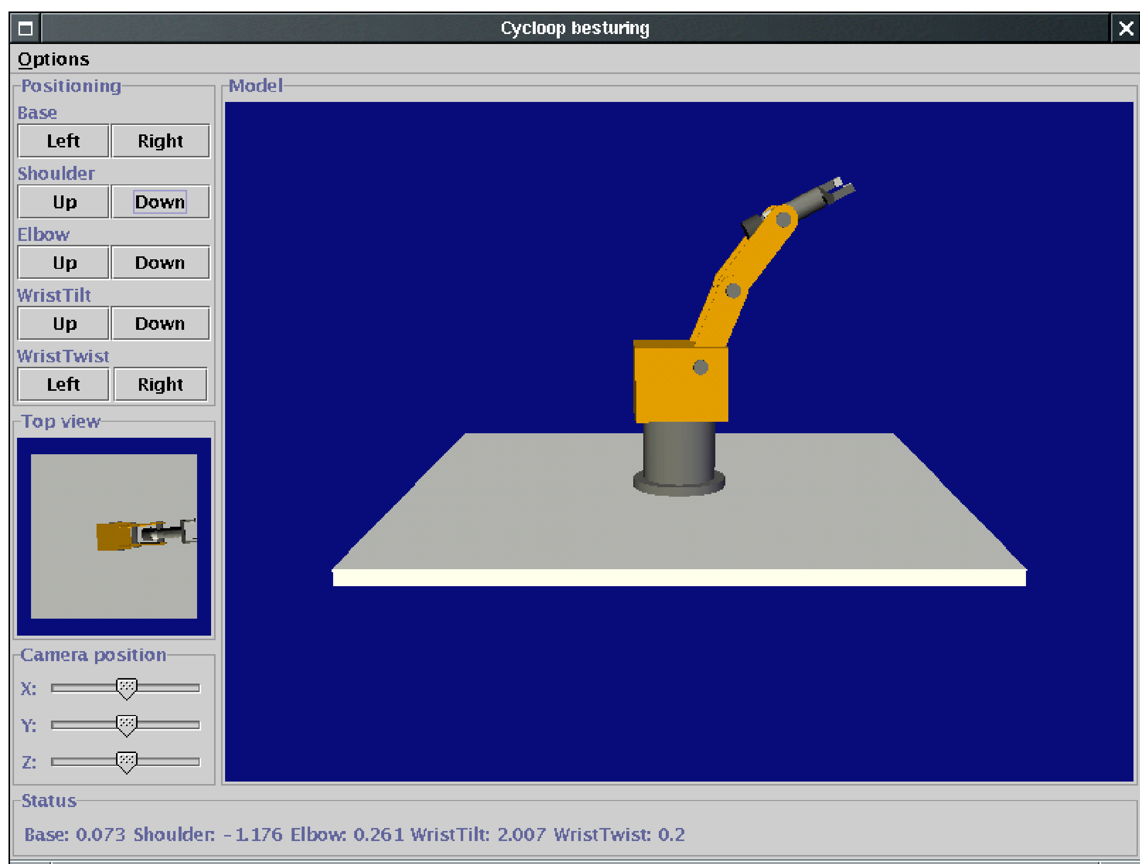
Er komt een dialoogvenster te voorschijn waarin men de grootte van verandering kan ingeven die via de invoermethodes wordt doorgegeven. Op een schaal van 1 tot 10 wordt de gevoeligheid steeds groter, en worden er dus kleinere hoeken voor de rotaties doorgegeven aan de robot.

3.6.2 Simulatie modus

Deze booleaanse waarde geeft aan of de robot al dan niet “intelligente” rotaties zal doorvoeren (zie § 3.3.5 op pagina 24).

3.6.3 Zend invoer naar robot

Als dit aan staat, worden de gegevens die de virtuele robot ontvangt ook direct doorgestuurd naar de echte robot, zoals in een volgend hoofdstuk beschreven wordt.



Figuur 3.7: Schermafdruk van de GUI bij de virtuele robot

3.6.4 Afsluiten

Sluit het programma af en zorgt ervoor dat alle gealloceerde en geopende bronnen gesloten worden.

3.7 Prestatiemeting

3.7.1 Meting zonder optimalisatie

De snelheid van deze applicatie meten is vrij nutteloos: ofwel is de simulatie snel genoeg – en volgt ze de echte robotbewegingen dus zo goed als direct, en kan ze onmiddellijk alle invoer van de gebruiker verwerken – ofwel is ze dat niet. Op alle testplatformen was de waargenomen snelheid bevredigend. Aangezien er zelfs testen gedaan zijn met softwarerendering kan men aannemen dat de huidige implementatie in zo goed als alle omstandigheden snel genoeg zal zijn.

Buiten snelheid is er natuurlijk nog een ander criterium dat men kan nagaan, namelijk bronnengebruik. Wat geheugen betreft ligt dit tussen de 35 en 40 MB, een niet ongewone hoeveelheid voor een hedendaagse 3D-applicatie. Bovendien is de nodige hoeveelheid redelijk constant.

Voor het processorgebruik is er een meting gebeurd op basis van het torens van Hanoi-programma waarbij de virtuele robot alleen voor simulatie instond. Resultaten zijn te vinden in tabel B.1 op pagina 55. De weergegeven resultaten zijn bekomen op de PC die ter beschikking werd gesteld voor het project Cycloop, een AMD Duron van 1 GHz met 1 GB RAM-geheugen. De videokaart was een NVidia GeForce 3D-kaart.

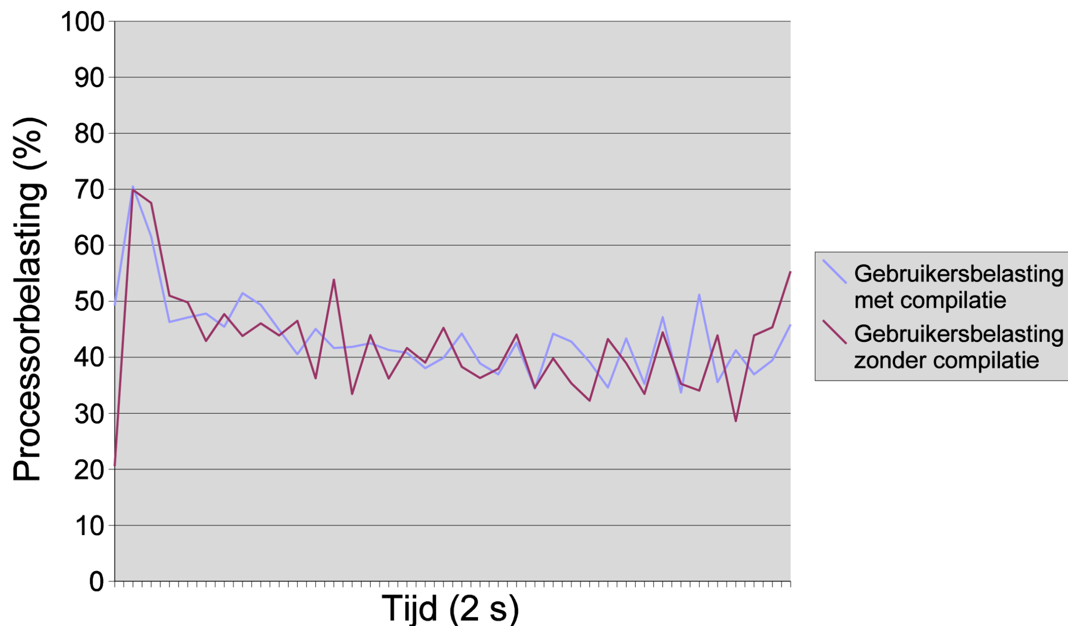
De resultaten blijken meestal onder de 50% processorbelasting te zitten, wat voor een grafisch intensieve toepassing niet echt veel te noemen is. De bewering dat 3D-applicaties onder Java gedoemd zouden zijn om oerverloos traag te zijn, wordt hiermee dan ook duidelijk weerlegd. Het is duidelijk dat het ontwerp gebaseerd op een onderliggende door hardware gesteunde laag van belang is bij het behalen van een dergelijk resultaat.

3.7.2 Meting met optimalisatie

Als men –zoals algemeen aangeraden wordt – de scènegraaf voor de uitvoering laat optimaliseren[5] door Java3D, dan worden onder meer opeenvolgende instanties van TransformGroup vervangen door één TransformGroup. Daarbij gaat het uiteraard om het product van de matrices die met de Transform3D-objecten geassocieerd zijn. Deze manier van werken bespaart veel rekenkracht, maar wordt alleen toegepast als de TransformGroups op alleen-lezen zijn ingesteld. Verder kunnen nog verscheidene optimalisaties doorgevoerd worden – een volledige lijst is te vinden op de webstek van Sun.

De vraag is nu of al deze optimalisaties in de praktijk merkbaar zijn. Om een antwoord hierop te vinden, is er een meting gedaan (opnieuw met de torens van Hanoi) met en

[5] Op voorhand optimaliseren is mogelijk door de compile()-methode aan te roepen van een BranchGroup-object.



Figuur 3.8: Grafiek voor processorbelasting bij de torens van Hanoi

zonder een voorgecompileerde scènegraaf. Elke meting werd dubbel uitgevoerd, en de gemiddelden van de 2 testreeksen werden opgenomen in de resultaattabel in de bijlagen.

In figuur 3.8 staat de grafiek die gebaseerd is op die resultaten en waaruit mag blijken dat voor deze robotsimulatie de optimalisaties geen positief effect hadden op het processorgebruik. Het omgekeerde beweren (dat er een negatieve invloed is dus) zou te voortvarend zijn: deze metingen hebben niet de pretentie om perfect te zijn en bovendien is het gemiddelde gemeten verschil bijzonder klein, nog geen 1.5%. Hoogst waarschijnlijk zal de positieve invloed pas tot uiting komen bij complexere werelden waarin er veel uitgebreidere scènegrafen vervat zijn.

Hoofdstuk 4

Interactie met de simulator

In het vorige hoofdstuk werd een overzicht gegeven van de logische opbouw van de gebruikte klassen binnen de virtuele robottoepassing. Wat nog niet in detail behandeld werd was de gebruikersinteractie. Worden de gangbare interactiemethodes over het algemeen als gebruiksvriendelijk beschouwd? Welke alternatieven bestaan er? Het zijn vragen die in dit hoofdstuk naar voor zullen komen.

4.1 Druktoetsen

4.1.1 Principe

Een bijzonder klassieke manier om een programma aan te sturen is het gebruik van grafische druktoetsen binnen een GUI-omgeving. Dit was dan ook de eerste zaak die geïmplementeerd werd op het gebied van gebruikersinteractie.

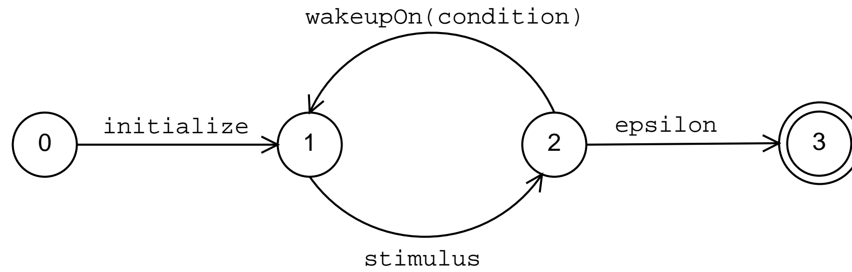
4.1.2 Implementatie

De achterliggende code is vrij rechttoe-rechtaan: een luisteraar geeft bij elke druk op een toets een rotatie door aan een robotdeel.

Deze waarde werd gekozen op 0.1 (nog steeds in radialen), dit geeft geen te grote beeldwijzigingen en is terzelfdertijd merkbaar op het scherm. Dit argument blijft opgaan voor de verschillende gevoeligheidsgraden, waarbij de doorgegeven rotatie gedeeld wordt door een getal van 1 tot 10.

4.2 Slepen-en-plaatsen

Druktoetsen zijn handig voor relatief kleine verplaatsingen, wanneer het echter nodig is om op een vlotte manier een grotere rotatie door te voeren, schiet deze manier tekort. Het is immers niet handig om vele keren na elkaar te klikken.



Figuur 4.1: De Behavior-klasse voorgesteld als deterministische automaat

4.2.1 Principe

Op zulke momenten komt duidelijk de nood naar boven aan een muisgestuurde besturing. Een veelgebruikt paradigma op dat vlak is het slepen-en-verplaatsen (in het Engels drag and drop). Daarbij neemt de gebruiker het te verplaatsen object beet op het scherm met de muis en versleept het daarna naar de gewenste positie ervan. Tijdens het slepen beweegt het object zich mee over het scherm, zodat de gebruiker er voortdurend zicht op heeft waar het zou belanden als hij op een bepaald moment de muistoets los zou laten.

4.2.2 Implementatie

Om slepen-en-plaatsen te gebruiken binnen de context van de Cycloop robotsimulator volstaat de traditionele aanpak van het concept niet, het gaat in casu om een driedimensionale transformatie die moet plaatsvinden in tegenstelling tot de normale tweedimensionale verplaatsing. Om een tot een bruikbaar concept te komen, moeten de tweedimensionale gegevens dus zo omgezet worden dat ze bruikbaar worden in de virtuele wereld.

4.2.2.1 Gedragingen

De invoer binnen het Java3D-platform wordt op een specifieke manier behandeld. Men kan een *Behavior*-object (of één van zijn afstammelingen) instantiëren, waarin men de acties beschrijft die moeten plaatsvinden als reactie op een bepaalde invoer. In een methode die de sprekende naam *processStimulus* meekreeg wordt de code geplaatst waarvan men wil dat ze uitgevoerd wordt als er invoer waargenomen wordt.

Op het einde van de reactiebeschrijving kan men dan opnieuw aangeven op welke conditie er gewacht moet worden, zodat er een voortdurende reactie kan plaatsvinden op een soort stimulus. Conceptueel is dit gedrag terug te brengen op een eindige deterministische automaat, zoals in figuur 4.1 weergegeven.

Vanuit toestand 0 wordt het gedrag geïnitialiseerd. Als vervolgens de stimulus optreedt waar het gedragobject op aan het wachten was, wordt de methode processStimulus uitgevoerd. Dan heeft men de keuze of men opnieuw op dezelfde stimulus wil wachten (door wakeupOn() aan te roepen) of dat men niet langer een bepaald gedrag wenst (en dus het gedragobject laat vernietigen).

4.2.2.2 Robotonderdelen laten roteren

In het utility-pakket dat bij de Java3D API geleverd wordt bevindt zich reeds een systeem dat voor de gestelde eisen vrij geschikt is. Het gaat dan om de *MouseRotate*-klasse, een afgeleide klasse van *MouseBehavior* (die op zijn beurt afstamt van *Behavior*). Dit gedrag leest de muisbewegingen binnen het scherm in en plaatst de rotaties die daar uit zouden voortkomen in een matrix. Om goed te zijn zou deze klasse dus zo omgebouwd moeten worden dat men er de horizontale en verticale bewegingen van de muis mee kan volgen zonder dat deze automatisch omgezet worden in een driedimensionale transformatiematrix.

Enig speurwerk op het internet bleek uit dat een dergelijke klasse reeds bestond, meer bepaald in [Feh00]. De klasse *MouseRotateOneAxis* zorgde samen met *PickRotateBehaviorOneAxis* voor het juiste gedrag. Door overerving werd voor elk specifiek robotonderdeel een gedragklasse gedefinieerd en zo was het mogelijk om dragen-en-slepen snel in de praktijk om te zetten. Het enige gedrag dat niet rechtstreeks met de muis te bewerkstelligen is het roteren rond de eigen as van de pols omdat het versleepbare gebied te klein zou zijn.

Men kan een bepaald gedrag (met een TransformGroup als kind) toevoegen op twee plaatsen binnen de scènegraaf: als kind van een BranchGroup of als kind van zijn eigen kind. De laatste manier van werken biedt bepaalde voordelen bij het instellen van de ruimtelijke grenzen waarbinnen een gedraging zich mag voordoen (zie hiervoor ook [Bou99], hoofdstuk 4). Vandaar dat de gedragingen, zoals in figuur 4.2 weergegeven, ook zo werden toegevoegd aan de scènegraaf.

4.3 Aanraakscherm

4.3.1 Principe

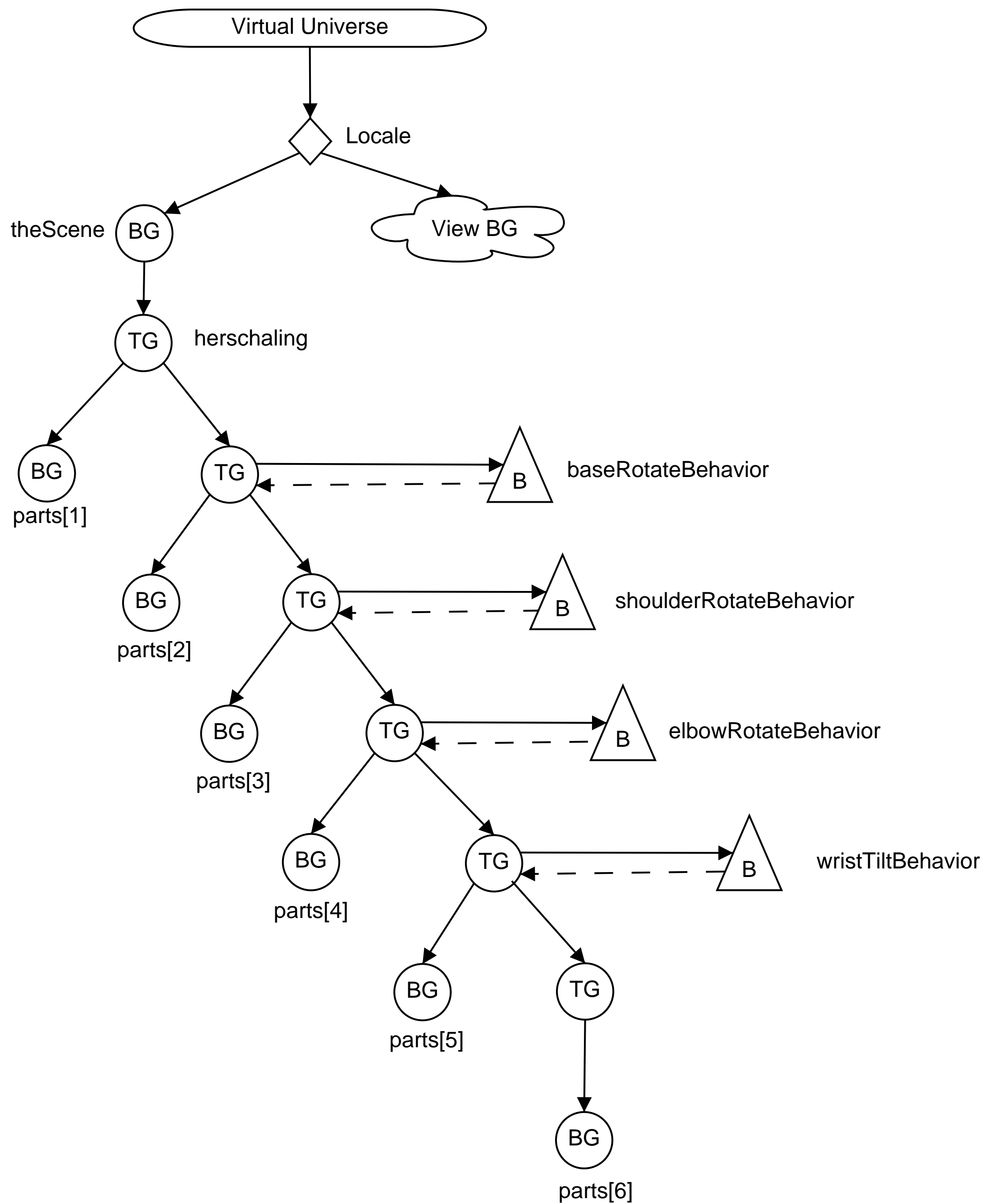
Om het aansturen nog intuïtiever te maken is dit programma ook getest met een aanraakscherm. De gebruiker duidt met zijn vinger een deel aan en bij het bewegen van zijn of haar vinger zal ook dat onderdeel meedraaien.

4.3.2 Implementatie

Aangezien een dergelijk scherm op dezelfde manier als een muis werkt, moesten er softwarematig geen veranderingen doorgevoerd worden. In de praktijk bleek deze invoermanier, zoals verwacht, zeer handig te zijn.

4.4 Joystick

Een belangrijke karakteristiek waarmee Java3D zich wil profileren als virtuele realiteitstaal is de ondersteuning van speciale invoerapparatuur. Daartoe behoren hoofdaan-



Figuur 4.2: Scènegraaf met gedragingen voor slepen-en-plaatsen

sturing (head sets) en gegevenshandschoenen (data gloves). Maar ook minder exotische apparaten, zoals joysticks kunnen gebruikt worden.

4.4.1 Principe

Bij het gebruik van een spelknuppel zou men zich kunnen afvragen of dit wel voldoende vrijheidsgraden ondersteunt om op een eenvoudige manier de robot te laten bewegen. Recente exemplaren beschikken vaak naast de traditionele knuppel over een roerwiel en hebben minstens 4 knoppen, wat de vrijheidsgraden op 5 brengt (als men de knoppen meetelt). Dit aantal komt perfect overeen met de bewegingsmogelijkheden van de robot, als men het voorlopig defecte grijpermechanisme niet meetelt. Het ontwerp van een joystick nodigt uit tot een zeer natuurlijke aansturing zonder een lange leerperiode.

4.4.2 Implementatie

4.4.2.1 De klasse InputDevice

De basisfilosofie van Java indachtig geeft Java3D geen directe toegang tot een specifiek invoerapparaat maar gebeurt dit via een hogere abstractielaag. Om aan invoerresultaten te komen moet men als programmeur een beroep doen op de *InputDevice*-interface. Dit is een soort van virtueel allesoverspannend invoerapparaat dat op een uniforme manier te gebruiken is, om met vaste methodes alle soorten invoer op te vragen.

Als men nu een specifiek invoerapparaat, hier dus een joystick, wil aanspreken moet men een klasse ontwerpen die deze abstracte interface implementeert. Ondanks de relatief jonge leeftijd van de Java3D API zijn er reeds verschillende implementaties beschikbaar voor de meest gangbare platformen en apparaten.

4.4.2.2 Sensoren

Wanneer we een laag lager gaan bekijken binnen het Java3D communicatiesysteem met de randapparaten komen we bij de Sensor-klasse terecht. Dit is een soort van omhulsel (wrapper) voor de communicatie met het besturingssysteem. Analoog bestaat ook een SensorRead-klasse die op haar beurt als omhulseldatastructuur dient voor de invoer van het apparaat.

4.4.2.3 JNI

Elke implementatie van InputDevice is natuurlijk ook verbonden met de onderliggende hardware, met daartussen uiteraard een driver van het besturingssysteem. De brug met het besturingssysteem wordt gelegd met de JNI-technologie. JNI staat voor Java Native Interface en staat voor een verbindingsmethode tussen een Java-programma en een programma dat in een andere taal geschreven is. Hier wordt de JNI-laag aangeroepen om toegang te krijgen tot een C-programmaatje (gecompileerd tot een dynamische gedeelde bibliotheek) dat op zijn beurt een driver van het besturingssysteem aanspreekt.

Voor het ontwikkelingsplatform dat gebruikt werd (Linux/x86 2.4) was er een implementatie van de vernoemde delen beschikbaar onder de naam J3djoystick. Deze door David Dixon-Peugh ontwikkelde driver wordt als vrije software verspreid via `http://sourceforge.net/projects/j3djoystick`.

4.4.2.4 Aanpassingen aan J3djoystick

Bij het uittesten van de joystick bleken er nog een paar zaken de mist in te gaan. Zo moesten er aanpassingen gedaan worden aan de nummers van de assen omdat de originele code ontworpen werd voor een logitec Cyberman2 en ik een standaard 3-assen en 4-knoppen model gebruikte.

Dit bleek echter nog niet voldoende te zijn: ook de knoppen weigerden alle dienst, als waarde werd steeds 0 (i.p.v. 1) teruggegeven bij activatie ervan. Na een bijzonder intensieve zoektocht bleek uiteindelijk dat de waarde ervan eenvoudigweg niet werd opgeslagen in het SensorRead-object, waarschijnlijk omdat de auteur van J3djoystick geen knoppen gebruikte bij de aansturing van zijn programma. Na aanpassing van dit deel werkte alles naar behoren.

4.4.2.5 Testen van joystick I/O

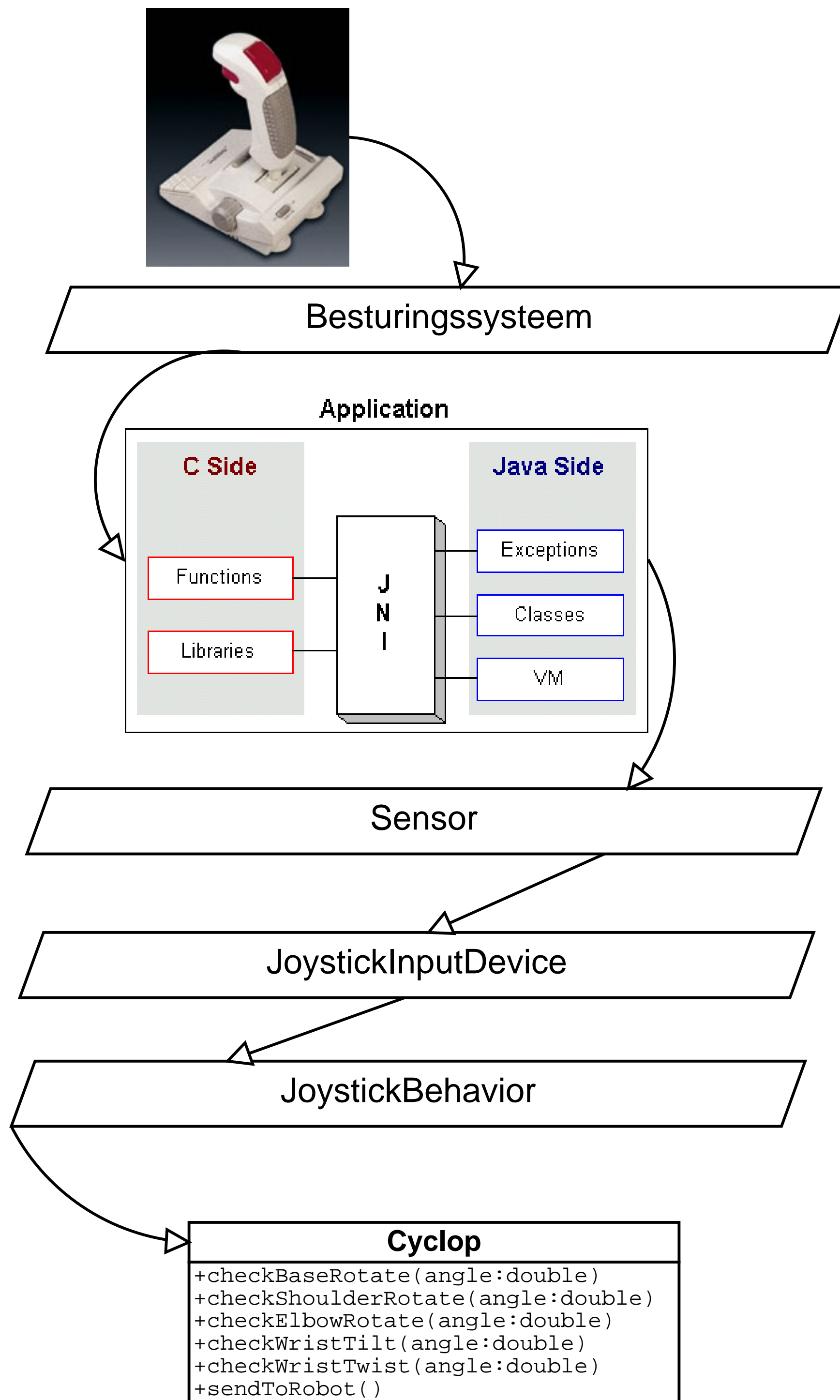
Toch kwamen er nog problemen naar boven bij het gebruik van de joystick als invoermiddel. Het processorgebruik bleek op te lopen tot 100%, zelfs als er niet aan de joystick geraakt werd. Bij het controleren van de instellingen bleken er twee zaken verkeerd geconfigureerd te zijn.

Ten eerste was stond de InputDevice ingesteld op niet-blokkeren, de oplossing hiervoor was de vraaggedreven modus (DEMAND_DRIVEN) te kiezen.

Een tweede probleem bleek te liggen bij de condities waarop het gedrag moest in werking treden. Aangezien er geen ondersteuning is voor een soort “ontwaak bij invoer”-voorwaarde, moest er een beroep gedaan worden op een activering wanneer een bepaald aantal keren het beeld ververst is *(WakeUpOnEllapsedFrames*). Dit was echter te rekenkrachtintensief en daarom werd er overgeschakeld op activatie wanneer een bepaalde tijdsduur verstreken is (i.c. 50 ms) door middel van *WakeUpOnEllapsedTime*. Dit was voldoende om het processorgebruik terug naar 0% te brengen wanneer het programma actief is en er geen invoer is.

4.4.2.6 Verwerking van de joystickinvoer

In de gedragsspecificatie van het joystickobject worden er bij een bepaalde invoer de bewegingsmethodes van de robot aangeroepen. De invoer van de verschillende assen bestaat uit een waarde uit het interval $[-1, 1]$, wanneer de absolute waarde groter is dan 0.5 wordt er een signaal doorgegeven dat de robot dient te bewegen. De knoppen, die als booleaanse waarde ontvangen worden, geven bij een 1-signaal een dergelijke reactie. Een overzicht van de relaties tussen de invoermogelijkheden van de joystick en de bestuurbare onderdelen van de robot is in tabel 4.1 weergegeven.



Figuur 4.3: Schematische doorstroming van gegevens bij invoer via de joystick

Jostick-invoer	Verantwoordelijk voor
X-as	basis roteren
Y-as	schouder roteren
Z-as	elleboog roteren
knop 1 en 2	pols kantelen
knop 3 en 4	pols roteren

Tabel 4.1: Invoerverwerking van de joystick

Hoofdstuk 5

Simulator-robot verbinding

In de voorgaande hoofdstukken werd de constructie van de virtuele robot belicht. Communicatiemogelijkheden met de echte robot zijn er echter nog niet en dat vormt dan ook meteen het onderwerp van dit hoofdstuk. Tenslotte zou het nut van deze toepassing ver te zoeken zijn mocht er geen verbinding met de echte robot gelegd kunnen worden.

5.1 Doel van de communicatie

De doelstelling van de communicatiemodule is tweezijdig.

5.1.1 De robot afbeelden

Langs de ene kant moet het mogelijk zijn om te zien in welke positie de echte robot zich bevindt als er een programma uitgevoerd wordt op de controllersturende computer. Een voorbeeld hiervan is de traditionele demonstratie van de torens van Hanoi. Een Java-programma met instructies voor de robot loopt op de computer die ook de controller aanstuurt en positie van de robot wordt voortdurend via het netwerk doorgestuurd naar de virtuele robot die de werking van de echte robot nabootst.

5.1.2 Een invoermethode vormen voor online besturing

Daarnaast moet het ook mogelijk zijn dat de gebruiker de virtuele robot op een bepaalde manier manipuleert en dat de echte robot automatisch de positie van de virtuele robot aanneemt. Als de gebruiker de virtuele robot met de joystick een hoek van 90° laat draaien, moet ook de echte robot dezelfde rotatie uitvoeren, en liefst zo snel mogelijk.

5.2 Keuze van communicatieprogrammatuur

Eén van de doelstellingen van het project Cycloop was dat de bewakingscamera via het internet toegankelijk zou zijn en dat de besturing ervan dus van op afstand zou kunnen ge-

beuren. Aangezien ook de robotcontroller via een USB-interface onder Java aangestuurd wordt, lag het voor de hand om een oplossing te zoeken die op Java gebaseerd is.

5.2.1 Remote Method Invocation

Voor communicatie tussen meerdere Java-programma's die niet op dezelfde computer uitgevoerd worden bestaat er een vrij uitgebreid pakket dat standaard bij de Java API geleverd wordt, met name RMI (Remote Method Invocation). RMI staat volledig in voor een transparante serialisering van de te transporteren objecten en het doorzenden ervan. Al bij al is RMI de beste manier om gedistribueerde Java-applicaties te schrijven op een hoog niveau.

De realiteit verhinderde echter het gebruik ervan. Het is namelijk de bedoeling dat de software waarmee de robotcontroller bestuurd wordt draait op een Java Virtuele Machine die aangepast is om beter te presteren bij waretijdstoessingen. Han Weyn zorgde voor een dergelijke JVM (zie ook [Wey02]). Jammer genoeg ondersteunt de daarbij gebruikte JVM geen RMI, zodat het noodzakelijk was andere technieken aan te wenden.

5.2.2 Sockets

Bij gebrek aan RMI werd het noodzakelijk om de oplossing op een lager niveau te gaan implementeren. Concreet betekende dit dat communicatie rechtstreeks via sockets zou verlopen, een veelgebruikte techniek binnen het TCP/IP-kader. Op dezelfde manier als een file geschreven en gelezen wordt, wordt er geschreven naar en gelezen uit een socket.

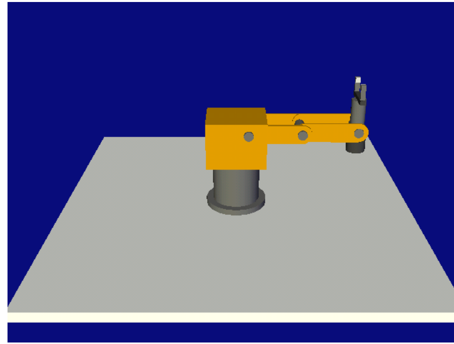
Verder wordt er gewerkt volgens het client-server principe: er is één server socket die voortdurend wacht op een aanvraag van een client socket. Is er een aanvraag tot communicatie van een client socket, dan wordt de communicatielijn geopend en kan de het clientprogramma gegevens doorsturen naar de server, die daar op zijn beurt eventueel kan op reageren.

De programmeur hoeft zich bij deze applicaties omwille van de betrouwbaarheid van het onderliggende TCP-protocol niet al te zeer te bekommeren over fouten die zouden optreden gedurende de communicatie. Daar de JVM die gebruikt werd om de robotcontroller aan te sturen sockets ondersteunde, lag de keuze voor sockets voor de hand.

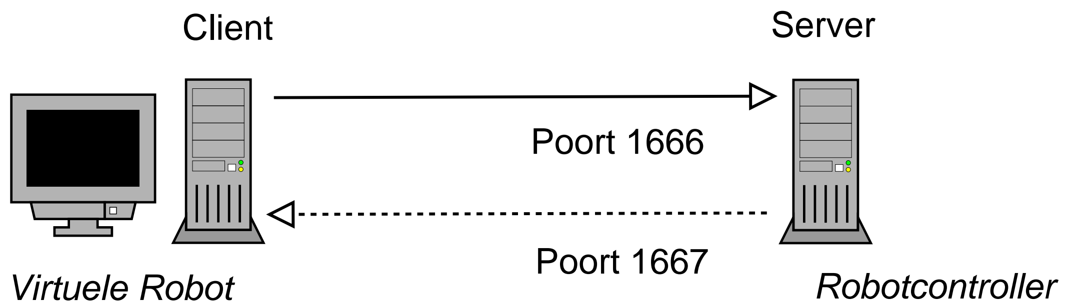
5.3 Concrete implementatie

5.3.1 Communicatietechniek

Om goed te communiceren is het natuurlijk noodzakelijk dat er een gemeenschappelijke grootte-eenheid gebruikt wordt. In casu is dit de hoeken van de verschillende gewrichten of simpelweg de gewrichtscoördinaten. Voor elk van de gewrichten houdt dit de hoek in met het daaropvolgende gewricht – een relatieve hoek dus en niet de hoek die gemaakt wordt met het grondvlak. De beginpositie (de positie waarin alle hoeken op 0 staan) is de positie waarin de robotarm volledig horizontaal staat en de pols een hoek van 90° daarmee maakt, zoals weergegeven in figuur 5.1.



Figuur 5.1: Beginpositie van de robot



Figuur 5.2: Uitwisseling van gewrichtscoördinaten via sockets

De beschrijving van de onderdelen in VRML zijn handmatig aangepast zodat ze zich bij het inladen automatisch in dezelfde positie bevinden als de startpositie. Dit was oorspronkelijk niet het geval – er werd gewerkt met een soort initiële ijkingsprocedure maar die manier van werken ging de mist in bij de Hanoi-test, vandaar de aanpassing.

De gemeenschappelijke gewrichtscoördinaten worden uitgewisseld op volgende manier:

- De invoer van de virtuele robot wordt doorgestuurd naar de robotcontroller waar er op poort 1666 geluisterd wordt.
- Als de robot op het scherm alleen ter simulatie dient luistert hij op poort 1667 naar inkomende gewrichtscoördinaten.

Het "luisteren" wordt verricht door een aparte draad die blokkeert (en dus geen onnodige bronnen verbruikt) als er geen invoer binnenstroomt via het netwerk. In figuur 5.2 wordt een schematisch overzicht gegeven van de communicatie tussen de virtuele en de echte robot.

5.3.2 Wisselwerking met de robotcontroller

Wat gebeurt er nu allemaal aan de zijde van de robotcontroller? Bij uitvoer van de gewrichtscoördinaten is dit vrij eenvoudig: er wordt elke 40 ms een array doorgestuurd met

daarin alle hoeken en de grijpertoestand (dit laatste voor toekomstig gebruik). 40 ms is hierbij geen willekeurige keuze: het gevolg is dat er net iets meer dan 24 veranderingen per seconde optreden, wat de norm is voor filmbeelden. Zo zullen er normaal gezien geen schokken optreden in het "afspelen" van de invoer.

Het omgekeerde principe is ongeveer van toepassing bij het doorsturen van invoer naar de robotcontroller. Dan wordt er een array van gewenste hoekcoördinaten ontvangen en zal de robot die positie aannemen. Bij het testen bleek hier nog een vervelend effect aanwezig te zijn. Bij het doorsturen van ruime bewegingen bleek de robot in vele kleine bewegingen te reageren.

Zoals in [Ven02] beschreven is, is het stuurprogramma van de robotcontroller gebaseerd op een PID-algoritme dat voortdurend tussen een beginpunt en eindpunt tussencoördinaten interpoleert die de weg beschrijven die de robot zal volgen. Dit verklaarde het vervelende gedrag: de robot kreeg voortdurend kleine verschillen doorgestuurd. Elke grote hoek is bij de robotsimulatie samengesteld uit vele kleine hoeken om tot een vloeiend beeld te komen. Bij het controleprogramma geeft dit echter het omgekeerde effect, door voortdurend interpolaties te maken tussen kleine verschillen krijgt men een patroon van kleine schokkende bewegingen.

Ook hiervoor bleek een vrij praktische oplossing te bestaan: als de invoer niet komt van een drukknop op het scherm (en er dus een kleine beweging zal plaats vinden) worden er pas om de 100 ms gewrichtscoördinaten doorgestuurd. Het gevolg ervan is dat minder interpolaties worden gemaakt en er dus grotere tussenstappen genomen worden door de robot, wat een gunstig gevolg heeft voor de snelheid en vlotheid van reageren op de invoer. Een verdere discussie over de communicatie met de robotcontroller en het interpolatiealgoritme is uiteraard te vinden in [Ven02].

5.3.3 Verschillende communicatiemodi

Men kan de communicatie op 2 manieren laten lopen:

1. Simulatiemodus: dit is de standaardsituatie bij het opstarten van de virtuele robot. Alle bewegingen die de echte robot uitvoert worden geïmiteerd.

2. Invoermodus: als het vakje "zend invoer naar robot" aangevinkt is, zullen alle bewegingen die men de virtuele robot laat maken ook door de reële robot uitgevoerd worden.

In het laatstvernoemde geval zal een ingevoerde beweging direct zichtbaar zijn op het scherm en dus niet eerst uitgevoerd worden en dan pas zoals in het eerste geval doorgestuurd worden naar de virtuele robot.

Dit is een ontwerpkeuze die men moet maken tussen een zo snel mogelijke reactie van de virtuele of de reële robot. Het probleem is echter dat als men de echte robot eerst de bewegingen laat maken en dan pas het model in actie laat treden dat het onmogelijk wordt om een precieze invoer te geven.

Stel dat men bijvoorbeeld een hoek van 120° wil laten maken, dan is het onmogelijk te weten hoe lang men de aansturende joystick dient te bewegen vooraleer de robot ter

plaatse zal zijn, genomen dat men geen zicht heeft op de echte robot. Dit wordt nog duidelijker bij het verslepen van de robot op het scherm: dit zou totaal onmogelijk worden. Vandaar dat gekozen is om de echte robot pas na het virtuele exemplaar te sturen.

5.4 Bedenkingen bij vertraging

Eén van de mogelijke kritiekpunten van de hier beschreven communicatiemethode is de vertraging die optreedt bij het netwerktransport van de gegevens. Zal dit in de praktijk voor problemen zorgen?

Een kleine praktijktest wees uit van niet. Op een lokaal netwerk duurde het gemiddeld een honderdtal μs om gegevens door te sturen. Bij een nationaal WAN werd dit een cijfer in de orde van enkele tientallen ms, een transatlantische verbinding gaf enkele honderden ms. In geen enkel van deze gevallen zal de vertraging dus de goede werking van de applicatie in het gedrang brengen.

5.5 Bedenkingen bij beveiliging

Uiteraard is het niet de bedoeling dat iedereen zo maar de robot kan aansturen. Een adequate beveiliging kan eventueel voorzien worden door de robotcontroller achter een firewall te plaatsen en alle communicatie met de virtuele robot te laten "tunnelen" over een versleutelde verbinding met bijvoorbeeld secure shell. Op die manier dient er geen aparte beveiligingsmodule geschreven te worden en is men toch zeker van een goede beveiliging.

Hoofdstuk 6

Toekomstperspectieven

Uiteraard is deze scriptie geen definitief einde voor het project Cycloop. Er zijn nog verschillende zaken waaraan verbeteringen mogelijk zijn en ook de uitbreidingsmogelijkheden zijn nog niet uitgeput. Dit hoofdstuk probeert een overzicht te geven van interessante uitbreidingen waarvoor het ontbrak aan tijd en know-how om te implementeren.

6.1 Botsingen

6.1.1 Principe

Een punt dat tot nu toe buiten beschouwing is gelaten, is botsingsverwerking. Het principe erachter is vrij eenvoudig: als 2 verschillende objecten zich deels op dezelfde plaats in de ruimte bevinden, is er sprake van een botsing. Als we de robot bekijken als een samenstelling van verschillende bewegende delen is er dus ook zoiets mogelijk als een inwendige botsing van de robot. Een robotarm kan bijvoorbeeld in aanraking komen met het basisgedeelte waarop de robot steunt en zo duidelijk schade toebrengen.

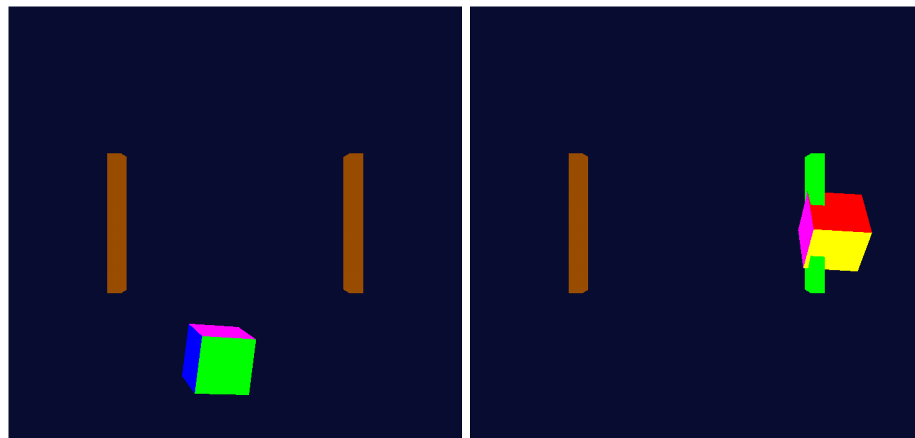
Een programma kan op verschillende manieren reageren op botsingen.

6.1.1.1 Negeren

De eenvoudigste reactie is voor de hand liggend: geen reactie. Botsingen worden toegestaan. Het spreekt voor zich dat dit scenario voor ernstige problemen kan zorgen, in de praktijk zal er waarschijnlijk schade toegebracht worden aan de botsende lichamen.

6.1.1.2 Botsingsdetectie

Een betere oplossing dan negatie is het automatisch detecteren van de botsingen. Als er zich een botsing voordoet, dan verandert het object bijvoorbeeld van kleur of wordt er een procedure opgeroepen. Deze techniek is vooral gangbaar bij het off-line programmeren van robots, waar getest moet worden of een programma geen gevaarlijke bewegingen zal genereren. Dan kan men na het opmerken van de botsing het programma zo aanpassen dat er een botsingsvrij pad afgelegd wordt door de robot.



(a) geen botsing

(b) de botsing wordt gemarkeerd door een kleurverandering

Figuur 6.1: Botsingsdetectie bij een bewegende kubus

6.1.1.3 Botsingspreventie

Een verdere logische stap is het proberen vermijden van botsingen. Daarbij wordt elke beweging die een robot zal maken op voorhand gecontroleerd op mogelijke botsingen die erop zouden kunnen volgen. Als een dergelijke botsing wordt voorspeld, wordt er een alternatieve gepland of stoppen alle bewegingen. Conceptueel valt botsingspreventie te beschouwen als een vorm van botsingsdetectie met een uitgebreide botsingszone – een soort buffer dus – waarin de objecten niet daadwerkelijk botsen maar wel in een gevarenzone komen en een verdere toenadering niet aangewezen is. Men kan dus stellen dat als er botsingsdetectie is, er ook aan botsingspreventie gedaan kan worden.

In on-line besturing van een robot is preventie de enige mogelijke optie, het zou immers onverantwoordelijk zijn om de mogelijkheid tot botsingen niet volledig uit te sluiten. Niet alleen de hoge kosten voor een robot spelen dan, nog veel meer zijn er dikwijls mensenlevens in het spel. Een botsing met een levend wezen moet uiteraard te allen prijze vermeden worden.

6.1.2 Implementatie

Een voor de hand liggende verwachting zou dus kunnen zijn dat er een vorm van botsingspreventie wordt gebruikt bij de virtuele robot teneinde botsingen met zichzelf of het steunplatform te vermijden. Aan de kant van de robotcontroller is er een dergelijk mechanisme, als er dus “schadelijke” gewrichtscoördinaten worden ontvangen door het Java-gestuurde robotcontollerprogramma wordt er een uitzondering opgeworpen en zal de robot stilvallen.

6.1.2.1 Botsingsdetectie in Java3D

Gegeven het belang van botsingsdetectie is het niet onlogisch dat er in de Java3D API voorzieningen zijn om dit te bewerkstelligen. Een gedrag dat geactiveerd wordt bij botsingen is de manier van werken die hiervoor gebruikt wordt. Helaas is op dit gebied de theorie mooier dan de praktijk.

Een kleine test met één van de voorbeeldprogramma's die bij Java3D geleverd worden (i.e. *TickTockCollision*) wees dit uit. Het programma voert botsingsdetectie uit bij een eenvoudige scène zoals die in figuur 6.1 getoond wordt, een kubus volgt een cirkelvormige baan en komt daarbij om beurten in botsing met 1 van de 2 aanwezige balken. Bij een botsing verkleurt de balk waarmee gebotst wordt. Hoewel dit zeker geen complex virtueel universum is, liep het processorgebruik bij dit voorbeeld toch voortdurend op tot 100%, wat uiteraard niet aanvaardbaar is.

Er zijn nog meer nadelen verbonden aan de Java3D API op dit gebied. Zo is de precisie soms ver te zoeken en kan het zijn dat door uitwendige oorzaken bij hetzelfde tafereel er nu eens een botsing wordt gedetecteerd en dan weer niet. Verdere bezwaren tegen het gebruik van de ingebouwde botsingsdetectie in Java3D zijn onder meer te vinden in [Feh00].

Al bij al werd het duidelijk dat deze manier van werken niet ideaal zou zijn.

6.1.2.2 Zelf botsingen detecteren

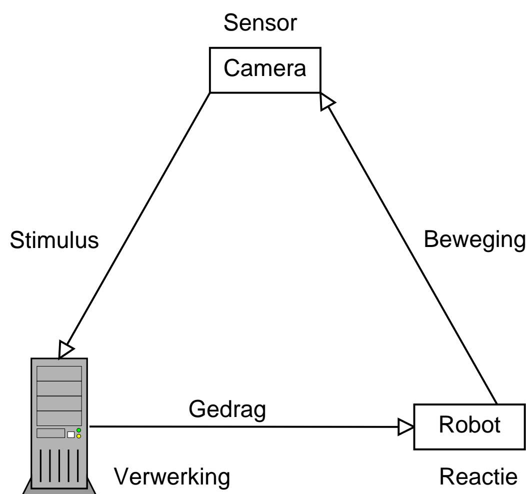
Als men zich niet kan beroepen op de ingebouwde detectiemogelijkheden van de API zit er weinig anders op dan zelf een implementatie te schrijven die dit wel goede resultaten biedt.

Alhoewel dit op het eerste gezicht misschien geen moeilijke opgave lijkt, bleek er toch heel wat werk bij te komen kijken. Veel gebruikte en kwalitatieve oplossingen zijn dikwijls gebaseerd op PID-algoritmes, wat dit onderwerp dus nauwer laat aansluiten bij de aansturing van de robotcontroller die eveneens PID-gebaseerd is. Deze methode is dan ook minder geschikt voor de virtuele robot omdat deze niet PID-gestuurd is.

Er bestaan pogingen om een alternatief pakket te schrijven dat botsingsdetectie uitvoert, onder meer [Cou01], maar wegens tijdsgebrek heb ik niet meer de mogelijkheid gehad om dit aan te passen en in het geheel van de virtuele robot te integreren. Het spreekt voor zich dat een dergelijke aanpassing in de toekomst een welkome uitbreiding zou vormen voor de huidige virtuele robot.

6.2 Beeldverwerking

Tot de doelstellingen van het Cycloopproject hoort eveneens de koppeling van een camera aan de robot. Het integreren van het ontvangen van de beelden met de hier uitgewerkte applicatie zal op zich waarschijnlijk geen grote problemen stellen. Het systeem programmeren met het doel automatisch voorwerpen te volgen zal waarschijnlijk een grotere inspanning vereisen.



Figuur 6.2: Perceptiecyclus van een visueel gestuurde robot

Het is goed om weten dat Java3D het ideale platform vormt om de robot aan te sturen op basis van visuele signalen. Dat is voor een groot deel toe te schrijven aan de Behavior-interface, die de mogelijkheid biedt om op een zeer natuurlijke wijze het (i.c. visuele) verwerkingsproces te bekijken. Men kan namelijk volledig werken in analogie met de prikkelverwerking van organismen.

Eerst en vooral wordt een beeld waargenomen door een zichtorgaan, wat dus de camera zal zijn. De perceptie kan echter pas plaats vinden als de beelden de verwerkende computer bereiken.

Aan de gebruikte sensor (de camera dus) kan vervolgens een bepaald gedrag toegeschreven worden door middel van de Behavior klasse. Als er bijvoorbeeld een beweging naar rechts wordt gedetecteerd kan er als stimulusverwerking een beweging van de robot (en dus de daaraan bevestigde camera) in dezelfde richting uitgevoerd worden. Dit principe wordt nog eens grafisch weergegeven in figuur 6.2.

Het handige aan de Java bibliotheek is dat het eigenlijke verwerken niet op dezelfde manier hoeft te gebeuren als de reacties die het teweeg brengt. Voor bijzonder tijdskritische applicaties zoals beeldperceptie is het best mogelijk dat het gebruik van een andere taal (C of zelfs machinetaal) taak aangewezen is. Bij een dergelijke noodzaak is het dan mogelijk om de JNI-verbindingslaag zo in te schakelen dat de verwerking van de visuele invoer gebeurt in een andere taal en het toch mogelijk blijft om het zeer elegante systeem van gedragingen te benutten in Java. Men scheidt dan als het ware de hersenen van de intelligente bewakingsrobot van de rest van het lichaam en de zenuwcellen. Op die wijze komt men via een zeer intuïtief paradigma tot het uiteindelijke doel van dit project: een intelligent bewakingssysteem dat die naam waardig is.

Bijlage A

Broncode op CD-ROM

Zie bijgevoegde CD-ROM voor:

- de volledige broncode
- JavaDoc-documentatie
- een overzicht van interessante webpagina's
- de Java3D API
- het VRML-loader pakket
- joystickbibliotheken voor Linux en Windows

Alle code die ontwikkeld is voor de virtuele robot is vrijgegeven onder de GNU GPL-licentie. Meer informatie hierover is te vinden op de CD-ROM.

Bijlage B

Metingen van processorgebruik

Volgende tabel geeft de “User load” weer gedurende het uitvoeren van de Torens van Hanoi-demonstratie. Hierbij speelt de virtuele robot alleen na wat hij als invoer van de echte robot krijgt. De cijfers werden bekomen door om de 2 seconden de resultaten van top weg te schrijven en zijn uiteraard uitgedrukt in %.

tijdstip (2 s)	Met optimalisatie	Zonder optimalisatie
0	49.25	20.55
1	70.5	69.9
2	61.55	67.55
3	46.3	51
4	47.1	49.8
5	47.8	42.9
6	45.45	47.7
7	51.45	43.8
8	49.35	46.05
9	44.9	43.9
10	40.55	46.5
11	45.05	36.25
12	41.65	53.85
13	41.85	33.45
14	42.5	43.95
15	41.3	36.2
16	40.8	41.65
17	38.05	39.05
18	39.9	45.25
19	44.25	38.3
20	38.9	36.3
21	36.95	37.95
22	42.6	44.05
23	34.4	34.55
24	44.2	39.8
25	42.8	35.35
26	39.15	32.25
27	34.6	43.25
28	43.35	38.95
29	35.25	33.45
30	47.15	44.45
31	33.7	35.25
32	51.15	34.05
33	35.55	43.9
34	41.25	28.6
35	36.95	43.9
36	39.45	45.35
37	45.85	55.35
gemiddelde	44.67	43.36

Tabel B.1: Metingen van processorgebruik met en zonder optimalisaties

Bibliografie

[ASZ+02] P. Abolmaesumi, S. Salcudaen, W. Zhu, M. Sirouspour, and S. DiMaio. Image-guided control of a robot for medical ultrasound. *IEEE Transactions on robotics and automaton*, 18(1):11–20, february 2002.

[Bou97] Vassilis Bourdakis. The future of vrml on large urban models. 1997. http://fos.prd.uth.gr/vas/papers/UKVRSIG97/.

[Bou99] Dennis J. Bouvier. *Getting started with the Java3D API*. Sun Microsystems and K Computing, version 1.5 (java3d api v. 1.1.2) edition, 1999. http://java.sun.com/products/java-media/3D/collateral/.

[CM01] Tom Christiaens and Filip Mechant. Ontwerp van een usb-interface voor een robotcontroller. Master's thesis, Universiteit Gent, 2001.

[Coo01] Prof. dr. K. Coolsaet. *Programmeren van GUI's met Java Swing*. Universiteit Gent, Vakgroep toegepaste wiskunde en informatica, 2001.

[Cou01] Justin Couch. Implementing terrain following and collision detection in java 3d. Technical report, J3d.org, 2001. http://www.j3d.org/tutorials/collision/index.html.

[Far96] Rik Farrow. Networking made easy. *;login:*, 21(6), December 1996.

[Feh00] Andreas Fehlmann. Furnishing planner. Master's thesis, Hochschule für Technik und Architektur Biel, 2000. http://cobra.hta-bi.bfh.ch/Home/swc/DemoJ3D/FurnishingPlanner.

[FR99] Eckhard Freund and Jürgen Rossmann. Projective virtual reality: bridging the gap between virtual reality and robotics. *IEEE Transactions on robotics and automaton*, 15(3):411–422, june 1999. http://www.irf.de/cosimir/vr/ProjektiveVR/ProjektiveVR.htm.

[KBC+01] O. Khatib, O. Brock, K.C. Chang, D. Ruspini, L. Sentis, and S. Viji. Human-centered robotics and interactive haptic simulation. Stanford University, 2001.

[Pel01] Stéphane Peltot. 3d simulatie van een robotbesturing. Master's thesis, Universiteit Gent, 2001.

[Poo89] Harry H. Poole. *Fundamentals of robotics engineering*. Van Nostrand Reinhold, 1989.

[RDCP$^+$96] H.L. Roediger, E. Deutsch Capaldi, Scott G. Paris, J. Polivy, and C. Peter Herman. *Psychologie, een inleiding*, chapter 1, pages 6–7. Academia Press, 1996.

[SECW99] N. Smith, C. Egert, E. Cuddihy, and D. Walters. Implementing virtual robots in java3d using a subsumption architecture. *Proceedings from the Association for the Advancement of Computing in Education*, pages 975–980, 1999.

[Ser99] Prof. dr. Herman Serras. *Inleiding tot de grafische technieken*. Universiteit Gent, Vakgroep wiskundige analyse, 1999.

[VDV01] Nick Vercammen and Gwijde De Vocht. Real-time robotbesturing in java. Master's thesis, Universiteit Gent, 2001.

[Ven02] Kris Venstermans. Realtime robotaansturing in java. Master's thesis, Universiteit Gent, 2002.

[Wey02] Han Weyn. Studie en implementatie van geheugensanering voor de jvm. Master's thesis, Universiteit Gent, 2001-2002.