TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Wiskunde en Informatica

AFSTUDEERVERSLAG

Animatie in Virtue

Door

J.C.L. Geerlings

Afstudeerdocent:	dr. ir. H.M.M. van de Wetering
Afstudeercommissie:	dr. ir. J.J. van Wijk
	dr. A.T.M. Aerts

Oktober 2001

Voorwoord

Dit is het verslag van mijn afstudeeropdracht ter afsluiting van mijn studie Technische Informatica aan de Technische Universiteit Eindhoven. Deze opdracht is uitgevoerd in samenwerking met de bedrijven Cebra en PhilemonWorks. De opdracht bestond voornamelijk uit het uitbreiden van een bestaande scripttaal voor het bouwen van 3D werelden, SmallScript3D, met mogelijkheden voor beweging en dynamische scènes, botsingsdetectie en met mogelijkheden tot het specificeren van interactie.

Graag wil ik van deze gelegenheid gebruik maken om een aantal mensen te bedanken.

Allereerst zijn dat de mensen die me vanuit de bedrijven geholpen hebben: Ernest Micklei van PhilemonWorks, die voor ondersteuning van Smalltalk en SmallScript3D gezorgd heeft, en Bas Vermeer van Cebra, die me gedurende het hele project geholpen, gesteund en gestimuleerd heeft.

Verder wil ik Bart Barenbrug en Gino van den Bergen bedanken voor hun ondersteuning bij het gebruik van DynaMo en SOLID, en voor hun hulp bij het oplossen van de problemen die ontstonden bij het combineren van SmallScript3D, DynaMo en SOLID. Ook wil ik Kees Huizing bedanken voor zijn hulp bij het maken van het Perl-script voor het inlezen van de headers van DynaMo.

Mijn ouders, mijn broers en mijn zus wil ik bedanken voor de morele ondersteuning en voor het geduld dat ze met me gehad hebben als er weer eens iets misgelopen was.

Tenslotte wil ik mijn afstudeerdocent, Huub van de Wetering, heel hartelijk bedanken voor zijn hulp en ondersteuning bij vrijwel alle onderdelen van dit project, want zonder hem en zijn hulp had ik het misschien niet eens gered.

Eindhoven, oktober 2001.
Johan Geerlings.

Inhoudsopgave

1 Inleiding

1.1 Inleiding

Het internet neemt tegenwoordig eens steeds belangrijkere plaats in in het dagelijks leven. Ook breedbandverbindingen met het internet worden steeds normaler. Op de Technische Universiteit Eindhoven is zo'n breedbandverbinding aanwezig. Met de extra mogelijkheden die deze breedbandverbinding biedt wordt meestal relatief weinig gedaan, dus is het de moeite waard om het gebruik hiervan aan te moedigen. Het bedrijf Cebra, een BV onder de TUE-holding, is bezig met het Virtue-project, een project dat bedoeld is voor het promoten van het gebruik van een breedband-internetverbinding door het aanbieden van 3D internet. Binnen het Virtue-project wordt gewerkt aan een virtuele 3D-wereld op internet. Binnen deze werelden moeten animatie en interactie een belangrijke rol gaan spelen.

Het toevoegen van de mogelijkheden voor het specificeren van beweging en interactie is het hoofddoel van mijn project. De resultaten van dit project zullen uiteindelijk toegepast gaan worden binnen Virtue. Dit project werd uitgevoerd in samenwerking met de bedrijven Cebra en PhilemonWorks. Een belangrijk onderdeel van dit project was het uitbreiden van de door PhilemonWorks ontwikkelde scripttaal voor het bouwen van 3D werelden, SmallScript3D, met ondersteuning voor beweging en dynamische scènes. Ook het toevoegen van mogelijkheden voor interactie en ondersteuning voor meerdere gebruikers speelde een belangrijke rol.

1.2 Virtue

Een van de projecten van Cebra is het Virtue-project [1]. In dit project wordt gewerkt aan de Virtue-wereld, een virtuele 3D-wereld op Internet. Deze 3D-wereld is gebaseerd op een model van de campus van de TU/e. Virtue is een afkorting voor Virtuele TU/e. De Virtue-wereld bestaat uit drie onderdelen: de Virtuele Bunker, waar studentenverenigingen hun eigen ruimte kunnen inrichten, de Virtuele Plaza, waar bedrijven en andere commerciële instellingen hun virtuele ruimte kunnen inrichten en diensten kunnen aanbieden, en SAfE, waarbij een identificatie- en authenticatie-infrastructuur voor e-communities wordt ontworpen. Binnen Virtue wordt geëxperimenteerd met breedband internet. Op de TU/e is een breedband netwerk aanwezig, maar van de mogelijkheden die dit biedt wordt relatief weinig gebruik gemaakt. Het Virtue project is opgezet met het doel om hierin verandering te brengen.

Belangrijke aspecten van de Virtue-wereld zijn 3D technologie, videocommunicatie en binnenkort ook software agents. Daarnaast biedt Virtue extra mogelijkheden voor het experimenteren met e-commerce. Het belangrijkste doel van Virtue is om de TU/e omgeving voor studenten en medewerkers een stuk leuker te maken. Het is de bedoeling om de gebruikers voorbij de grenzen van het huidige internet te voeren, waar het zoeken (en vaak niet vinden) van informatie centraal staat. Het belangrijkste doel is het onderzoeken hoe de virtuele wereld zich ontwikkelt wanneer we ons in plaats van met het zoeken naar informatie bezig gaan houden met communiceren en actief bezig zijn.

1.3 Projectorganisatie

Dit project is uitgevoerd in samenwerking met twee bedrijven: Cebra en PhilemonWorks. PhilemonWorks is één van de partners van Cebra bij het Virtue-project. PhilemonWorks houdt zich hierbij met name bezig met het 3D-deel van het project.

1.3.1 Cebra

Cebra (Center for Electronic Business Research & Application) [2] is een aparte B.V. onder de TUE Holding die zich bezighoudt met de ontwikkeling van kennis op het gebied van e-commerce. De voornaamste doelstelling van Cebra is het versnellen van de kennisuitwisseling door het concentreren

van de e-commerce kennis op de TUE op één centrale plaats. Binnen de TU Eindhoven bestaat een diversiteit aan e-commerce kennis, waaronder e-procurement, e-logistiek, e-marketing etc. op de faculteit Technologie management; ontwerp en bouw van E-commerce systemen bij de faculteit Wiskunde en Informatica; Versleuteling van gegevens bij Cryptografie; User Interface Design bij IPO, 3D Virtuele omgevingen bij de faculteit Bouwkunde, etc. Binnen Cebra moet deze kennis toegankelijk worden gemaakt voor zowel universiteit als bedrijfsleven.

1.3.2 PhilemonWorks

PhilemonWorks [3] is een bedrijf dat zich voornamelijk bezighoudt met 3D-visualisatie, animatie, virtual-reality en internet. Hierbij specialiseert PhilemonWorks zich vooral op het weergeven van visualisaties en animaties op webpagina's. Ook speelt PhilemonWorks een rol bij de ontwikkeling van full-motion simulatoren voor defensie, luchtvaart en transport. PhilemonWorks heeft eigen software voor het bouwen van 3D-werelden ontwikkeld: SmallScript3D. Dit is een object-geörienteerde ontwikkelomgeving voor het bouwen van 3d-werelden, die zo is ontworpen dat zij vrijwel alle standaard visualisatie-technieken ondersteunt. Met SmallScript3D is het eenvoudig om werelden in verschillende gangbare formaten te maken. PhilemonWorks gebruikt SmallScript3D ook voor het Virtue-project.

2 3D-Animatie

2.1 Het bouwen van een 3D-wereld

Er bestaan verschillende manieren om 3D-werelden te maken. De meest eenvoudige hiervan is het bouwen van de wereld in een editor, zoals 3D studio Max. De belangrijkste kenmerken van editors zijn dat ze interactief, visueel en eenvoudig in gebruik zijn. Verder is er in dit soort programma's veel feedback: in de editor is de wereld te bekijken, en alle wijzigingen aan de wereld zijn direct zichtbaar.

Met een editor is op een relatief eenvoudige manier een 3D wereld te maken: in principe kan iedereen na een beetje oefening met dit soort programma's overweg. Doordat alle wijzigingen direct zichtbaar zijn worden ook veel fouten voorkomen, doordat het meteen opvalt als er bijvoorbeeld een object verkeerd is geplaatst. Het belangrijkste nadeel van een editor is dat het maken van een wereld nogal bewerkelijk is: het kost relatief veel tijd om een 3D-wereld te maken, omdat ieder object in deze wereld afzonderlijk geplaatst moet worden. Ook is het hergebruik van objecten lastiger. Het is bijvoorbeeld wel mogelijk om een boom te maken en deze meerdere keren in een wereld te plaatsen, maar om bijvoorbeeld een weg met 100 bomen erlangs te maken zal deze boom ook 100 keer daadwerkelijk neergezet moeten worden.

Een andere methode is om de 3D-wereld te programmeren in een scripttaal, zoals SmallScript3D, en de wereld vervolgens hieruit te genereren. Het belangrijkste voordeel hiervan is de snelheid: een programmeur met ervaring in deze scripttaal zal in staat zijn om heel snel een 3D wereld te programmeren. Ook is het op deze manier heel eenvoudig om objecten meerdere keren te plaatsen in een wereld. Als er bijvoorbeeld eenmaal een boom is gemaakt, dan is het is heel eenvoudig om een weg te maken waarlangs om de 25 meter zo'n boom neergezet wordt, of bijvoorbeeld een bos waarin honderd bomen op willekeurige posities geplaatst zijn. Hergebruik van objecten is op deze manier dus heel gemakkelijk. Een ander voordeel van het gebruik van een scripttaal is dat deze taal niet afhankelijk hoeft te zijn van het formaat van de 3D-wereld. Het is mogelijk om de taal zodanig te maken dat deze meerdere formaten ondersteunt. Bij het genereren van de wereld uit het script moet dan worden aan gegeven wat het doelformaat is, bijvoorbeeld VRML. Het is eenvoudig om uit hetzelfde script werelden in meerdere formaten te genereren.

Het belangrijkste nadeel van deze methode is dat er programmeerkennis voor vereist is. Deze methode is dus niet geschikt voor de incidentele gebruiker die op een zondagmiddag eens een 3D-wereldje in elkaar wil gaan zetten. Ook het ontbreken van directe feedback is een belangrijk nadeel: om de wereld te kunnen testen moet deze eerst worden gegenereerd vanuit het script, en vervolgens worden geopend in een viewer. Dit is nogal omslachtig, en kan ook veel tijd kosten. Omdat er geen directe feedback is is het ook veel gemakkelijker om fouten te maken.

In het algemeen is een editor beter geschikt voor het maken van losse 3D objecten dan een scripttaal, omdat het bij 3D objecten voornamelijk gaat om het uiterlijk. In een editor is direct te zien hoe een object er uitziet, en is het dus eenvoudig om hier direct aanpassingen aan te maken. Een scripttaal geschikter voor het maken van complete 3D werelden waarin bestaande objecten worden geplaatst, omdat het in een scripttaal heel eenvoudig is om bestaande objecten meerdere malen in een wereld te plaatsen. Omdat het bij het Virtue-project gaat om het bouwen van complete 3D werelden is de logische keus hier dus een scripttaal. Omdat PhilemonWorks, een van de partners van Cebra in dit Virtue-project, al een eigen scripttaal voor het bouwen van 3D werelden ontwikkeld heeft en deze taal hiervoor inderdaad goed bruikbaar blijkt te zijn is de keuze op deze taal gevallen: SmallScript3D.

SmallScript3D is heel goed bruikbaar voor het bouwen van 3D werelden, maar is zeer beperkt in de mogelijkheden voor beweging. Het is wel mogelijk om objecten in een 3D wereld een eenvoudige beweging of rotatie te laten maken, maar complexe bewegingen en interactie zijn niet mogelijk. In het Virtue-project is het de bedoeling om dynamische werelden te creëren, waarin veel animatie plaatsvindt en veel interactie mogelijk is. Om SmallScript3D goed te kunnen gebruiken binnen het Virtue-project is het nodig om de dynamische aspecten van SmallScript3D uit te breiden.

2.1.1 SmallScript3D

SmallScript3D is een uitbreiding van de programmeertaal Smalltalk. SmallScript3D bestaat uit een verzameling van klassen en methoden (in Smalltalk), waarmee het mogelijk is om 3D-werelden te bouwen. Een met SmallScript3D gebouwde 3D-wereld kan worden vertaald naar verschillende gangbare formaten zoals VRML, POVRay, Direct3D en Scol.

Een voorbeeld van een kleine wereld, gebouwd met SmallScript3D:

```
1    | builder |
2    builder := SceneBuilder vrml.
3    builder color: Color yellow.
4    builder add: (Sphere center: 20,40,10 radius: 10).
5    builder color: Color green.
6    builder add: (Box origin: 0,0,0 corner: 20,20,20).
7
8    builder add:(Camera at: 100,50,50 lookingAt:0,20,0).
9    builder writeToFile: 'Bol en Box'
```

Opmerking: de regelnummers horen niet bij het script; ze zijn toegevoegd om in dit document uitleg over het script te kunnen geven.

Het declareren van variabelen in Smalltalk gebeurt door aan het begin van het programma de namen van deze variabelen tussen verticale strepen te zetten. Het declareren van twee variabelen `bol` en `kubus` zou dus als volgt gaan: `| bol kubus |`. Bij het declareren van variabelen hoeft alleen de naam genoemd te worden; het type maakt niet uit. Dit type wordt dynamisch bepaald op het moment dat een variabele een waarde krijgt. In dit voorbeeld wordt in regel 1 wordt een variabele `'builder'` gedeclareerd. In regel 2 wordt `'builder'` geïnitialiseerd: aan de variabele wordt een scenebuilder toegekend. Een scenebuilder bevat de wereld zoals die in SmallScript3D wordt beschreven. In deze toekenning is `'SceneBuilder'` de klasse, en `'vrml'` de constructor voor de VRML-versie van de scenebuilder. Er kan hier ook gekozen worden voor een van de andere builders voor de andere formaten die door ss3d ondersteund worden.

In regel 3 wordt de kleur ingesteld. `'builder color:'` betekent hier dat van het object `'builder'` de methode `'color'` wordt aangeroepen (om de kleur in de wereld in te stellen). De dubbele punt achter `'color'` betekent dat deze functie een parameter verwacht. Deze parameter is `'Color yellow'`, waarbij `'Color'` de klasse is en `'yellow'` de constructor voor een instantie van deze klasse die bij de gele kleur hoort. Als in een scenebuilder een kleur is ingesteld krijgen alle objecten die vervolgens aangemaakt worden deze kleur.

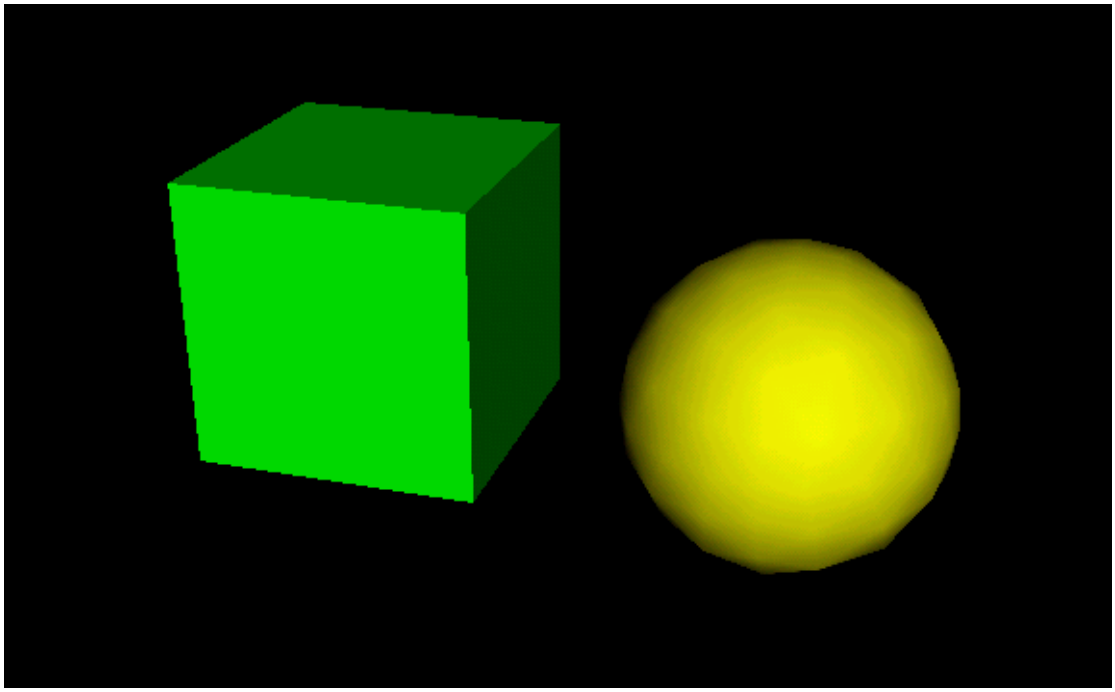
In regel 4 wordt een bol aan de wereld toegevoegd. `'builder add:'` betekent dat er een object aan de wereld toegevoegd moet worden. Dit object wordt gecreëerd in `'Sphere center: 20,40,10 radius: 10'`. Hierbij is `'Sphere'` de klasse van het object, en `'center: 20,40,10 radius: 10'` de constructor. Er wordt hier dus een gele bol (de kleur is in regel 3 ingesteld) gemaakt, met als middelpunt het punt (20,40,10) en als straal 10. Deze bol wordt aan de wereld toegevoegd.

In regel 5 wordt de kleur ingesteld op groen, op dezelfde manier als in regel 3. Alle nieuwe objecten die nu aan de wereld worden toegevoegd krijgen de kleur groen. In regel 6 wordt een kubus aan de wereld toegevoegd. Dit gaat op dezelfde manier als het toevoegen van de bol in regel 4: er wordt in `'Box origin: 0,0,0 corner: 20,20,20'` een instantie van de klasse `'box'` aangemaakt, met als hoekpunten (0,0,0) en (20,20,20), en deze instantie wordt in `'builder add:'` aan de wereld toegevoegd, waarbij de kubus de kleur groen krijgt.

Tenslotte wordt in regel 8 nog een camerapositie ingesteld. Dat gebeurt door het toevoegen van een camera aan de wereld. In `'Camera at: 100,50,50 lookingAt:0,20,0'` wordt deze camera gecreëerd in het punt (100,50,50) en gericht op het punt (0,20,0). De camera wordt vervolgens aan de scenebuilder toegevoegd.

In regel 9 wordt de VRML-file gegenereerd. Hier wordt in `'builder writeToFile:'` de methode `'writeToFile'` aangeroepen. Deze methode zorgt voor ervoor dat er een bestand

gegenereerd wordt, met daarin de wereld in de scenebuilder. Omdat er in regel 2 is gekozen voor de VRML-builder zal dit een VRML-bestand zijn. De bestandsnaam `''Bol en Box''` is de parameter van deze methode. Deze methode genereert nu het bestand 'Bol en Box.wrl'. Dit bestand kan in een VRML-browser worden geopend en bekeken. De wereld die zo ontstaan is ziet er als volgt uit:



SmallScript3D is zeer geschikt voor het bouwen van statische 3D-werelden. Ook zijn er enkele zeer beperkte mogelijkheden om objecten te laten bewegen, maar de dynamische mogelijkheden van SmallScript3D zijn zeer beperkt. Ook zijn er nauwelijks mogelijkheden voor interactie. Dat wordt veroorzaakt door het feit dat het genereren van de wereld uit het SmallScript3D script een eenmalige gebeurtenis is: nadat de wereld is gegenereerd is SmallScript3D hier niet meer bij betrokken, en loopt deze wereld volledig op zichzelf. Het is wel mogelijk om bij het bouwen van de wereld in SmallScript3D sensoren (zoals touch sensors of proximity sensors) in te stellen als deze in het doelformaat ondersteund worden en hiermee wat eenvoudige interactie toe te voegen, maar echte interactie zoals het oppakken van en gooien met een bal is niet mogelijk.

2.2 Animatie in Virtue

2.2.1 Opdrachtomschrijving

Cebra wil voor het Virtue-project gebruik gaan maken van SmallScript3D. In het Virtue-project is het de bedoeling om dynamische werelden te creëren, waarin veel animatie plaatsvindt en veel interactie mogelijk is. SmallScript3D is al geschikt voor het maken van 3D-werelden en objecten, maar bevat zeer weinig ondersteuning voor beweging in deze werelden. Om SmallScript3D goed te kunnen gebruiken binnen het Virtue-project is het nodig om de dynamische aspecten van SmallScript3D uit te breiden. Verder moet SmallScript3D worden uitgebreid met mogelijkheden voor (het specificeren van) interactie.

De opdracht bestaat uit het uitbreiden van SmallScript3D met ondersteuning voor beweging en animatie. Hiermee moet het mogelijk worden gemaakt om objecten in een 3D-wereld op een eenvoudige manier te laten bewegen, en om zo dynamische werelden met veel animatie mogelijk te maken. Op deze manier moet een 'natuurlijke omgeving' gecreëerd kunnen worden, waarin objecten op een realistische manier bewegen en op elkaar reageren. Dit betekent ondermeer dat de wereld aan een aantal basiswetten van de natuurkunde moet voldoen. Objecten moeten bijvoorbeeld tegen elkaar kunnen botsen en daarbij op een realistische manier reageren. Ook moeten wetten als $F = m \times a$ gelden in de wereld, dus objecten moeten versnellen als er een kracht (bijvoorbeeld zwaartekracht) op

werkt. SmallScript3D moet dus uitgebreid worden met klassen en methoden waarmee bewegingen (zoals verplaatsingen en rotaties) toegekend kunnen worden aan objecten of groepen objecten.

Een ander onderdeel van de opdracht is het mogelijk maken van interactie tussen gebruikers en de 3D-wereld. Hiervoor moeten er klassen en methoden aan SmallScript3D worden toegevoegd die interactie tussen een gebruiker en objecten of objecten onderling mogelijk maken.

Verder moet het mogelijk zijn om met meerdere gebruikers in een wereld te zijn. Deze gebruikers moeten elkaar kunnen zien en op elkaar kunnen reageren en met elkaar kunnen communiceren. Als een actie van een bepaalde gebruiker wijzigingen aan de wereld tot gevolg heeft (bijvoorbeeld bij het verplaatsen van een object) moet deze wijzigingen ook voor de andere gebruikers zichtbaar zijn. Hiervoor gaat een client-server architectuur gebruikt worden, waarbij de wereld zal draaien op een server, en de gebruikers (clients) op deze server kunnen inloggen. De server moet zorgen voor de synchronisatie tussen de server en de clients: de wereld op alle clients moet gelijk blijven aan die op de server. Hierbij is de beschikbaarheid van een breedbandverbinding, een van de uitgangspunten van het Virtue-project, een noodzakelijke voorwaarde, want zonder deze breedbandverbinding is het waarschijnlijk niet mogelijk om de synchronisatie op deze manier te realiseren.

Tenslotte moet er nog een test- en demonstratie-wereld, SmallWorld, worden gemaakt. SmallWorld is bedoeld voor het testen en demonstreren van de uitbreidingen van SmallScript3D.

2.2.2 User Requirements

De user requirements zijn de eisen waaraan het systeem moet voldoen volgens de klant. Deze eisen zijn opgesplitst in 4 delen. In het eerste deel worden de eisen met betrekking tot beweging beschreven. Het tweede deel beschrijft de eisen aan de communicatie en de architectuur, die meerdere gebruikers moet kunnen ondersteunen. In het derde deel worden de eisen aan interactie behandeld. Tenslotte worden in het vierde deel de eisen aan de test- en demonstatie-wereld, SmallWorld, beschreven.

Beweging

Met SmallScript3D moet een dynamische wereld gemaakt kunnen worden, waarin veel beweging plaats kan vinden. De belangrijkste eis van de gebruiker is de mogelijkheid om objecten in een wereld op een natuurgetrouwe manier te laten bewegen en op deze manier een ‘natuurlijke omgeving’ te creëren. Bij het plaatsen van objecten in een 3D-wereld moet op een eenvoudige manier aan deze objecten een beweging kunnen worden toegekend. Het soort beweging moet ingesteld kunnen worden (bijvoorbeeld: een rotatie om een bepaalde as of een verplaatsing). Verder moeten er andere kenmerken van deze beweging ingesteld kunnen worden: de snelheid of tijdsduur van de beweging, het tijdstip waarop de beweging begint, of de beweging eenmaal plaatsvindt of voortdurend herhaald wordt en het moment waarop een herhaalde beweging moet stoppen of moet veranderen in een andere beweging. Verder moeten bewegingen kunnen plaatsvinden als gevolg van bepaalde gebeurtenissen binnen de wereld, zoals het naderen van een gebruiker, botsingen tussen objecten of het eindigen van een andere beweging.

Een andere eis is dat objecten niet door elkaar heen mogen kunnen bewegen. Ook mogen gebruikers niet door muren heen kunnen lopen.Er is dus botsingdetectie nodig in de wereld. Er moet een logische reactie plaatsvinden als er een botsing plaatsvindt. Als er bijvoorbeeld een bal tegen een muur aankomt dan zal deze terug moeten stuiteren.

Communicatie en Architectuur

Het moet in de Virtue-werelden mogelijk zijn om met meerdere gebruikers in dezelfde wereld of ruimte aanwezig te zijn. Deze gebruikers moeten elkaar kunnen zien. Hiervoor moet er communicatie tussen de gebruikers plaatsvinden, zodat de gebruikers van elkaar weten waar ze zijn. De Virtue-werelden maken standaard gebruik van een server. Deze server kan ook gebruikt worden voor het afhandelen van de communicatie. Dit heeft als voordeel dat er geen communicatie tussen de gebruikers onderling nodig is. Iedere gebruiker hoeft dus alleen verbinding te maken met de server. Omdat er geen rechtstreekse communicatie tussen de gebruikers onderling is is het ook niet nodig dat deze gebruikers gegevens zoals IP-adressen van elkaar weten. Iedere gebruiker kan dus anoniem blijven als deze dat

wil. Een ander belangrijk voordeel van het afhandelen van de communicatie via de server is dat andere gebruikers in een wereld geen last hebben van een gebruiker met een trage verbinding met het internet: iedere gebruiker is alleen afhankelijk van de snelheid en kwaliteit van zijn eigen verbinding met de server, en een gebruiker die met een trage verbinding (bijvoorbeeld via een modem) zal hier alleen zelf de gevolgen van ondervinden.

Een andere belangrijke eis is dat de wereld bij alle gebruikers hetzelfde dient te zijn. Dit houdt onder andere in dat wijzigingen die door een gebruiker aan de wereld worden gemaakt ook voor de andere gebruikers zichtbaar moeten zijn. Als een gebruiker bijvoorbeeld een bepaald object optilt en ergens anders weer neerzet moet deze verplaatsing ook voor de andere gebruikers zichtbaar zijn. Ook hiervoor is er communicatie nodig. Ook deze communicatie zal worden afgehandeld door de server. De server zorgt dus voor de synchronisatie: de werelden bij de afzonderlijke gebruikers zullen zoveel mogelijk gelijk gehouden worden aan de wereld op de server.

Interactie

Behalve veel animatie is het de bedoeling dat de Virtue-werelden ook veel interactiviteit gaan bevatten. Hiervoor moet het mogelijk zijn om objecten in een wereld te manipuleren. Objecten moeten opgepakt en neergezet kunnen worden, en het moet mogelijk zijn om ze te verplaatsen door eraan te trekken, er tegenaan te duwen of ermee te gooien. Hierbij moet ervoor gezorgd worden dat deze reactie voor alle gebruikers die in de wereld aanwezig zijn te zien is.

Verder moeten gebruikers die zich bij elkaar in dezelfde wereld of ruimte bevinden met elkaar kunnen communiceren en op elkaar kunnen reageren. Het moet bijvoorbeeld mogelijk zijn dat twee gebruikers een bal overgooien.

SmallWorld

SmallWorld is de prototype-wereld, waarmee de uitbreidingen van SmallScript3D getest worden. In SmallWorld zullen zowel interactie als verschillende soorten bewegingen en botsingen mogelijk zijn. SmallWorld is tevens bedoeld als een demonstratiewereld, waarin de mogelijkheden van SmallScript3D worden gedemonstreerd. SmallWorld moet daarom verschillende elementen bevatten waarmee de functionaliteit wordt aangetoond. De belangrijkste onderdelen hiervan zijn het aantonen van de werking van $F = m \times a$ met behulp van een versnellend object (bijvoorbeeld een blokje dat versneld wordt door de zwaartekracht), het aantonen van de mogelijkheid tot interactie (met een object, bijvoorbeeld een blokje, dat de gebruiker kan verplaatsen, optillen of laten vallen), het aantonen van de werking van botsingsdetectie (bijvoorbeeld met een bal die door een gebruiker tegen een muur gegooid kan worden) en van een correcte botsingsrespons (de bal moet op een natuurlijke manier terugstuiteren). Tenslotte moet aangetoond worden dat deze elementen gecombineerd kunnen worden en dat hiermee complexe bewegingen mogelijk zijn. Dit kan bijvoorbeeld met een rollercoaster. Verder kunnen er nog andere elementen aan de wereld worden toegevoegd, zoals een klapdeurtje, een zweefmolen of enige natuurkundige opstellingen, zoals veren met massa's, botsende balletjes of een bal die versneld van een helling afrolt.

2.2.3 Realisatie

Voor het testen van de in SmallScript3D gemaakte werelden zal gebruik gemaakt worden van VRML. In VRML is alleen vooraf gedefinieerde beweging en interactie mogelijk, maar VRML kan gekoppeld worden met Java of Javascript om echt dynamische scènes en interactie mogelijk te maken.

De **eerste fase** van de opdracht bestaat uit het zoeken naar bestaande bibliotheken voor beweging en botsingsdetectie. Deze kunnen dan gebruikt worden door SmallScript3D. Het gebruik van bestaande bibliotheken heeft een aantal voordelen. Een van deze voordelen is dat deze bibliotheken al in de praktijk gebruikt en getest zijn, waarbij de meeste fouten al zullen zijn gevonden en gecorrigeerd. Ook kan het gebruik van deze bibliotheken snelheidswinst opleveren, doordat de gebruikte algoritmes beter geoptimaliseerd zullen zijn. Het belangrijkste voordeel is echter dat het door het gebruik van de bibliotheken niet meer nodig is om zowel dynamica als botsingsdetectie zelf te implementeren, waardoor er veel tijd wordt bespaard.

De **tweede fase**, het uitbreiden van SmallScript3D met beweging, kan dan worden opgelost door het koppelen van SmallScript3D met deze bibliotheken, en het schrijven van klassen en methoden die van deze bibliotheek gebruik gaan maken. Deze klassen zullen in SmallScript3D gebruikt kunnen worden, en voor het berekenen van de bewegingen gebruik maken van de functionaliteit van de bibliotheken.

De **derde fase** is het toevoegen van de mogelijkheid voor het specificeren van interactie. In deze fase moet SmallScript3D worden uitgebreid met klassen en methoden die in staat zijn om acties van gebruikers om te zetten in krachten, en die krachten vervolgens in de dynamica-bibliotheek in te voeren, waardoor de gevolgen van deze acties zichtbaar zijn als nieuwe of gewijzigde bewegingen van objecten in de wereld.

Tenslotte wordt in de **vierde fase** een uitgebreide testwereld, SmallWorld, gemaakt. Deze wereld zal worden gebruikt voor het testen van de aan SmallScript3D toegevoegde mogelijkheden. SmallWorld zal een dynamische wereld worden, waarin veel verschillende soorten beweging en interactie getest worden. SmallWorld zal dus veel verschillende bewegende objecten bevatten, zoals vallende balletjes en een rollercoaster.

3 Functioneel Ontwerp

3.1 Analyse

Cebra wil gebruik gaan maken van SmallScript3D voor hun Virtue-project. Hiervoor is het nodig dat vooral het dynamische deel van SmallScript3D verder ontwikkeld wordt: het moet mogelijk gemaakt worden om op een eenvoudige manier objecten te laten bewegen. SmallScript3D moet dus uitgebreid worden met klassen en methoden waarmee bewegingen (zoals verplaatsingen en rotaties) toegekend kunnen worden aan objecten of groepen objecten. Ook moet interactie tussen gebruikers en objecten en tussen objecten onderling mogelijk zijn.

3.1.1 Beweging

Er zijn verschillende manieren om bewegingen van objecten te specificeren. De twee belangrijkste methoden hiervoor zijn kinematica en dynamica. Bij kinematica worden bewegingen gespecificeerd door van een object de route vast te leggen door middel van snelheden en hoeken op bepaalde tijdstippen. Hieruit kan dan de positie berekend worden. Bij Inverse kinematica werkt dit precies andersom: nu worden de positie en oriëntatie waar een object op een bepaald tijdstip moet zijn gespecificeerd, en worden hieruit snelheden en hoeken berekend.

Met dynamica kunnen bewegingen worden gespecificeerd door het opgeven van krachten. Als van ieder object de massa bekend is en er bekend is welke krachten op dit object werken en hoe groot die zijn dan kunnen vervolgens de bewegingen van die objecten berekend worden. Met dynamica kunnen ook op een eenvoudige manier globale krachten zoals zwaartekracht en wrijving worden gespecificeerd. Krachten zoals zwaartekracht werken op alle objecten in het systeem, en wrijving kan worden gebruikt om het effect van bepaalde krachten gedeeltelijk of geheel teniet te doen.

Met kinematica is het eenvoudig om simpele bewegingen te specificeren: geef een object een snelheid en een richting en het object maakt een rechte beweging. Lastiger is het om met kinematica complexe bewegingen te beschrijven, zoals bijvoorbeeld de beweging van een weggegooide bal waarop zwaartekracht werkt. Hierbij moet namelijk voortdurend de snelheid en de richting aangepast worden. In dit soort gevallen is het veel handiger om dynamica te gebruiken voor het specificeren van de beweging: de beweging kan nu gespecificeerd worden door het instellen van de kracht waarmee de bal wordt weggegooid, de richting en de zwaartekracht. Hieruit wordt dan de gehele beweging van de bal berekend.

Kinematica is geschikter voor werelden met eenvoudige bewegingen; dynamica is geschikter bij werelden waarin complexe bewegingen plaatsvinden. Dynamica is ook veel beter geschikt voor het berekenen van ‘natuurlijke’ bewegingen, zoals die van een schuin naar boven gegooide bal.

Een laatste voordeel van dynamica is dat een wereld zichzelf in principe kan handhaven: nadat alle krachten zijn ingesteld kan de wereld zichzelf in beweging houden door verder te rekenen met de bestaande krachten. Een goed voorbeeld hiervan is een rollercoaster: nadat hier alle initiële krachten (waaronder de zwaartekracht) en enkele andere parameters zijn ingesteld kan er bovenaan een karretje ‘losgelaten’ worden. Dit karretje zal dan geheel vanzelf naar beneden bewegen, waarbij het systeem alle snelheden en posities berekent, zonder dat de gebruiker zich er nog mee hoeft te bemoeien.

Omdat het in het Virtue-project vooral gaat om grote werelden waarin het mogelijk moet zijn om bewegingen natuurgetrouw weer te geven is voor Virtue dynamica dus de beste oplossing. Het doel is dus om SmallScript3D uit te breiden met klassen en methoden voor dynamica. Voor dynamica zal gebruik worden gemaakt van een bestaande bibliotheek.

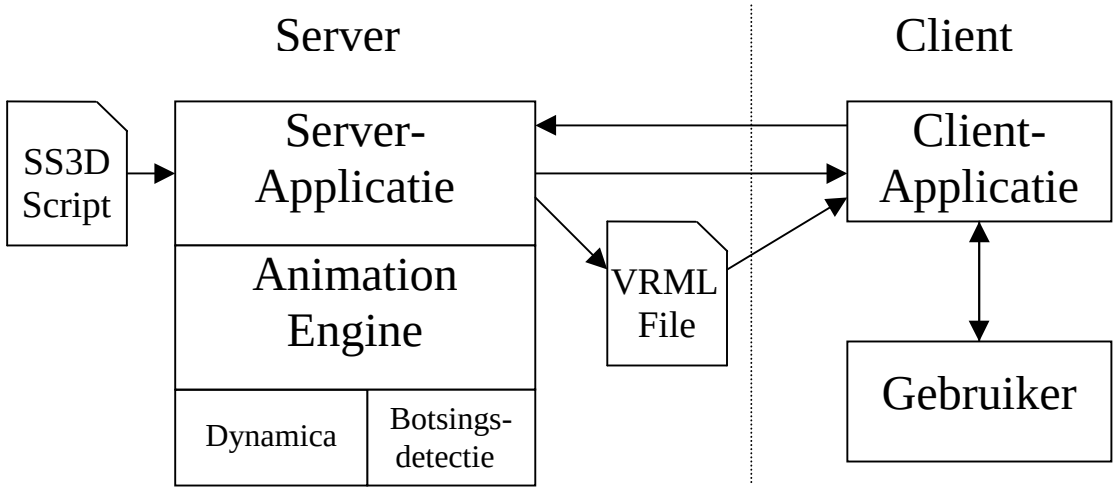
3.1.2 Botsingsdetectie

Om een natuurlijke 3D-wereld te kunnen creëren is het nodig dat objecten in die wereld op elkaar kunnen reageren. Objecten mogen bijvoorbeeld niet door elkaar heen kunnen bewegen. Als twee objecten elkaar raken moeten deze objecten een natuurlijke reactie geven: een bal die een muur raakt zal bijvoorbeeld terug moeten stuiteren, en een fles die door een bal geraakt wordt moet omvallen. Hiervoor moet bepaald kunnen worden wanneer twee willekeurige objecten elkaar raken (of elkaar gedeeltelijk overlappen): botsingsdetectie. SmallScript3D moet dus ook uitgebreid worden met klassen en methoden voor botsingsdetectie. Om de afhandeling van botsingen zo realistisch en natuurgetrouw mogelijk te laten verlopen is het nodig dat er bij iedere botsing wordt berekend welke krachten er op de botsende objecten moeten worden uitgeoefend om een correcte reactie te krijgen. Dit zal het beste werken als de botsingsdetectie is gekoppeld aan de dynamica-bibliotheek, omdat dan alle benodigde gegevens voor de afhandeling van de botsing eenvoudig toegankelijk zijn. Ook voor botsingsdetectie zal gebruik worden gemaakt van een bestaande bibliotheek.

3.1.3 Interactie

Behalve beweging en botsingsdetectie moeten er ook klassen en methoden voor interactie aan SmallScript3D worden toegevoegd. Alle interactie zal worden afgehandeld door de server. Op de server worden alle acties van de gebruikers opgevangen, en als deze acties gevolgen hebben voor de wereld kan de server deze gevolgen zichtbaar maken. Dit kan bijvoorbeeld door het toevoegen van krachten aan de wereld. Deze krachten worden dan door de dynamica-bibliotheek verwerkt, en de gevolgen ervan zijn uiteindelijk voor alle gebruikers zichtbaar. Ook kunnen objecten toegevoegd of verwijderd worden, en kunnen er andere acties (buiten de wereld) plaatsvinden, zoals bijvoorbeeld het openen van een webbrowser met de homepage van een bedrijf.

3.2 Beschrijving Oplossing



Het systeem bestaat uit verschillende lagen. De bovenste laag is de server-applicatie. Deze applicatie wordt uitgevoerd op de server, die geïnitialiseerd wordt door een SmallScript3D-script. De server zorgt verder voor de communicatie met de clients, waardoor acties van de gebruikers kunnen worden verwerkt. In de applicatielaag worden de communicatie en de synchronisatie afgehandeld.

De Server-applicatie maakt gebruik van de Animation Engine. In de Animation Engine wordt de actuele status van de wereld bijgehouden. De Animation Engine zorgt ervoor dat bewegingen en acties van gebruikers zichtbaar gemaakt worden. De Animation Engine berekent voortdurend nieuwe posities en oriëntaties voor alle objecten in de wereld. Hierbij zal de Animation Engine gebruik maken van bibliotheken voor dynamica en botsingsdetectie. De bibliotheken die hiervoor gebruikt worden zijn DynaMo voor de dynamica en SOLID voor de botsingsdetectie. Deze bibliotheken zijn beide ontwikkeld op de Technische Universiteit Eindhoven (TU/e).

Zowel de Server-applicatie als de Animation Enigine worden geïnitialiseerd met behulp van een SmallScript3D-script. Met dit script wordt de initiële situatie in de wereld beschreven, de bewegingen die in deze initiële situatie aan de gang zijn en alle interactie die in de wereld mogelijk is. Met behulp van het SmallScript3D-script wordt ook de VRML-file gegenereerd die gebruikt wordt voor het initialiseren van de clients.

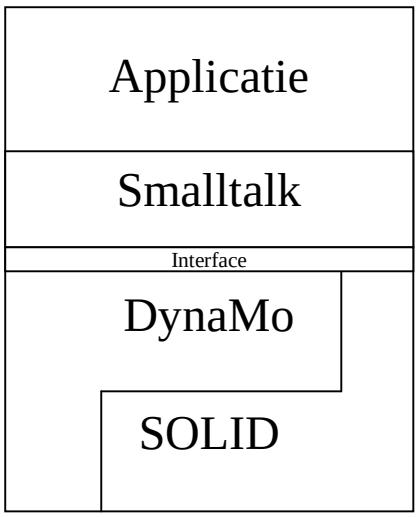
3.2.1 DynaMo

DynaMo [4] kan bewegingen van objecten berekenen met behulp van krachten, impulsen en constraints. DynaMo rekent dus aan dynamica, en maakt hierbij gebruik van natuurkundige wetten zoals $F = m \times a$. De DynaMo-bibliotheek is ontwikkeld door Bart Barenbrug. In DynaMo moeten de posities en oriëntaties van alle objecten in de wereld worden ingevoerd. Van deze objecten worden daarna nog enkele andere parameters ingesteld, waaronder de massa, de wrijving en de elasticiteit (bij botsingen). Daarna kunnen er constraints worden ingesteld: beperkingen in de bewegingsvrijheid van objecten. Er kan bijvoorbeeld ingesteld worden dat twee objecten altijd op een bepaald punt met elkaar verbonden moeten blijven, of dat een object aan één kant aan een muur vastzit. Tenslotte moeten er nog enkele globale parameters worden ingesteld, waaronder de zwaartekracht. Vervolgens kan DynaMo, door te rekenen met de opgegeven krachten, de posities en oriëntaties van ieder object in de wereld gaan berekenen. Dit berekenen kan met een vooraf ingestelde nauwkeurigheid, bijvoorbeeld 25 keer per seconde. De posities en oriëntaties die door DynaMo zijn berekend kunnen door een ander programma gebruikt worden om de wereld en de beweging die hierin plaatsvindt weer te geven.

3.2.2 SOLID

SOLID [5] is ontwikkeld door Gino van den Bergen. SOLID is ontworpen voor botsingsdetectie van bewegende driedimensionale objecten. In SOLID worden de positie, oriëntatie en vorm van alle objecten in een wereld ingevoerd. Iedere keer als deze objecten bewegen (van plaats veranderen) moeten de nieuwe posities van deze objecten aan SOLID worden doorgegeven. SOLID controleert dan of deze nieuwe posities ergens botsingen veroorzaken. Zo ja, dan geeft SOLID aan de toepassing die van SOLID gebruik maakt door dat er een botsing plaatsvindt, waar deze plaatsvindt en om welke objecten het gaat. Verder wordt de ‘Penetration depth’ teruggegeven. Deze penetration depth is de kortste vector waarover een van de objecten verplaatst moet worden om de twee objecten elkaar te laten raken. Deze penetration depth kan gebruikt worden als een benadering van de normaal van de botsing. Aan de hand van de van SOLID ontvangen gegevens kan ervoor gezorgd worden dat er op een juiste manier op deze botsing wordt gereageerd.

3.2.3 Realisatie



SOLID en DynaMo worden aan Smalltalk gekoppeld. Hiervoor is een interface tussen Smalltalk en deze in C++ geschreven bibliotheken nodig. DynaMo wordt volledig vanuit Smalltalk aangestuurd, en SOLID wordt geïnitialiseerd vanuit Smalltalk en vervolgens verder aangestuurd vanuit DynaMo. De functionaliteit van beide bibliotheken is toegankelijk via klassen en methoden die aan SmallScript3D zijn toegevoegd.

Via de interface tussen Smalltalk en de bibliotheken is de volledige functionaliteit die SOLID en DynaMo bieden te gebruiken in Smalltalk, en dus ook vanuit SmallScript3D. Het testen van de interface gebeurt door (via de interface) gebruik te maken van SOLID en DynaMo, en de resultaten te verwerken in een VRML-file. Door het bekijken van de VRML-file in een VRML-browser kan dan worden gecontroleerd of SOLID en DynaMo correct functioneren.

Vervolgens wordt de functionaliteit (uit de bibliotheken) aan SmallScript3D toegevoegd: er worden in SmallScript3D klassen en methoden gemaakt voor het specificeren van beweging. Deze klassen en methoden zullen gebruik maken van de functionaliteit van SOLID en DynaMo. Het toevoegen van deze functionaliteit gebeurt met een cyclische aanpak. De onderdelen hiervan zijn:

- Het (via Smalltalk) gebruiken van een nieuwe functie/beweging uit de bibliotheken
- Het testen van die functie in VRML
- De functionaliteit toevoegen aan SmallScript3D

Deze punten worden herhaald voor alle functies die aan SmallScript3D worden toegevoegd.

3.3 Software Requirements

De software requirements zijn de eisen waaraan het systeem moet voldoen om aan de user requirements te voldoen. Deze eisen zijn opgesplitst in 3 delen: SmallScript3D, de Animation Engine en SmallWorld.

3.3.1 SmallScript3D

In deze sectie worden de eisen waaraan SmallScript3D moet voldoen weergegeven. Deze eisen zijn verdeeld in twee groepen: de eisen met betrekking tot beweging en animatie en de eisen met betrekking tot interactie.

Beweging

Voor beweging zal er gebruik worden gemaakt van de DynaMo-bibliotheek. SmallScript3D zal zodanig worden uitgebreid dat de functionaliteit van DynaMo te gebruiken is. In SmallScript3D zal mogelijk zijn:

B01 Het aanmaken van objecten, met als parameters:
   a. De massa
   b. De botsingselasticiteit
   c. De inertiatensor

B02 Het aanmaken van verbindingen tussen objecten (de constraints), met parameters:
   a. De breekkracht van de verbinding (bij welke kracht breekt de verbinding)
   b. Het type verbinding (scharnierverbindingen, touwen of kogelverbindingen, ...)

B03 Het instellen van krachten
   a. Op bepaalde objecten
   b. Globaal (bijvoorbeeld zwaartekracht)

B04 Het instellen van veranderingen van bepaalde krachten als gevolg van events:
   a. Het verstrijken van tijd (het bereiken van een bepaalde waarde van een timer)
   b. Acties van een gebruiker
   c. Onderlinge interactie tussen objecten (zoals bij een botsing)

B05 Het instellen van de beginsnelheid en -richting van objecten

B06 Het instellen van constraints:
   a. Maximale snelheden van objecten
   b. Maximale krachten die op objecten kunnen worden werken

c. Begrenzing van het gebied waarin een bepaald object zich mag bevinden

Interactie

Verder zal SmallScript3D worden uitgebreid met een aantal klassen die interactie tussen een gebruiker en de wereld of tussen gebruikers onderling mogelijk maken. Hierbij is ook botsingsdetectie nodig. Voor botsingsdetectie zal gebruik worden gemaakt van de SOLID-bibliotheek.

Deze klassen zullen mogelijk maken:
Het definiëren van bewegingen of aanpassingen van de eigenschappen (zoals positie, oriëntatie, grootte, vorm, kleur of zichtbaarheid) van objecten die plaats moeten vinden als gevolg van:

I01 Botsingen tussen:
   a. Een gebruiker en een object
   b. Objecten onderling
I02 Input van de gebruiker (met de muis of het toetsenbord)
I03 Het activeren van Sensoren door:
   a. De nabijheid van een gebruiker of object
   b. Het aanklikken van een bepaald object
I04 Het bereiken van een bepaalde waarde van een timer

3.3.2 De Animation Engine

Voor beweging zal het systeem gebruik maken van de DynaMo-bibliotheek, voor botsingsdetectie wordt de SOLID-bibliotheek gebruikt. De Animation Engine zal deze bibliotheken gebruiken bij het berekenen van alle bewegingen die in een 3D-wereld plaatsvinden.

S01 Het aanroepen van SOLID voor het detecteren van:
   a. Botsingen tussen objecten en een gebruiker
   b. Botsingen tussen objecten onderling
S02 Het aanroepen van DynaMo voor het bewegen van alle objecten aan de hand van de krachten die werken op die objecten
S03 Het synchroniseren van de objecten in de wereld (positie en oriëntatie) met de objecten in DynaMo
S04 Het berekenen van nieuwe krachten aan de hand van de input van de gebruikers en de bestaande krachten

3.3.3 SmallWorld

SmallWorld is de prototype-wereld, waarmee de uitbreidingen van SmallScript3D getest zullen worden. In SmallWorld zullen zowel interactie als verschillende soorten bewegingen en botsingen mogelijk zijn.

In SmallWorld dient minimaal aanwezig te zijn:

W01 Een blokje dat versnelt (of valt) bij aanklikken door een gebruiker
W02 Een blokje waarmee een gebruiker de volgende operaties kan uitvoeren:
   a. Verschuiven
   b. Optillen en laten vallen
   c. Gooien
W03 Een bal die door een gebruiker tegen een muur gegooid kan worden
W04 Een rollercoaster

Eventueel is nog in de testwereld aanwezig:

W05 Een klapdeurtje
W06 Een zweefmolen

W07 Een plaats met een aantal opstellingen waar natuurkundige ‘proefjes’ uitgevoerd kunnen worden, waaronder:

a. Veren met massa’s
b. Botsende balletjes
c. Een bal die versnelt van een helling

4 Technisch Ontwerp

In dit hoofdstuk wordt het technisch ontwerp van het systeem beschreven. In de eerste sectie wordt een beschrijving van de architectuur gegeven en de tweede sectie beschrijft het gebruik van SOLID en DynaMo.

4.1 Architectuur

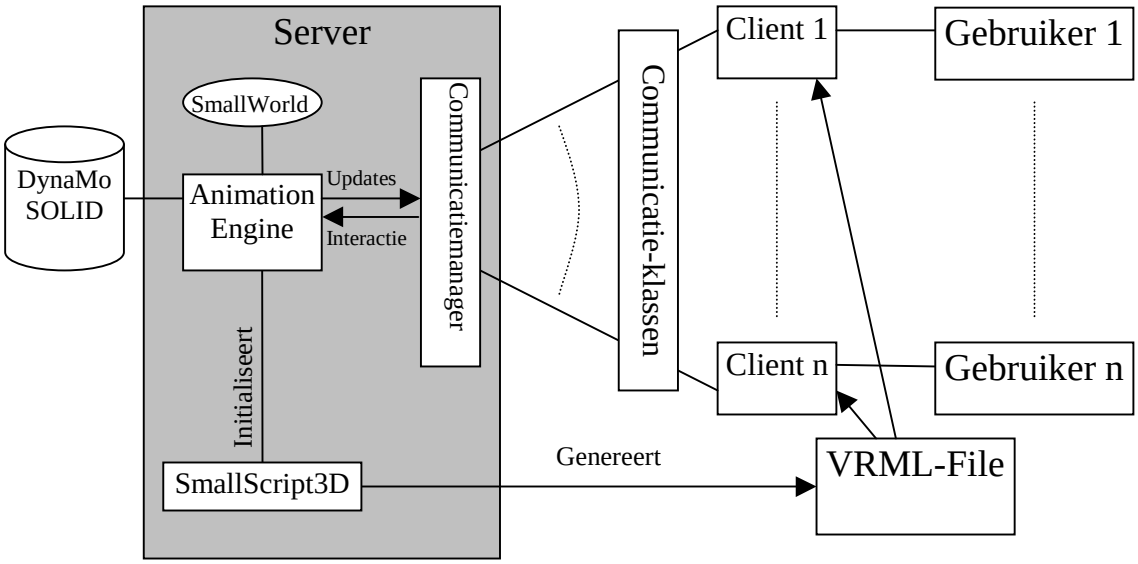
De architectuur van het systeem zal bestaan uit een server met een aantal clients. Door gebruik te maken van een server worden verschillende communicatie- en synchronisatieproblemen voorkomen. Het is op deze manier niet nodig dat de clients onderling gegevens uitwisselen: iedere client communiceert alleen met de server, en de server zorgt ervoor dat de werelden in alle clients gelijk blijven. Het andere grote voordeel van het gebruik van een client-server architectuur is dat de server alle berekeningen voor de bewegingen kan uitvoeren. Hierdoor hoeven de clients niet zelf te gaan rekenen, maar kunnen zich bezighouden met communicatie en rendering.

De clients ontvangen voortdurend updates over alle objecten in de wereld. Omdat het bij het Virtue-project kan gaan om grote werelden met daarin een groot aantal objecten kan de hoeveelheid data die bij deze updates verzonden moet worden ook behoorlijk groot worden. Om de synchronisatie tussen de server en de clients te garanderen is een breedband-verbinding dus een vereiste, omdat het bij een verbinding via een modem bij de grotere werelden waarschijnlijk niet mogelijk is om alle benodigde data op tijd bij de clients te krijgen. Omdat deze breedband-internetverbinding een van de uitgangspunten van het Virtue-project zal de synchronisatie geen problemen opleveren.

Door deze manier van synchroniseren zullen er vertragingen ontstaan tussen het moment dat een gebruiker een actie uitvoert en het moment dat het resultaat van deze actie in de wereld zichtbaar is. Door de beschikbaarheid van de breedbandverbinding zal deze vertraging relatief klein blijven, en dus geen problemen opleveren.

4.1.1 Algemene beschrijving

In SmallScript3D wordt een beschrijving van de wereld gemaakt. Vanuit deze beschrijving wordt de VRML-file gegenereerd. De beschrijving wordt ook gebruikt voor het initialiseren van de wereld in de Animation Engine.



Op de clients wordt de wereld geïnitialiseerd door het inlezen van de VRML-file. Na het initialiseren wordt – via de communicatieklassen - verbinding gemaakt met de server. De Animation Engine in de

server zorgt voor het bijhouden en bijwerken van de actuele status van de wereld. Veranderingen van die wereld worden via de communicatie-klassen van PhilemonWorks gecommuniceerd naar de clients. Input van de gebruikers (bij de clients) worden doorgegeven naar de server, waarna de Animation Engine de nieuwe situatie berekent. Bij deze berekeningen maakt de Animation Engine gebruik van de van de bibliotheken SOLID en DynaMo. De wijzigingen worden vervolgens doorgegeven aan alle clients.

4.1.2 Componenten

De Server
De server bestaat uit een aantal verschillende componenten, waaronder een communicatiemanager, een Animation Engine en de SmallScript3D-componenten. SmallScript3D wordt gebruikt voor het maken van een beschrijving van de 3D-wereld en van de animatie die hierin plaatsvindt. Ook worden de mogelijke interacties en de gevolgen ervan hierin beschreven. De SmallScript3D-beschrijving wordt gebruikt voor het initialiseren van de Animation Engine. In de Animation Engine wordt de actuele status van de wereld bijgehouden en worden de bewegingen berekend. De communicatiemanager zorgt voor de communicatie met de clients, en zorgt ervoor dat de wereld in de clients gelijk blijft aan die op de server, door de updates in de wereld van de Animation Engine door te geven aan de clients en alle externe acties (bijvoorbeeld input van een gebruiker) van de clients door te geven aan de Animation Engine.

De Animation Engine
In de Animation Engine wordt de actuele status van de wereld bijgehouden. Aan de hand van de SmallScript3D-beschrijving wordt de wereld geïnitialiseerd. Als een gebruiker in een van de clients acties uitvoert zal die client die acties doorgeven aan de communicatiemanager in de server. De gevolgen van deze acties zullen vervolgens door de Animation Engine worden berekend. De Animation Engine zal de nieuwe situatie in de wereld berekenen, door de nieuwe posities en oriëntaties van alle objecten in de wereld te berekenen, en zal deze vervolgens versturen naar alle clients. Voor de berekeningen van de bewegingen zal de Animation Engine gebruik maken van de bibliotheken SOLID (voor botsingsdetectie) en DynaMo (voor dynamica).

De Communicatiemanager
De communicatiemanager zorgt voor het afhandelen van de communicatie in de server. Alle communicatie tussen de server en de clients zal plaatsvinden via de communicatie-klassen van PhilemonWorks. In de server zal de communicatiemanager zorgen voor het ontvangen van berichten, het doorgeven van relevante berichten aan de Animation Engine en voor het doorsturen van updates van de wereld naar de clients.

De Clients
Een client is een VRML-browser. Op een client wordt de wereld geïnitialiseerd door het inlezen van de VRML-file. Na het initialiseren wordt – via de communicatieklassen - verbinding gemaakt met de server en wordt de lokale wereld gesynchroniseerd met de wereld op de server. De client zorgt ervoor dat alle input van de gebruiker doorgegeven wordt aan de server. De client zorgt er verder voor dat de wereld in de client gesynchroniseerd blijft met de wereld op de server, door de wereld voortdurend te actualiseren aan de hand van de berichten van de server.

SmallScript3D
De beschrijving van de wereld wordt gemaakt met SmallScript3D. Deze beschrijving wordt zowel gebruikt voor het genereren van de VRML-file als voor het initialiseren van de wereld op de server. In de SmallScript3D-file zit een beschrijving van de statische wereld, van alle krachten die in de beginsituatie op objecten werken, en van de beginsnelheden en –richtingen van de objecten. Ook zijn de mogelijke interacties met hun gevolgen erin beschreven.

Communicatie
De communicatie-klassen zorgen voor de uitwisseling van gegevens tussen de server en de clients. Deze communicatie-klassen zullen worden gemaakt door PhilemonWorks.

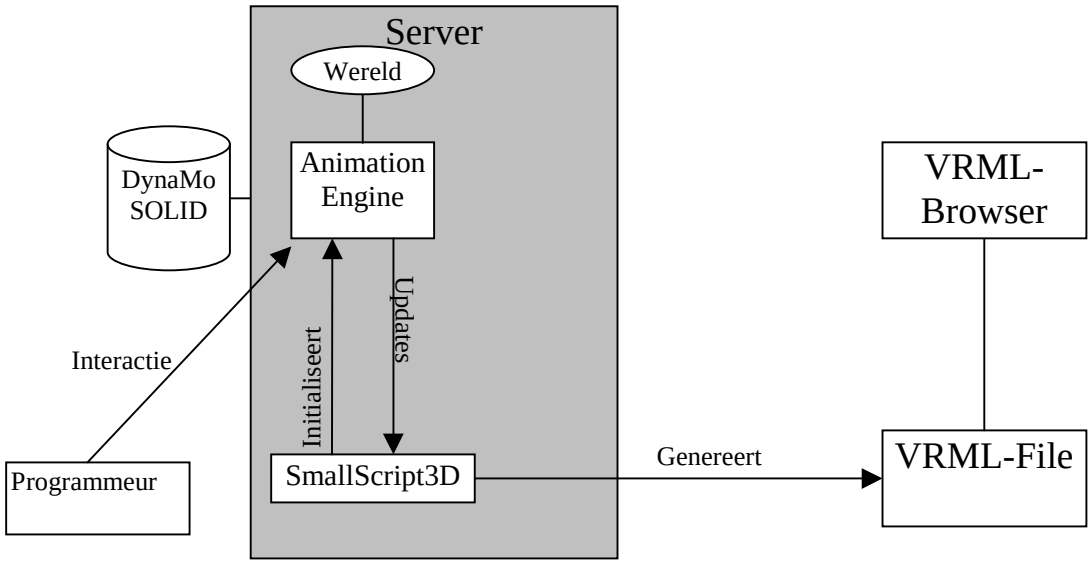
De VRML-file
De VRML-file wordt gegenereerd vanuit de SmallScript3D-beschrijving van de wereld. In de VRML-file zit de beschrijving van de statische wereld. Ook zullen hierin de links naar de communicatie-klassen zitten, zodat die door de VRML-browser op de clients gebruikt kunnen worden.

SmallWorld
SmallWorld is de prototype-wereld, waarin de dynamische aspecten van SmallScript3D getest zullen worden. In SmallWorld zullen een aantal verschillende bewegende objecten aanwezig zijn, en verschillende objecten waarop interactie mogelijk is.

4.1.3 Aanpassingen

Voor de communicatie tussen de Server en de clients zal gebruik gemaakt worden van communicatie-klassen. Deze communicatie-klassen zullen worden gemaakt door PhilemonWorks. Bij het ontwerpen van dit systeem waren deze communicatie-klassen nog niet beschikbaar. Hierdoor was het nog niet mogelijk om met een server en meerdere clients te werken en konden de bewegingen niet realtime berekend en weergegeven worden. Om de functionaliteit van SOLID en DynaMo toch te kunnen testen worden alle bewegingen binnen het systeem uitgerekend en worden vervolgens opgeslagen in een VRML-file. Dit heeft gevolgen voor de gebruikte architectuur: de communicatie en de clients vallen weg, en de updates vanuit de Animation Engine worden via SmallScript3D rechtstreeks in de VRML-file verwerkt. Deze VRML-file bevat dan een soort film, waarin de bewegingen die in de server zijn uitgevoerd wordt weergegeven.

Door het niet beschikbaar zijn van de communicatie-klassen is er ook geen interactie met gebruikers mogelijk. Deze interactie wordt nu door de programmeur gesimuleerd: deze voegt op bepaalde momenten krachten toe aan de wereld, alsof een gebruiker bepaalde handelingen heeft verricht. Op deze manier kan de interactie binnen de wereld ook getest worden. De architectuur komt er met deze wijzigingen als volgt uit te zien:



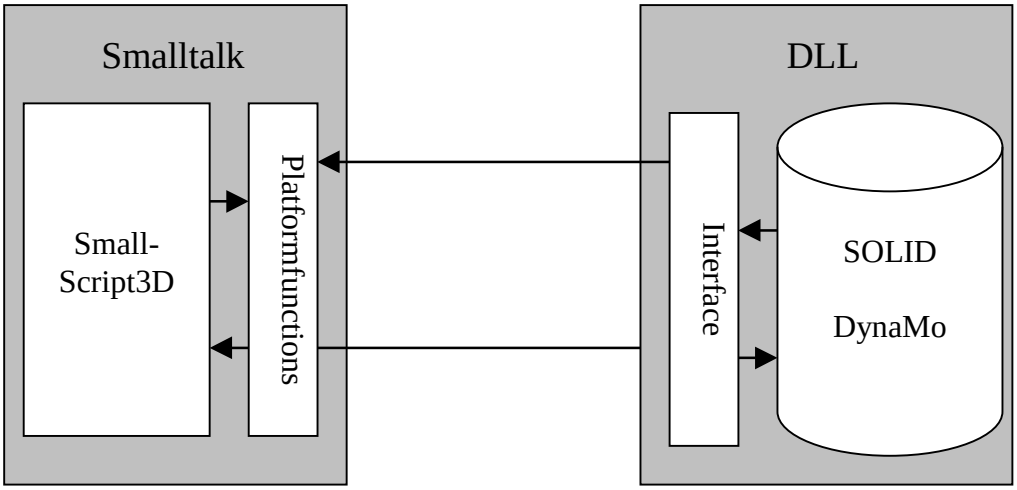
4.2 Interface

SOLID en DynaMo, de bibliotheken voor botsingsdetectie en dynamica, zijn geïmplementeerd in C en C++. Beide bibliotheken zullen gebruikt worden via een DLL. Een DLL (Dynamic Link Library) is een bestand waarin functies en klassen met hun methoden zodanig kunnen worden opgeslagen dat externe programma's hiervan gebruik kunnen maken. DLL's worden veel gebruikt door Windows en Windows-programma's. Smalltalk kan functies in een DLL aanroepen, maar omdat Smalltalk niet overweg kan met C++ klassen en methoden moet er aan de DLL's een extra 'interface-laag' worden

toegevoegd. Deze interface-laag bevat dan functies voor het aanmaken en vernietigen van klassen, en voor het aanroepen van de publieke methoden ervan.

Smalltalk zal voor het aanroepen van de functies in de DLL gebruik maken van platformfuncties. De platformfuncties zijn een onderdeel van de taal Smalltalk waarmee externe functies, bijvoorbeeld in een DLL, aangeroepen kunnen worden. Voor iedere functie in de DLL die door Smalltalk gebruikt wordt moet zo'n platformfunctie worden aangemaakt.

De interface tussen Smalltalk en de DLL's met de bibliotheken SOLID en Dynamo ziet er als volgt uit:



Meer informatie over de interface en het generen hiervan is te vinden in sectie 6.1 en 6.2.

4.3 Gebruik Solid en Dynamo

4.3.1 DynaMo

Om gebruik te kunnen maken van DynaMo moet ieder object in SmallScript3D dat kan bewegen of botsen een corresponderend object (een companion) in DynaMo hebben. Voor statische objecten zoals vloeren en muren, die niet kunnen bewegen maar waarmee wel gebotst kan worden, moet dit de companion een DL_Geo zijn. Voor statische objecten kan zo'n geo vanuit SmallScript3D eenvoudig worden aangemaakt. Bij het aanmaken van een geo moeten de positie en oriëntatie ervan worden ingesteld. Deze positie en oriëntatie blijven constant: de enige manier waarop deze nog gewijzigd kunnen worden is door dit vanuit SmallScript3D te doen. Voor statische objecten is het dus niet nodig om tijdens het uitvoeren van een programma de posities en oriëntaties bij te werken.

Voor dynamische objecten moet de DynaMo-companion van de klasse DL_Dyna zijn. Dyna's worden beïnvloed door de zwaartekracht en door andere krachten. DynaMo berekent voortdurend nieuwe posities en oriëntaties van alle dyna's in het systeem. Bij het aanmaken van een companion voor dynamische SmallScript3D-objecten moet er eerst een 'BasicObject' worden aangemaakt. Dit BasicObject is een object in de interface, dus buiten DynaMo. Aan dit object moeten vervolgens een positie en een oriëntatie worden gegeven. Tenlotte moer er een nieuwe dyna worden aangemaakt. Deze dyna moet dan als companion van het aangemaakte BasicObject worden ingesteld. Hierbij neemt de dyna automatisch de positie en oriëntatie van dit object over. Het aanmaken van de companion moet op deze manier omdat DynaMo bij het wijzigen van de positie of oriëntatie van een dyna een callback-functie aanroept. Omdat DynaMo met deze callback niet rechtstreeks een functie in Smalltalk kan aanroepen zijn de BasicObjects aan de interface toegevoegd. Deze BasicObjects vangen de callbacks vanuit DynaMo op, en updaten dan hun positie en oriëntatie. Deze positie en oriëntatie kunnen vervolgens eenvoudig vanuit Smalltalk worden opgevraagd. Van dynamische objecten moet de positie en oriëntatie in SmallScript3D voortdurend worden bijgewerkt.

4.3.2 SOLID

Voor het toevoegen van objecten heeft SOLID vijf functies. Hiervan zijn er vier voor het toevoegen van standaard-objecten: SolidAddBox voor het toevoegen van een balk, SolidAddCone voor het toevoegen van een kegel, SolidAddCylinder voor een cylinder en SolidAddSphere voor een bol. Voor complexere voorwerpen (die niet gemaakt kunnen worden met de voorgaande vier functies) moet SolidAddNewComplexShape worden gebruikt. Hiermee kan een object met een afwijkende, door de gebruiker gedefinieerde structuur worden gemaakt.

Om zo'n structuur te maken moet de functie SolidMakeNewComplexShape worden aangeroepen. Deze functie geeft een nummer, de ShapeID, terug. Door het aanroepen van deze functie kan een nieuwe structuur worden opgebouwd door het toevoegen van driehoeken (met SolidShapeAddTriangle) en vierhoeken (met SolidShapeAddQuad). Deze drie- en vierhoeken worden altijd aan de laatst aangemaakte structuur toegevoegd. Er kan dus altijd maar één structuur tegelijk worden opgebouwd. Als het opbouwen van de structuur is voltooid moet de functie SolidMakeEndComplexShape worden aangeroepen. Hierdoor wordt de gemaakte structuur opgeslagen, en kan deze worden gebruikt. Hiervoor moet de ShapeID van de gemaakte structuur als parameter worden meegegeven bij een aanroep van SolidAddNewComplexShape.

Bij het toevoegen van objecten aan SOLID moet de positie van het object worden meegegeven. De oriëntatie van de objecten moet apart worden ingesteld.

Nadat alle objecten aan SOLID zijn toegevoegd wordt de verdere afhandeling van de botsingsdetectie gedaan door DynaMo. Iedere keer nadat de functie *dynamics* van DynaMo is aangeroepen (voor het berekenen van de situatie in de wereld op het volgende tijdstip) zal de positie en oriëntatie van alle objecten in SOLID worden bijgewerkt. Dit gebeurt door (vanuit de interface) van ieder DynaMo-object het bijbehorende SOLID-object en de positie en oriëntatie op te vragen en deze door te sturen naar SOLID. Dit gebeurt door het aanroepen van SolidUpdatePos en SolidUpdateOrient. Nadat alle objecten in SOLID zijn bijgewerkt wordt de botsingsdetectie uitgevoerd door vanuit de interface de SOLID-functie *DT_Test* aan te roepen. Deze functie zorgt voor het detecteren en afhandelen van de botsingen.

4.4 Animation Engine

De Animation Engine zorgt voor het berekenen van de bewegingen: de posities en oriëntaties van alle objecten in de 3D-wereld op ieder tijdstip. De Animation Engine is een loop, waarin vier stappen voortdurend herhaald worden.

De **eerste stap** is optioneel: het toevoegen, verwijderen of wijzigen van krachten of constraints aan DynaMo. Hiermee kunnen nieuwe bewegingen aan de wereld toegevoegd worden of bestaande bewegingen worden gewijzigd. Als er acties van een gebruiker verwerkt moeten worden dan moet dat in deze stap gebeuren.

De **tweede stap** is het berekenen van de posities en oriëntaties van alle objecten op het volgende tijdstip. Dit gebeurt door in DynaMo de functie *dynamics* aan te roepen. Deze functie rekent alle nieuwe posities en oriëntaties uit aan de hand van de posities en oriëntaties op het vorige tijdstip en alle gedefinieerde constraints. DynaMo zorgt ervoor dat alle nieuwe posities en oriëntaties ook aan SOLID worden doorgegeven, zodat er botsingsdetectie kan worden uitgevoerd. Nadat DynaMo alle berekeningen heeft afgehandeld zal DynaMo de botsingsdetectie-functie van SOLID aanroepen. SOLID controleert dan of er ergens botsingen plaatsvinden. Als dat het geval is zal SOLID ervoor zorgen dat DynaMo in de volgende loop op deze botsing reageert. Dit gebeurt door het aanmaken van een nieuwe collision-constraint in DynaMo.

De **derde stap** is het bijwerken van alle posities en oriëntaties in de wereld in SmallScript3D. Hierbij wordt van ieder object de nieuwe positie en oriëntatie uit DynaMo gehaald, en worden de posities en oriëntaties van alle objecten op de server aangepast aan die in DynaMo.

In de **vierde stap** moeten tenslotte de clients worden bijgewerkt: alle nieuwe posities en oriëntaties moeten worden gecommuniceerd naar de clients. In de huidige versie van het systeem is deze communicatie nog niet beschikbaar, en worden de updates opgeslagen om later gebruikt te kunnen worden voor het genereren van een VRML-file.

5 SmallWorld

In dit hoofdstuk worden de verschillende onderdelen van de test- en demonstratie-wereld SmallWorld beschreven. Van de meeste hieronder beschreven voorbeelden zijn screenshots van de wereld zoals deze te zien is in een VRML-browser bijgevoegd. Het volledige SmallScript3D-script van één van de SmallWorld-onderdelen is als voorbeeld opgenomen in appendix C.

5.1 Stuiterende Objecten

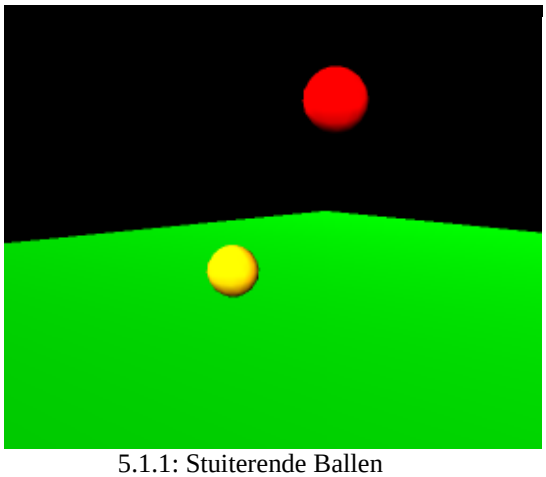
Deze eerste onderdelen van SmallWorld testen en demonstreren enkele basisconcepten van een dynamische wereld: zwaartekracht en botsingsdetectie. In deze voorbeelden worden alleen botsingen met statische objecten zoals vloeren en muren getest.

5.1.1 Ballen

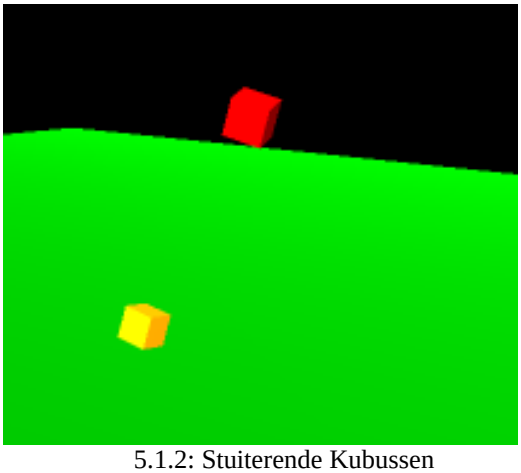
In dit voorbeeld vallen twee ballen van een hoogte op een vloer, en stuiteren vervolgens weer omhoog. Het feit dat de ballen vallen als ze losgelaten worden toont aan dat de zwaartekracht werkt, het feit dat de ballen omhoog stuiteren als ze de vloer raken toont aan dat ook de botsingsdetectie werkt, en dat er op een juiste manier op zo'n botsing gereageerd wordt.

5.1.2 Kubussen

Dit is hetzelfde voorbeeld als 5.1.1, maar dan met twee kubussen in plaats van ballen. Bij dit voorbeeld is goed te zien hoe bij een botsing bewegingsenergie omgezet kan worden in rotatie-energie (de kubus stuitert minder hoog, maar begint om zijn eigen as te draaien), en hoe deze rotatie-energie bij een volgende botsing weer omgezet kan worden in bewegingsenergie: de kubus draait minder snel, maar stuitert opeens weer hoger omhoog.



5.1.1: Stuiterende Ballen



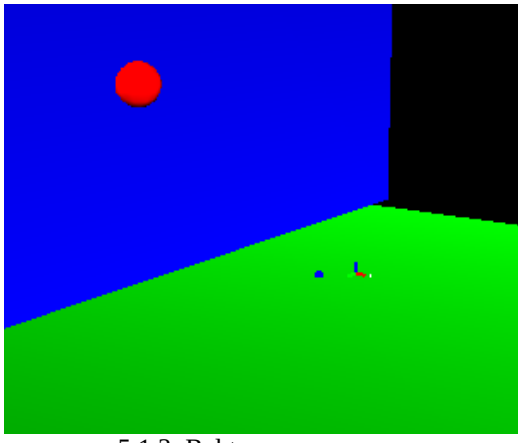
5.1.2: Stuiterende Kubussen

5.1.3 Bal tegen een Muur

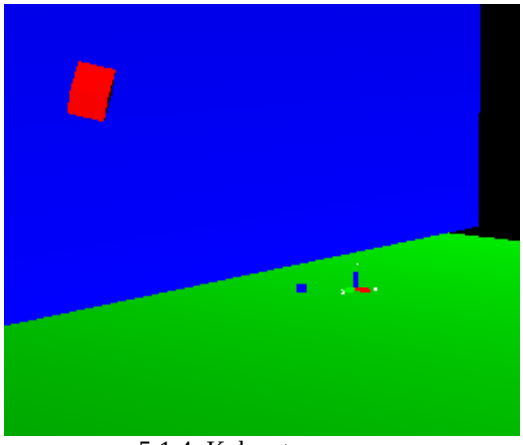
In dit voorbeeld wordt een bal 'afgeschoten' in de richting van een muur. De bal stuitert tegen deze muur en beweegt vervolgens in een andere richting verder. De richting waarin de bal beweegt na de botsing hangt af van de richting van de bal op het moment dat deze de muur raakt. Met dit voorbeeld wordt aangetoond dat botsingen met muren of andere statische objecten gedetecteerd worden en dat hierop op een natuurlijke manier wordt gereageerd.

5.1.4 Kubus tegen een Muur

Dit is hetzelfde voorbeeld als 5.1.3, waarbij de bal weer is vervangen door de kubus. In dit voorbeeld is, net zoals in voorbeeld 5.1.2, duidelijk te zien dat de kubus bij de botsing tegen de muur een deel van zijn beweging omzet in een rotatie rondom zijn as. Deze rotatie wordt nog versterkt op het moment dat de bal de vloer raakt: de bal stuitert nauwelijks nog omhoog, maar begint wel heel snel om zijn as te roteren.



5.1.3: Bal tegen een muur



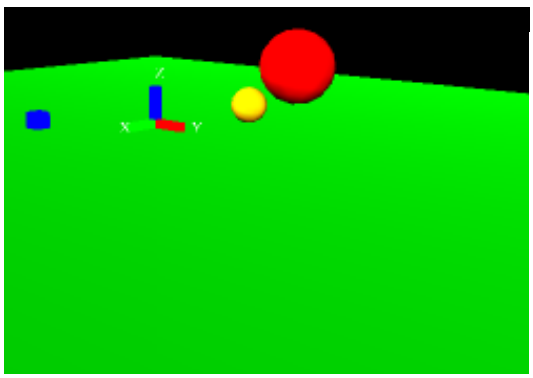
5.1.4: Kubus tegen een muur

5.2 Botsingen met dynamische objecten

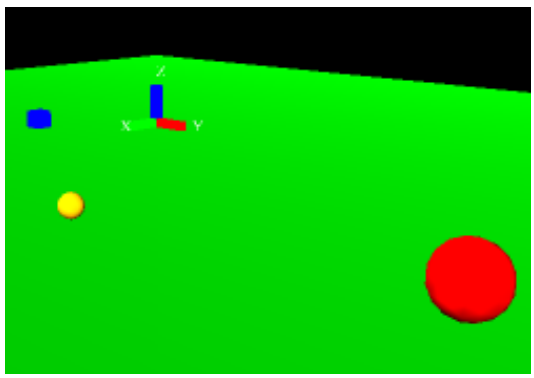
In de volgende voorbeelden worden botsingen tussen bewegende objecten gedemonstreerd en getest. Met deze botsingen worden ook enkele kleine experimenten uitgevoerd.

5.2.1 Botsende ballen

In dit voorbeeld wordt de rode bal van een hoogte losgelaten, zodat deze begint te vallen en te stuiteren. De gele bal wordt in de richting van de rode bal afgeschoten, zodat deze ballen elkaar raken, en met elkaar botsen. Na de botsing bewegen de ballen uit elkaar. Met dit voorbeeld wordt aangetoond dat botsingen tussen bewegende objecten onderling mogelijk zijn, dat deze botsingen gedetecteerd worden, en dat er door beide objecten correct op wordt gereageerd.



Botsing tussen de ballen



De ballen bewegen uit elkaar

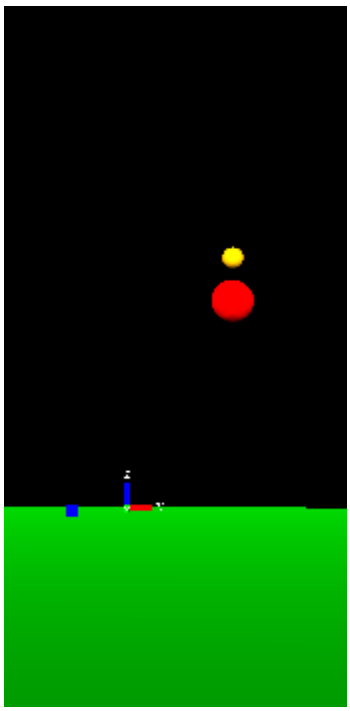
5.2.2 Basketbal en tennisbal

Het volgende voorbeeld is geïnspireerd door een vraag uit de Nationale wetenschapsquiz van 2000. Vraag 20 in deze quiz is:

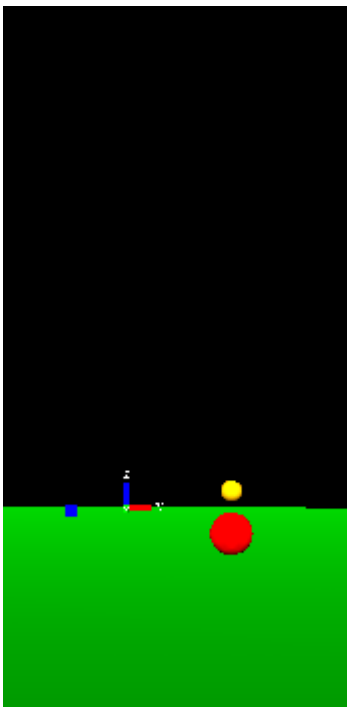
> *Je hebt een basketbal en daarop leg je een tennisbal. Je laat ze samen van 1 meter hoogte vallen. Wat gebeurt er als de basketbal de grond raakt?*

Het antwoord op deze vraag is dat de basketbal bij de botsing met de grond vrijwel al zijn energie overdraagt aan de tennisbal, die hierdoor als een kanonskogel wegschiet en een zeer grote hoogte kan bereiken. De basketbal blijft na de botsing vrijwel stil liggen.

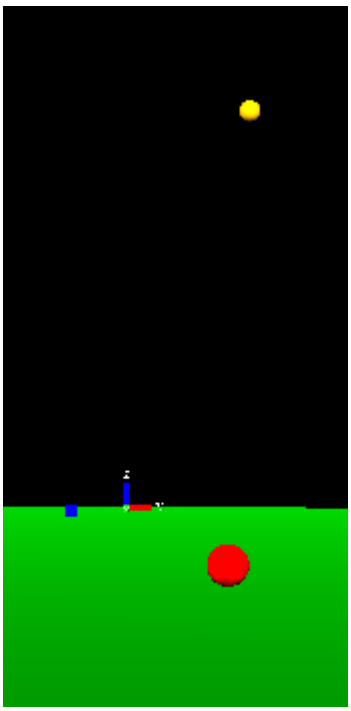
Dit voorbeeld is een simulatie van dit experiment: twee ballen, waarbij de onderste een grotere omvang en massa heeft, worden vlak boven elkaar losgelaten en stuiteren op de grond. Er gebeurt inderdaad wat we volgens de wetenschapsquiz mogen verwachten: de bovenste bal (de 'tennisbal') krijgt een flinke stoot van de onderste bal (de 'basketbal'), en schiet als een kanonskogel omhoog.



De ballen vallen…



Stuiteren op de vloer…

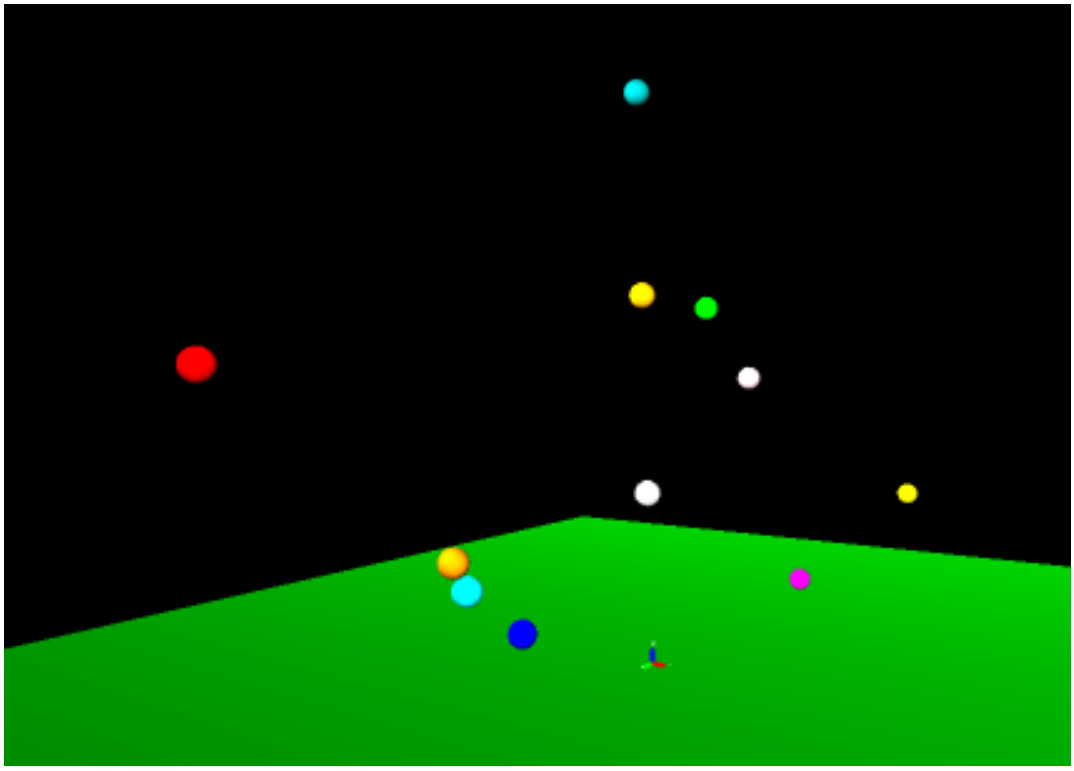


En de tennisbal schiet omhoog

5.2.3 Veel ballen op een vloer

In dit voorbeeld worden een groot aantal ballen op gelijke afstanden recht boven elkaar gezet. Deze ballen worden vervolgens losgelaten en vallen dan op een vloer en op elkaar. Dit voorbeeld heeft enkele overeenkomsten met voorbeeld 5.2.2: de ballen beginnen recht boven elkaar, en zullen elkaar dus bij het terugstuiteren van de vloer raken en wegstoten. De voornaamste verschillen met voorbeeld 5.2.2 zijn dat alle ballen dezelfde grootte en massa hebben, dat er een grotere afstand tussen de ballen zit, en dat er een veel groter aantal ballen is.

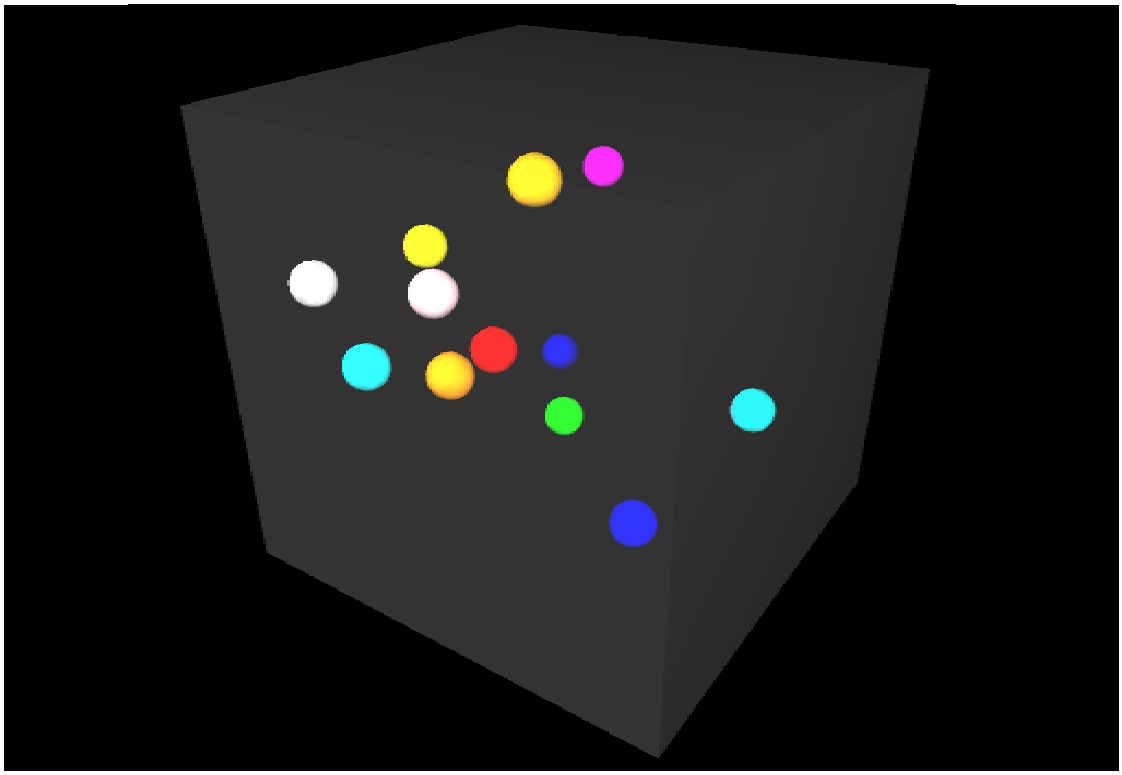
Als de eerste bal de vloer raakt stuitert deze recht omhoog. Hierdoor wordt de tweede bal, die korte tijd later komt, geraakt. Deze bal stuitert dan ook omhoog, terwijl de eerste bal weer omlaag gaat. Ook de tweede bal kan nergens heen, omdat even later de volgende bal beneden is. Van bovenaf komen er steeds meer ballen met steeds hogere snelheid (door de langere valafstand) naar beneden vallen. Hierdoor beginnen de eerste ballen, die al beneden zijn, met hoge snelheid alle kanten op te schieten. Het volledige script voor het genereren van dit voorbeeld is (met uitleg) opgenomen in appendix C.



5.2.3: Chaos met vallende ballen. In het midden komen nieuwe ballen naar beneden vallen, terwijl de ballen die al beneden zijn in alle richtingen wegschieten.

5.2.4 Willekeurig botsende ballen

In dit voorbeeld stuiteren een aantal ballen tegen de binnenwanden van een kubus en tegen elkaar. De ballen worden bij het uitvoeren van het SmallScript3D-script voor dit wereld op willekeurige posities geplaatst, en krijgen een willekeurige beginsnelheid en –richting.

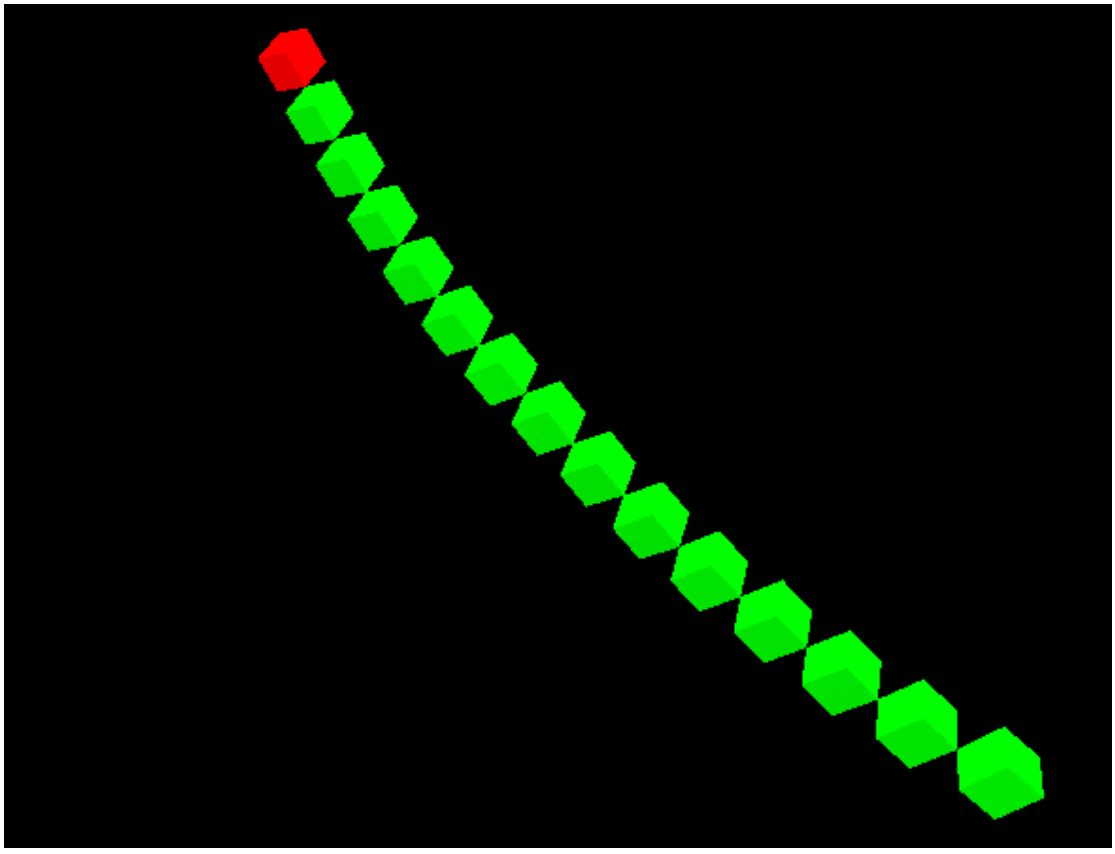


5.3 Constraints

DynaMo heeft de mogelijkheid tot het specificeren van constraints: beperkingen in de bewegingsvrijheid van objecten. Voorbeelden van dit soort constraints zijn objecten die altijd op een bepaald punt met elkaar verbonden moeten blijven of twee objecten die niet verder dan een bepaalde afstand van elkaar mogen bewegen (en dus met een 'touw' aan elkaar vastzitten).

5.3.1 Slingerende kubussen

In dit voorbeeld zijn vijftien kubussen met hun hoekpunten aan elkaar verbonden. De eerste kubus (de rode) is met een hoekpunt verbonden aan de wereld, waardoor deze niet van zijn plaats af kan. De kubussen worden van een hoogte losgelaten. Omdat de kubussen met elkaar zijn verbonden en de rode kubus niet van zijn plaats kan komen beginnen de kubussen een slingerbeweging uit te voeren. Dit voorbeeld is een van de voorbeelden die bij de DynaMo-bibliotheek bijgevoegd zijn. Dit voorbeeld is rechtstreeks overgenomen in SmallScript3D voor het testen van het functioneren van DynaMo en van de posities en oriëntaties van objecten.



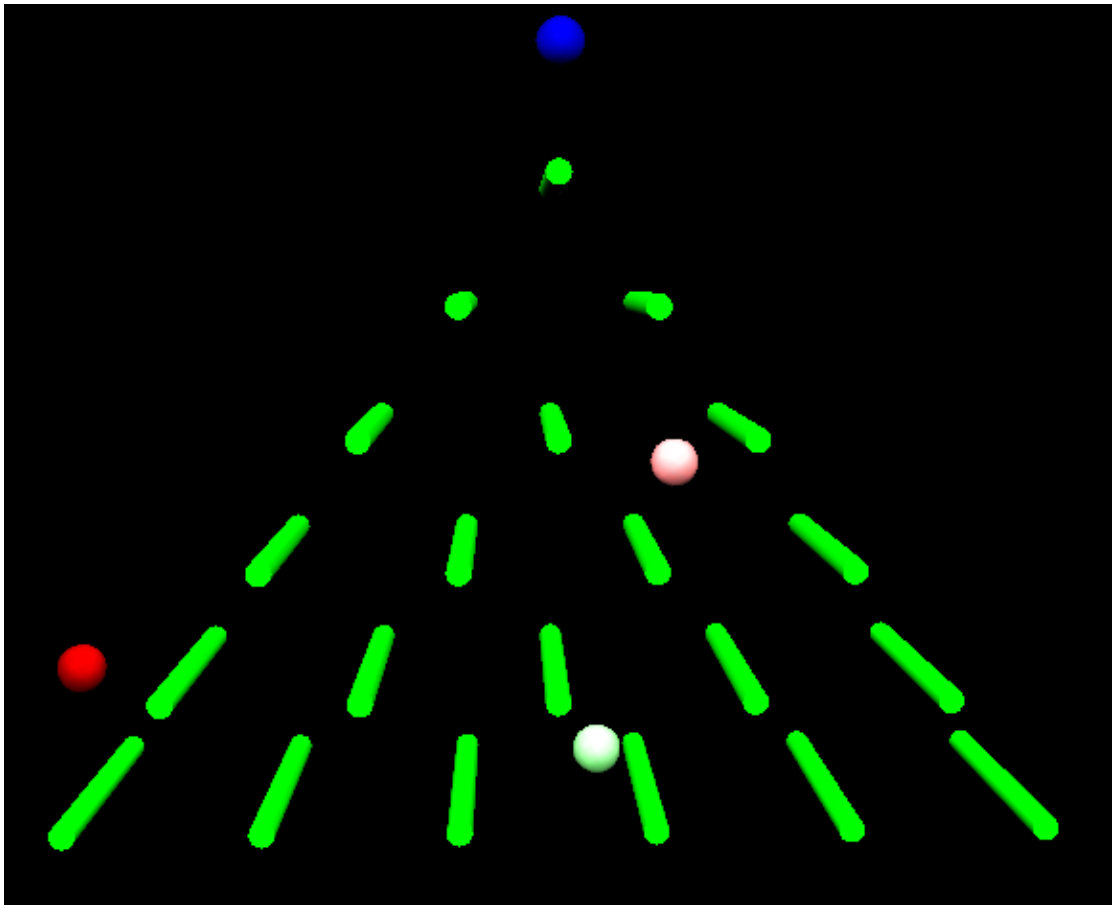
5.3.2 Kabelbaan

In dit voorbeeld hangt een blokje met een touw aan een kabel die schuin omlaag loopt: een zeer eenvoudige kabelbaan. Het blokje wordt bovenaan losgelaten, en versnelt vervolgens langs de kabel naar beneden. In dit voorbeeld worden zowel de bar-constraint (waarmee twee objecten met een touw verbonden kunnen worden, dus niet meer dan een bepaalde afstand van elkaar mogen bewegen) als de die point-to-curve constraint (waarmee een punt verbonden kan worden met een curve) van DynaMo gebruikt.

5.4 Galtonbord

In dit voorbeeld vallen een aantal balletjes op een pyramide van cylinders. Dit is een nabootsing van het klassieke Galton-bord, waarbij kogeltjes door eendriehoek van spijkers vallen. Bij iedere spijker kan zo'n kogeltje zowel naar links als naar rechts vallen. Als er een groot aantal kogeltjes door dit bord vallen onstaat er onderaan het bord (waar de kogeltjes eruit vallen) een normale verdeling: in het midden zijn de meeste kogeltjes gevallen, en naar de zijkanten toe is het aantal gevallen kogeltjes steeds minder.

Het Galtonbord in SmallWorld probeert dit gedrag te simuleren. Het is lastig om dit Galton-bord precies goed te krijgen, omdat er nogal veel parameters zijn: de grootte van de ballen, de grootte van de cylinders, de afstand tussen de cylinders, zowel horizontaal als verticaal, de hoogte vanwaar de ballen worden losgelaten, de zwaartekracht, de elasticiteit en de massa van de ballen, etc. Ook stuiteren de ballen minder willekeurig als in de echte wereld: als de ballen vanaf dezelfde positie beginnen te vallen stuiteren ze ook op precies dezelfde manier naar beneden. Dat laatste kan worden opgelost door een afwijking aan te brengen in de startpositie van de ballen, zodat ze iedere keer van een iets andere positie vertrekken. Op deze manier, en bij een goede keuze van de parameters, geeft dit Galton-bord een redelijk resultaat, maar levert geen perfecte normale verdeling van de ballen op.



6 Implementatiedetails

In dit hoofdstuk worden de details van de oplossingen van problemen die optraden bij de implementatie van het systeem behandeld.

6.1 Het maken van de Interface-laag

6.1.1 DynaMo

Voor het maken van de interface van DynaMo is gebruik gemaakt van een Perl-script. Met dit script zijn de header-files van DynaMo ingelezen en is vervolgens de interface gegenereerd.

Omdat Smalltalk niet overweg kan met de klassen en methoden van C++ was het voor de communicatie tussen Smalltalk en DynaMo nodig om een interface te genereren. Deze interface bestaat uit functies voor het aanmaken en verwijderen van objecten in DynaMo en voor het aanroepen van de publieke methoden van deze klassen. Omdat DynaMo een groot aantal klassen bevat is er voor het maken van de interface gebruik gemaakt van een Perl-script. Met dit script zijn alle header-files van DynaMo ingelezen. Het genereren van deze interface liep in drie delen:

- Het genereren van de header-files voor de interface, met hierin de declaraties van de functies in de interface
- Het genereren van de C++-files, waarin de functies uit de interface zijn geïmplementeerd
- Het genereren van de platformfuncties aan de Smalltalk-kant van de interface

Voor elk van deze stappen is het deel van het script voor het inlezen en parsen van de header-files van DynaMo gelijk gebleven, maar is het deel voor het genereren van de output aangepast.

Met behulp van de Perl-scripts is voor iedere constructor, destructor en publieke methode in de DynaMo-klassen een functie in de interface gegeneerd. Als naam voor iedere functie is gekozen voor de naam van de bijbehorende klasse (zonder de prefix DL_, die aan alle klasse-namen in DynaMo is toegevoegd), gevolgd door de naam van de functie, of door ‘New’ of ‘Destroy’ voor de constructor en destructor van de klasse. De functie ‘get_position’ in de klasse DL_Geo kreeg hierbij bijvoorbeeld de naam GeoGet_position, en de constructor van de klasse DL_Geo kreeg de naam GeoNew.

Bij het genereren van de interface traden in een aantal gevallen nog problemen op:

- **Overloading:** Omdat de functies in de interface moeten voldoen aan de C-standaard moet iedere functie in de interface een unieke naam hebben. Als een methode van een klasse op meerdere manieren kon worden aangeroepen (bijvoorbeeld: de constructor van een vector kan worden aangeroepen met drie getallen als parameters, of met een punt of een (andere) vector als parameter) moest voor ieder van deze gevallen een unieke naam worden gegenereerd voor de functie in de interface. Dit is opgelost door bij het eerste voorkomen van een naam deze naam op de normale manier te genereren (de naam van de klasse met de naam van de functie erachter), en door bij ieder volgend voorkomen van dezelfde functienaam een volgnummer aan deze gegenereerde naam toe te voegen (bijvoorbeeld: de eerste constructor van een vector heet VectorNew, de volgende VectorNew2, en zo verder voor alle andere constructors van vector).

- **Inheritance:** Bij inheritance binnen de klassen van DynaMo ontstonden ook problemen. Deze problemen hebben niets te maken met het genereren van de interface, maar omdat ze wel zijn ontstaan door de manier waarop de interface is gegenereerd worden ze hier behandeld. Het probleem is dat bepaalde klassen (bijvoorbeeld DL_Dyna) erven van andere klassen binnen DynaMo (DL_Dyna erft bijvoorbeeld van DL_Geo). Als er een instantie van DL_Dyna wordt gemaakt dan kunnen hiervan ook alle methoden die bij DL_Geo horen worden aangeroepen. Het probleem is dat deze methoden niet als methoden van DL_Dyna in de interface zitten: de methode *get_position* kan bijvoorbeeld voor zowel DL_Dyna als voor DL_Geo worden aangeroepen, maar alleen *GeoGet_position* is een functie in de interface, en *DynaGet_position* niet.

Dit probleem zou opgelost kunnen worden door bij iedere klasse die erft van een bepaalde klasse ook alle methoden van de klasse waarvan geërfd wordt aan de interface toe te voegen. Dit zou moeten werken, maar de nadelen hiervan zijn dat de interface hierdoor veel groter wordt: veel functies die in feite gelijk zijn zullen meerdere keren (met meerdere namen) aan de interface worden toegevoegd. Vooral in gevallen van multiple inheritance of van een hele serie klassen die van elkaar erft zou dit erg veel worden, en daardoor ook moeilijk te genereren. Omdat er voor de koppeling tussen Smalltalk en DynaMo een andere mogelijkheid was om dit probleem op te lossen is ervoor gekozen om deze extra functies niet te genereren.

Voor de koppeling van DynaMo met Smalltalk kon het probleem worden opgelost door gebruik te maken van inheritance in Smalltalk. Bij het genereren van de smalltalk-kant van de interface wordt de klassenstructuur van DynaMo (inclusief inheritance) overgenomen. Hierdoor kunnen van objecten ook de methoden van de bovenliggende klassen worden aangeroepen. Vanuit Smalltalk wordt dan vervolgens de correcte functie in de interface aangeroepen. Dit betekent bijvoorbeeld dat bij het aanroepen van *Get_position* van een Smalltalk-dyna de *Get_position* van de Smalltalk-geo gebruikt zal worden, en dat de interface van DynaMo dus correct wordt aangeroepen met *GeoGet_position.*

- **Afwijkende declaraties:** Op een aantal plaatsen in de headers van DynaMo waren een aantal functies op afwijkende manieren gedeclareerd. Het belangrijkste geval waren een aantal methoden waarvan de declaratie over meerdere regels uitgespreid was. In deze gevallen kon de functie in de interface niet met het Perl-script worden gegenereerd. In deze gevallen is ervoor gekozen om handmatig deze functies aan te passen en ze aan de interface toe te voegen. Ook traden er op een paar andere plaatsen fouten op, waarbij een aantal variabelen van functies niet goed werden ingelezen door het Perl-script. Ook waren sommige typen voor variabelen incorrect. Omdat ook dit een klein aantal gevallen betrof zijn deze handmatig gecorrigeerd.

Een aangepaste versie van het Perl-script is gebruikt voor het genereren van de interface in Smalltalk. Smalltalk maakt gebruik van platformfuncties voor het aanroepen van functies in een DLL. Voor iedere functie in de interface van DynaMo moest dus een bijbehorende platformfunctie in Smalltalk worden gegenereerd. Om in Smalltalk de klassenstructuur van DynaMo over te kunnen nemen is ook bij dit script uitgegaan van de header-files van DynaMo.

Het volledige gebruikte Perl-script met uitleg is te zien in appendix C.

6.1.2 SOLID

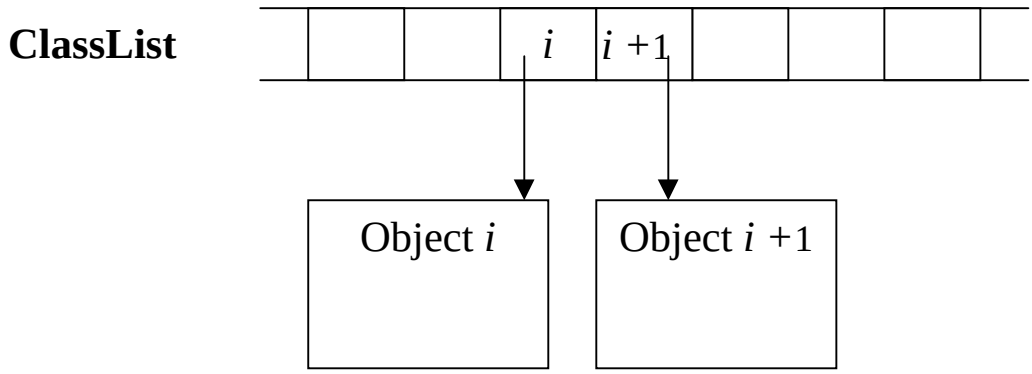
Bij het maken van de interface van SOLID was geen script nodig. Dit komt voornamelijk doordat de klassen en methoden van SOLID alleen intern binnen SOLID gebruikt worden en dus niet van buitenaf toegankelijk zijn. SOLID wordt aangestuurd via een (bestaande) interface van functies. Het aansturen van SOLID gebeurt voornamelijk vanuit DynaMo. Hierbij kan de bestaande interface van SOLID worden gebruikt. Vanuit Smalltalk wordt maar een beperkt aantal van de functies van SOLID gebruikt, voornamelijk de functies voor het toevoegen van objecten en voor het instellen van de positie en oriëntatie. Voor deze functies is een interface tussen SOLID en Smalltalk gemaakt. Omdat het om een klein aantal functies ging is dit handmatig gebeurt.

6.2 Interface

6.2.1 DynaMo

De interface tussen Smalltalk en DynaMo is zodanig opgezet dat alle functionaliteit die door DynaMo wordt geboden te gebruiken is vanuit Smalltalk. Hiervoor is voor iedere publieke methode in DynaMo een bijbehorende functie in de interface gegenereerd. Ook voor de constructor en destructor van iedere klasse is een functie gegenereerd. Bij het aanroepen van een constructor-functie in de interface wordt een nieuwe instantie van de betreffende klasse aangemaakt door het aanroepen van de echte constructor

van deze klasse. Het aanmaken van een nieuwe dyna (een dynamische object in DynaMo, waarop krachten kunnen werken) gebeurt bijvoorbeeld door vanuit Smalltalk de functie DynaNew aan te roepen. Deze functie zorgt er vervolgens voor dat er een nieuwe instantie van DL_dyna wordt aangemaakt, door het aanroepen van de constructor van deze klasse met de parameters die aan DynaNew zijn meegegeven. Het aangemaakte object wordt vervolgens opgeslagen in een lijst, ClassList. ClassList is een variabele van het type TList (een soort array, waarin pointers naar objecten kunnen worden opgeslagen) die in de DLL wordt aangemaakt voor het opslaan van instanties van alle DynaMo-klassen. Bij het opslaan in deze klassenlijst wordt de index (de positie van de opgeslagen klasse in de lijst) verkregen. Deze index wordt door de constructor-functie (in het voorbeeld de functie DynaNew) teruggegeven aan Smalltalk, en wordt in Smalltalk opgeslagen en gebruikt als het 'objectnummer'. Dit objectnummer wordt vervolgens als extra parameter meegegeven bij alle andere functie-aanroepen vanuit Smalltalk die met dit object te maken hebben.



ClassList is in feite een array met pointers naar objecten. Het 'objectnummer' van een bepaald object is de index van de pointer naar dat object in de ClassList.

Bij een aanroep van een bepaalde functie vanuit Smalltalk zal deze functie eerst het betreffende object uit de klassenlijst halen, waarbij gebruik wordt gemaakt van het objectnummer. Van dit object zal vervolgens de gevraagde methode worden aangeroepen, met de door Smalltalk meegegeven parameters. Indien deze methode een resultaat terug geeft zal dit resultaat worden terug gegeven aan Smalltalk. Smalltalk heeft dus geen rechtstreekse toegang tot de klassen van DynaMo of de instanties hiervan, maar kan alleen met deze klassen werken via de interface. Alle aangemaakte objecten van de gebruikte klassen worden opgeslagen in ClassList, en zullen alleen gebruikt worden in aanroepen vanuit de functies in de interface van de DLL of in aanroepen vanuit andere klassen binnen de DLL.

Het objectnummer van een DynaMo-klasse wordt ook gebruikt als parameter bij aanroepen van functies uit andere klassen. Als er bijvoorbeeld drie vectoren zijn aangemaakt, met objectnummers v1, v2 en v3, dan kunnen deze vectoren bij het aanmaken van een matrix als parameters worden gebruikt. Hiervoor moet de functie MatrixNew1 worden aangeroepen, met de nummers v1, v2 en v3 als parameters. Binnen de interface zullen dan de vectoren die bij v1, v2 en v3 horen uit de klassenlijst worden gehaald, en zal vervolgens de constructor van DL_Matrix worden aangeroepen met die vectoren als parameter.

6.2.2 SOLID

De interface tussen Smalltalk en SOLID staat het aanmaken en verwijderen van objecten vanuit Smalltalk toe. Voor ieder object in Smalltalk of DynaMo waarvoor botsingsdetectie moet worden uitgevoerd zal een object aan SOLID moeten worden toegevoegd. Deze objecten zullen binnen SOLID worden bijgehouden in SolidList. SolidList werkt op dezelfde manier als ClassList van DynaMo.

6.3 Botsingsrespons

Het uitvoeren van de botsingsdetectie gebeurt vanuit de interface van DynaMo. Iedere keer nadat in DynaMo de functie *dynamics* (voor het berekenen van de nieuwe posities en oriëntaties van alle

objecten op het volgende tijdstip) is aangeroepen wordt de functie *DT_Test* in SOLID aangeroepen. SOLID test nu of er ergens botsingen plaatsvinden. Indien dit het geval is zal SOLID een callback-functie in de interface aanroepen. Deze functie zorgt voor het afhandelen van de botsing.

De parameters waarmee SOLID de callback-functie aanroept als SOLID een botsing detecteert zijn: de objecten, de relatieve coördinaten waar de botsing plaatsvindt, en de penetration depth. Deze penetration depth is de kortste vector waarover een van de objecten verplaatst moet worden om de twee objecten elkaar te laten raken. Deze penetration depth kan gebruikt worden als een benadering van de normaal van de botsing. In de callback-functie wordt de botsing vervolgens afgehandeld door het toevoegen van een collision-constraint aan DynaMo. Deze collision-constraint is een onderdeel van DynaMo voor het afhandelen van botsingen: als er ergens een botsing plaatsvindt kan er zo'n collision-constraint aan DynaMo worden toegevoegd, waarbij als parameters de botsende objecten, de relatieve coördinaten van de botsingspunten op die objecten en de normaal van de botsing meegegeven moeten worden. De posities en relatieve coördinaten van DynaMo zorgt er vervolgens voor dat de botsing op een correcte manier wordt afgehandeld.

6.4 Oriëntatie

6.4.1 SmallScript3D en VRML

Voor het berekenen van de posities en oriëntaties van bewegende objecten in een 3D-wereld wordt gebruik gemaakt van DynaMo. De posities van objecten worden in DynaMo weergegeven met coördinaten. Bij het maken van een VRML-file van zo'n wereld kunnen deze coördinaten rechtstreeks worden overgenomen. Met de oriëntaties gaat dit minder gemakkelijk: er bestaan verschillende gangbare manieren zijn om oriëntaties te representeren. De meest gebruikte manieren hiervoor zijn **euler-hoeken** (de hoeken waarover een object om de drie standaard-assen (de x-, y- en z-as) geroteerd moet worden), de **as-hoek** notatie (waarbij een rotatie-as en de hoek die om die as gedraaid moet worden zijn gegeven), een **orientatiematrix** (die, vermenigvuldigd met de locale coördinaten van een punt van een object, de coördinaten van de 'gedraaide' versie van dit object teruggeeft) en **quaternions** (vier-dimensionale vectoren, waarmee het eenvoudig is om oriëntaties te interpoleren).

In DynaMo worden de oriëntaties van objecten gerepresenteerd door middel van een rotatiematrix; in VRML gebeurt dit door het aangeven van een rotatie-as en een hoek. De representatie van DynaMo moet dus omgerekend worden naar die van VRML. SOLID gebruikt quaternions voor de oriëntatie. Ook voor het doorgeven van oriëntaties van DynaMo naar SOLID moet er dus worden omgerekend.

De correcte methode voor de omrekening van oriëntatiematrices naar as-hoek-notaties [6] verloopt als volgt:

> Gegeven een as (x, y, z) waarvoor geldt dat $x^2 + y^2 + z^2 = 1$. Een rotatie met hoek a rond deze as kan dan worden beschreven door de volgende rotatie-matrix:
>
> $$R = \begin{pmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{pmatrix}$$
>
> waarbij $s = \sin(a)$, $c = \cos(a)$, $t = 1 - \cos(a)$
>
> Het berekenen van een as (*x*,*y*,*z*) en een hoek *a* uit een gegeven matrix gaat als volgt:
>
> $$a = \arccos\left(\frac{R_{11} + R_{22} + R_{33} - 1}{2}\right)$$
> $$(x, y, z) = \frac{(R_{23} - R_{32},\quad R_{31} - R_{13},\quad R_{12} - R_{21})}{2 * \sin(a)}$$

Met deze methode blijven nog enkele problemen over:

- Door afrondingsfouten binnen DynaMo kan het gebeuren dat, bij het berekenen van de hoek, het getal waarvan de arccosinus berekend moet worden iets groter dan 1 of iets kleiner dan -1 is. In deze gevallen zal de afrondingsfout gecorrigeerd moeten worden door dit getal op 1 of op -1 te zetten.
- Het kan gebeuren dat $\sin(a) = 0$. In dit geval is het onmogelijk om met deze methode een rotatie-as te berekenen. In dit geval moet de rotatie-as dus anders berekend worden. Als $\sin(a) = 0$ geldt dat $a = 0 \vee a = \pi$

  In het eerste geval, $a = 0$, is het probleem eenvoudig op te lossen: er is geen rotatie, dus iedere willekeurige rotatie-as voldoet. Het andere geval, $a = \pi$, is lastiger: in dit geval is er een rotatie van 180° om een onbekende as. Deze as kan niet op bovenstaande manier uit de rotatie-matrix worden gehaald, en zal dus op een andere manier bepaald moeten worden. Door het invullen van de waarden voor *s*, *c* en *t* bij $a = \pi$ in de matrix komt hier voor R_{11} de waarde $2x^2 - 1$ uit. Dit betekent dat in het geval $a = \pi$ geldt dat $x = \sqrt{\frac{R_{11}+1}{2}}$. Het berekenen van *y* en *z* gaat op precies dezelfde manier: $y = \sqrt{\frac{R_{22}+1}{2}}$ en $z = \sqrt{\frac{R_{33}+1}{2}}$ als $a = \pi$.

Met deze aanpassingen worden zowel de posities als de oriëntaties van objecten correct overgenomen vanuit DynaMo, en ook correct weergegeven in VRML.

6.4.2 SOLID

In SOLID worden oriëntaties gerepresenteerd door middel van quaternions. Omdat SOLID zelf is staat is om as-hoek notaties om te rekenen naar quaternions en de as-hoek notatie toch al berekend moet worden (voor de weergave in VRML) kan voor het instellen van de oriëntaties in SOLID gebruik gemaakt worden van de in DynaMo berekende as-hoek notatie.

7 Resultaten en Conclusie

In dit hoofdstuk worden de resultaten besproken. In het eerste deel wordt beschreven wat de nieuwe mogelijkheden van het systeem zijn, in het tweede deel wordt beschreven wat er in de toekomst nog aan verbeterd kan worden.

7.1 Bereikte resultaten

7.1.1 Beweging

De bibliotheken SOLID en DynaMo zijn gekoppeld aan Smalltalk. Hiervoor zijn zowel SOLID als DynaMo gecompileerd naar een DLL, en zijn de DLL's uitgebreid met een interface-laag voor de communicatie met Smalltalk. De interface-laag bestaat uit functies voor het aanmaken en vernietigen van klassen en voor het aanroepen van de publieke methoden van deze klassen.

Door deze koppeling van Smalltalk met SOLID en DynaMo kan er vanuit SmallScript3D volledig gebruik gemaakt worden van de functionaliteit van deze bibliotheken. Het is dus mogelijk om objecten in een wereld te definiëren en vervolgens aan deze objecten bewegingen toe te kennen. Dit gebeurt door het definiëren van krachten die op bepaalde objecten werken, of van krachten die een globaal effect hebben in een wereld (bijvoorbeeld zwaartekracht). Hiermee kan een natuurlijke omgeving worden gecreëerd, met objecten die realistische bewegingen uitvoeren en op een realistische manier op elkaar reageren.

De Animation Engine kan aan de hand van de initiële krachten en de ingestelde bewegingen nieuwe posities en oriëntaties berekenen door gebruik te maken van DynaMo. DynaMo kan hierbij, indien gewenst, botsingsdetectie laten uitvoeren door SOLID en de gedetecteerde botsingen verwerken. De resultaten van de berekeningen kunnen vervolgens in een VRML-file worden gezet, en dan worden bekeken in een VRML-browser. Hierbij worden zowel de posities als de oriëntaties van de objecten correct weergegeven.

Het uitvoeren van botsingsdetectie is mogelijk, en kan naar wens aan of uit worden geschakeld. De botsingsdetectie wordt vanuit DynaMo aangestuurd.

7.1.2 Communicatie en Architectuur

Omdat de communicatieklassen van PhilemonWorks nog niet beschikbaar zijn was het niet mogelijk om de gewenste client-server architectuur volledig te realiseren. Het systeem is wel zoveel mogelijk zodanig opgezet dat het omvormen naar een client-server architectuur door het toevoegen van deze communicatieklassen geen grote problemen oplevert.

7.1.3 Interactie

Omdat de client-server architectuur niet volledig is gerealiseerd was ook het testen van de interactie moeilijk. Het is wel mogelijk om interactie te simuleren door het toevoegen van krachten op bepaalde objecten op vooraf gedefinieerde tijdstippen.

7.1.4 SmallWorld

Met SmallScript3D zijn verschillende testwerelden gemaakt. Bij het maken van deze testwerelden lag de nadruk op het gebruiken van de nieuwe mogelijkheden die door de koppeling van SmallScript3D met SOLID en DynaMo beschikbaar zijn gekomen. De gemaakte werelden testen en demonstreren verschillende van deze mogelijkheden. Samen vormen deze testwerelden SmallWorld. Met

SmallWorld is de correcte werking van het systeem gedemonstreerd. Verder zijn de demonstraties uit SmallWorld een voorproefje van de vele dingen die met SmallScript3D mogelijk gemaakt zijn.

7.2 Toekomstig werk

7.2.1 Communicatieklassen

Om het systeem te kunnen gebruiken in een client-server architectuur moet het systeem uitgebreid worden met communicatie-klassen. Deze communicatieklassen zijn nog niet beschikbaar. Het belangrijkste gevolg hiervan is dat het niet mogelijk is om te testen met een client-server architectuur, omdat er geen mogelijkheid is om clients met een server te laten communiceren. Hierdoor kan er ook niet getest worden met meerdere clients.

Een gevolg van het niet beschikbaar zijn van de communicatie is dat het ook niet mogelijk is om realtime gebruik te maken van een wereld. SmallScript3D heeft geen eigen renderer, en moet dus gebruik maken van externe formaten als VRML. Zonder een manier om te communiceren met een VRML-browser is het niet mogelijk om realtime testen uit te voeren. Het testen van bewegingen op de server gebeurt door het onthouden van de posities en oriëntaties op bepaalde momenten, en door deze dan vervolgens achteraf te gebruiken voor het maken van een grote VRML, waarin alle bewegingen die hebben plaatsgevonden vastgelegd zijn.

Omdat de wereld niet real-time afgebeeld wordt is het ook niet mogelijk om interactie met zo'n wereld te hebben. De enige manier om interactie te testen is door tijdens het berekenen van de bewegingen in de server op bepaalde vooraf ingestelde momenten acties van een gebruiker te simuleren, bijvoorbeeld door voor korte tijd ergens een kracht toe te voegen. De reacties op deze acties zijn daarna te zien in de VRML-file, maar kunnen niet beïnvloed worden.

7.2.2 Editor

Voor het toekennen van de bewegingen aan objecten kan een editor gemaakt worden. Het zou dan een mogelijkheid zijn om een wereld (die gemaakt is met een andere editor, bijvoorbeeld 3D studio Max, of met SmallScript3D) in deze editor in te laden en dan van ieder object in de wereld de kenmerken in te stellen. Ook kunnen er krachten worden toegevoegd en objecten worden verplaatst en gedraaid. Zo'n editor zou de uitbreidingen van SmallScript3D ook toegankelijk maken voor mensen die geen kennis van programmeren hebben en dus niet in staat zijn om zelf de wereld in SmallScript3D te programmeren.

7.2.3 Scol

Op dit moment wordt binnen het Virtue-project gebruik gemaakt van de programmeertaal Scol. Scol is een programmeertaal die ontwikkeld is door Cryonetworks voor het bouwen, renderen en weergeven van 3D-werelden. Omdat Cebra gebruikt maakt van Scol in de eerste versie van Virtue was het in eerste instantie de bedoeling om Scol ook te gebruiken voor het bekijken van de in SmallScript3D gemaakte werelden en voor het testen van de uitbreidingen van SmallScript3D. Omdat het maken van uitgebreidere of geavanceerdere bewegingen met behulp van Scol zodanig lastig bleek te zijn dat dit buitengewoon veel tijd zou gaan kosten is er voor dit project besloten om Scol te laten vallen en te vervangen door VRML als taal voor het testen van de gemaakte 3D-werelden.

In de toekomst kan SmallScript3D alsnog gebruikt gaan worden in combinatie met Scol. Cebra heeft in het afgelopen jaar veel ervaring opgedaan met deze taal, en zal nu waarschijnlijk beter in staat zijn om problemen die ontstaan bij de koppeling van SmallScript3D met Scol op te lossen. Als deze koppeling tot stand kan worden gebracht kan SmallScript3D al snel gebruikt gaan worden binnen Virtue.

7.3 Conclusie

SmallScript3D is een veelbelovende taal: het is veelzijdig, compact, relatief eenvoudig te leren en gebruiken en kan gebruikt worden voor het maken van verschillende soorten 3D werelden. Door SmallScript3D uit te breiden met mogelijkheden voor beweging wordt de veelzijdigheid van deze taal vergroot. SmallScript3D is hierdoor uitstekend geschikt voor het gebruik binnen het Virtue-project. Met SmallScript3D kunnen op een eenvoudige en snelle manier grote, complexe werelden worden gemaakt, waarin animatie en interactie een belangrijke rol spelen.

SmallScript3D is ook goed geschikt om te gebruiken in een server. In SmallScript3D kan een 3D-wereld worden gedefinieerd. Dit script wordt vervolgens zowel gebruikt voor het maken van een VRML-file voor het initialiseren van de clients als voor het initialiseren van de server. Doordat alle clients hun wereld aanpassen aan die op de server worden hierdoor een hoop problemen voorkomen: de synchronisatie is eenvoudiger, doordat de clients niet onderling hoeven te communiceren. Hiervoor is wel een breedband-verbinding tussen de server en de clients noodzakelijk, omdat er bij grotere, complexere werelden grote hoeveelheden data verstuurd moeten worden.

Een ander belangrijk voordeel van deze opbouw is het feit dat alle berekeningen aan bewegingen en dergelijke op de server plaatsvinden. De clients hoeven zich hier dus niet zelf mee bezig te houden, en kunnen hun tijd besteden aan rendering en weergave, waardoor de kwaliteit en de snelheid van die weergave toe kunnen nemen, of aan andere dingen zoals video. Ook hierdoor is SmallScript3D uitermate geschikt voor gebruik binnen het Virtue-project: binnen het Virtue-project worden door bedrijven en studentenverenigingen 3D ruimtes gemaakt, die vervolgens op een server gedraaid kunnen worden. Hierop kunnen dan meerdere gebruikers (clients) tegelijk inloggen. Doordat de server alle berekeningen voor beweging en animatie uitvoert zijn er complexere werelden mogelijk dan wanneer alle bewegingen op de clients berekend moeten worden. Ook is het op deze manier eenvoudiger om de werelden op alle clients te synchroniseren, en om acties van een gebruiker direct zichtbaar te laten zijn voor andere gebruikers.

Appendix A: Perl

In deze appendix is één van de drie Perl-scripts die gebruikt zijn voor het genereren van de interface opgenomen. Het script dat hier is opgenomen is het script voor het genereren van de header-files van de DynaMo-kant van de interface. De andere twee scripts, voor het genereren van de C++-files van de DynaMo-kant en voor het genereren van de Smalltalk-kant van de interface, zijn grotendeels gelijk aan het hier volgende script. De verschillen zitten vrijwel uitsluitend in de delen die de code genereren; de delen voor het inlezen en parsen van de header-files van DynaMo zijn gelijk. Uitleg over de werking van het script is als commentaar (regels beginnend met #) in het script opgenomen. De perl-scripts zijn uitgevoerd met een bestandsnaam ‘filelist.txt’ als parameter. In filelist.txt waren de namen van alle header-files van DynaMo opgenomen.

```
# Perl-script voor het genereren van de header-file voor de interface
# van DynaMo

# Hieronder worden enkele variabelen gedeclareerd en geinitialiseerd
$in_class = 0;         # Zitten we binnen een klasse?
$in_public = 0;        # Zitten we binnen het publieke deel van een klasse?
$internal = 0;         # Zitten we binnen een deel van een klasse voor intern gebruik?
$classname = "";       # De naam van de klasse
$aantconstr = 0;       # Aantal constructors

# Ga naar de directorie ‘inc’, waar de Dynamo-headers ziten. Als deze directorie niet
# wordt gevonden, stop dan het programma met een foutmelding
chdir("inc") || die ("Kan directorie ‘Inc’ niet vinden!");

# Zolang we nog niet aan het einde van het bestand filelist.txt zijn ...
while (<>) {
       # Lees de volgende regel van filelist.txt en haal hier een bestandsnaam uit,
       # bijvoorbeeld ‘matrix.h’
       if (/(\w+)\.(\w+)/) {$FileName = "${1}"; $FileExt = "${2}"};

       # open het bestand met die naam, of stop met een foutmelding
       open(CURFILE,"${FileName}.${FileExt}") || die("Kan bestand
                                    ${FileName}.${FileExt} niet openen! ");

       # open een bestand voor de uitvoer, met naam ‘*Interface.h’, in het voorbeeld #
       ‘matrixInterface.h’
       open(F, ">gen\\${FileName}Interface.h");

       # Print de 'header' van de c++-header in het uitvoerbestand. In deze koptekst
       # worden enkele dingen gedefinieerd die nodig zijn om de DLL te compileren
       print F "#ifndef ${FileName}InterfaceH\n";
       print F "#define ${FileName}InterfaceH\n";
       print F "#include <vcl.h>\n";
       print F "#include \"${FileName}.h\"\n";
       print F "#ifdef __cplusplus\n";
       print F "extern \"C\" {\n";
       print F "#endif\n";

       print F "#ifdef _BUILD_DLL_\n";
       print F "#define ExtF __declspec(dllexport)\n";
       print F "#define Extclass class __declspec(dllexport)\n";
       print F "#else\n";
       print F "#define ExtF __declspec(dllimport)\n";
       print F "#define Extclass class __declspec(dllimport)\n";
       print F "#endif\n";

# Zolang we nog niet aan het einde van het bestand, bijvoorbeeld matrix.h, zijn ...
while (<CURFILE>) {
  # klassen met inheritance
  if (/^\s*class\s+(\w+)\s*:\s*public\s+(\w+)/) {
    $in_class = 1;     # We zitten nu in een klasse
    $classname = $1;   # De naam van de klasse is de eerste (\w+)
    $parent    = $2;   # De naam van de parent is de tweede (\w+)
    $haak = 0;         # Telt accolades

    # Print een commentaar-regel in de uitvoer
    print F "//class $classname extends $parent \n\n";
  } else {
    # klassen zonder inheritance
    # Voorkom ‘forward references’ zoals "class DL_bar;"
       if (/^\s*class\s+(\w+)[^;\w]+/) {
              $in_class = 1;               # We zitten nu in een klasse
              $classname = $1;       # De naam van de klasse is de eerste (\w+)
              $haak = 0;             # Telt accolades
              print F "//class $classname\n\n";
       }
  }
```

```
# Haal de prefix DL_ voor de klassenaam weg
if ($classname =~ /^DL_(\w+)/) {
      $classN = $1
      }

# Als we het woord ‘public’ aan het begin van een regel tegenkomen maken we
# ‘in_public’ 1. Als in_public=0 kunnen de methoden worden overgeslagen
if ($in_class && /^\s*public:\s*$/) { $in_public = 1; }

$comment=0;  # Als 1: de hele regel is commentaar, dus negeer de regel

# methoden na het commentaar 'for internal use ony' hoeven niet
if ($in_class && $haak==0 && $in_public && /internal/) {$internal=1};

# methoden na het commentaar 'for external use' moeten wel
if ($in_class && $haak==0 && $in_public && /external/) {$internal=0};

# commentaar-teken aan begin regel ?
if ($in_class && $haak==0 && $in_public && /^\s*\/\//) {$comment=1};

# Geen commentaar, niet intern, wel in een klasse, publieke methode?
if ( $in_class && $haak==0 && $in_public && $internal==0 && $comment==0 &&
    /((virtual\s+)?(\w+\*?\s+)?[~\w]+\([^\)]*\))/ ) {

# Een methode die een constructor of destructor kan zijn is gevonden
$parameterMode=0;    # Huidige item is geen deel van een parameter definitie
$firstTime=1;        # Nog geen returntype gevonden
$methodName="";
$returnType="";
@parameterList=();   # lijst van gevonden parametertypen
@parameterTypeNames=(); # lijst van gevonden parametertypen met ‘aangepaste’ naam
@parameterNames=();  # Namen van de parameters
$parameters=0;       # Aantal gevonden parameters

# We gebruiken ‘split’ om de regel op te delen in ',' '(' ')' en spaties en
# verwerken de resterende items één voor één.
# Bijvoorbeeld, de regel void functienaam(int a, int b) wordt gesplitst in:
#    void, functienaam, (, int, a, ',', int, b, ) .
foreach $item (split /([\(,\)\s])/, $1) {

      # Verwijder de triviale gevallen
      if ($item eq ""        )  { next; }   # skip lege strings
      if ($item =~ /\s/      )  { next; }   # skip spaties
      if ($item =~ /\)/      )  { next; }   # skip sluithaakjes
      if ($item =~ /virtual/ )  { next; }   # skip ‘virtual’
      if ($item=~ /\*.*/)       { next; }   # skip sterretjes
      # haakje openen: nu komen de parameters
      if ($item =~ /\(/      )  { $parameterMode=1; $nextIsType=1; next; }
      # komma: nu volgende parameter
      if ($item =~ /,/       )  { $nextIsType=1; next; }

      if (!$parameterMode) { # een methode-naam of een returntype
             if ($firstTime) { $firstTime=0; $returnType=$item; }
             else { $methodName=$item; }
             next;
       }

      if ($nextIsType) {    # een parameter-type
             if ($item =~ /^void$/) {$isVoid=1;} #een void zonder *: negeer
             else {  #voeg de parameter toe aan de parameterlijst
                    $parameters = $parameters + 1;
                    @parameterList = (@parameterList,"$item");
                    # Verwijder de * of de & als die aan het einde van de naam staat
                   if ($item =~ /\*$/) {chop($item)};
                   if ($item =~ /\&$/) {chop($item)};

                    # Als het item begint met DL_, en dus een Dynamo-type is
                    if ($item =~ /^DL_(\w+)/) {
                      $pname = $1 }  # haal de DL_ weg
                      else {$pname = ${item}};
                    if ("$pname" eq 'int') {}        # Integer is ok: skip
                    elsif ("$pname" eq 'Scalar') {}  # Scalar (double) is ok: skip
                    elsif ("$pname" eq 'boolean'){}  # Boolean ook ok: skip

                    # In andere gevallen: een Dynamo-klasse
                    # Voeg een nummer aan de naam toe, en maak dit een integer-type
                    # Dit integer-type wordt in de functie omgezet naar de juiste
                    # klasse, door deze uit de ClassList te halen
                    else {$pname = "${pname}Number"};
                      $pnum=1;
                      $tel=0;
                      # Kijk of deze naam+nummer al bestaat; zo ja: verhoog nummer
                      while ($tel<$parameters-1) {
                        $tester = $parameterTypeNames[$tel];
                        chop($tester);
                            if ($tester eq $pname) {$pnum++};
                           $tel = $tel+1;
                    };
             # Voeg het type aan de lijst toe
```

```
                @parameterTypeNames = (@parameterTypeNames,"${pname}$pnum");
                } #else (void)

                $nextIsType=0;

                next;
        }
    }  # Einde van foreach

    # Extra dingen voor destructors/constructors (deze hebben geen type informatie)
    if ($methodName eq "") {
        if ($returnType =~ /^\~.*/) {$methodName=$returnType; $returnType="destructor"}
        else {$methodName=$returnType; $returnType="constructor";}
        }

# Vanaf hier: de verzamelde gegevens afhandelen. We hebben:
# classname, returnType, methodName, @parameterList

if ($returnType eq "constructor") {
        # Genereer constructor
        #  Functieheader:
     print F "  ExtF int";
     print F " \u${classN}New";
         if ($aantconstr > 0) {print F "$aantconstr"}
         print F "(";  #Nummer de constructors indien meerdere
     $tel=0;
        while ($tel<$parameters) {
             $tempStr = "\u${parameterTypeNames[$tel]}";
             if ($tempStr =~ /^Scalar/) {print F "double $tempStr";}
               else {print F "int $tempStr";}
             $tel = $tel+1;
             if ($tel < $parameters) {print F ', '}
        }
     print F ");\n\n";
#  Einde Functieheader.
        $aantconstr++;
   }
elsif ($returnType eq "destructor") {
        # Genereer destructor
        #  Functieheader:
     print F "  ExtF int";
     print F " \u${classN}Destroy(int \u${classN}Number0);\n\n";
#  Einde Functieheader.
   }
else {  # Genereer standaard-functie: methode

#  Functieheader:
     print F "  ExtF ";

     # Print het returntype: double voor een Scalar, int voor de rest
     if ("\L$returnType" =~ /scalar/) {print F "double"} else {print F "int"};
     # Print de methode-naam: de klassenaam met daarachter de 'originele' methodenaam
     print F " \u$classN"."\u$methodName"."(";

     #Print alle parameters
     $tel=0;
        print F "int \u${classN}Number0";
        if ($tel < $parameters) {print F ', '}
        while ($tel<$parameters) {
             $tempStr = "\u${parameterTypeNames[$tel]}";
             # Bij parametertype Scalar: type is double; Anders: type is int
             if ($tempStr =~ /^Scalar/) {print F "double $tempStr";}
               else {print F "int $tempStr";}
             $tel = $tel+1;
             if ($tel < $parameters) {print F ', '}
        }
     print F ");\n\n";
#  Einde Functieheader.
  }
}

 # Controleer of we nog in het deel voor de publieke methoden zitten
  if ($in_public && /^\s*[protected:|private:]\s*$/) {$in_public = 0; $internal=0;}

  # Einde van klasse als accolade in de eerste of tweede kolom
  if ($in_class && /^\s{0,1}\}/) {
    $in_class = 0; $classname="-"; $in_public=0; $internal=0; $aantconstr=0;
  }

  # Hiermee worden de bodies van inline-methoden genegeerd
  if ($in_class && $in_public && /[^\{]*\{.*/) { $haak++; }
  if ($in_class && $in_public && /[^\}]*\}.*/) { $haak--; }

# Print de 'footer' van de c++-header
        print F "#ifdef __cplusplus\n";
        print F "}\n";
        print F "#endif\n\n";
        print F "#endif\n";
```

```
} #Einde van de while voor deze file: volgende file
} #Einde van de while voor filelist.txt: alle bestanden gehad
```

Appendix B: Gebruik van SOLID en DynaMo

De procedure voor het maken van een 3D-wereld die gebruik maakt van SOLID en DynaMo gaat binnen Smalltalk als volgt:

1. **Initialiseer DynaMo:** in deze stap wordt de wereld in DynaMo geïnitialiseerd. De acties die in deze stap worden uitgevoerd zij het initialiseren van het dynasystem, de constraint-manager, het kiezen van een integrator, het instellen van de stapgrootte (de tijd tussen twee 'frames'), en enkele andere dingen. In het voorbeeld in de appendix is een gedetailleerdere toelichting hiervan te vinden.
2. **Initialiseer SOLID:** in deze stap wordt de SOLID-bibliotheek geïnitialiseerd. De acties die in deze stap moeten worden uitgevoerd zijn het aanroepen van UseCollisions van DynaMo en SolidInit in SOLID. Hiermee wordt aangegeven dat er botsingsdetectie uitgevoerd moet worden en wordt SOLID geprepareerd.
3. **Toevoegen objecten:** in deze stap moeten alle objecten worden aangemaakt en worden toegevoegd aan SmallScript3D, SOLID en DynaMo. Het toevoegen van objecten is een herhaling van de volgende stappen:
    a. **Aanmaken SmallScript3D-object:** Dit maakt het object aan in de wereld op de server. Hierbij hoort op dit moment ook het aanmaken van variabelen om alle posities en oriëntaties in op te slaan om hiervan later een VRML-bestand te maken. In een client-server architectuur is dit niet meer nodig
    b. **Aanmaken DynaMo-object:** Voor ieder object op de server moet een object in DynaMo worden aangemaakt. Van dit object moeten de positie en oriëntatie worden ingesteld. Ook moeten enkele andere kenmerken van het object worden ingesteld, zoals de massa, de elasticiteit (die bepaalt hoeveel hoeveel van de energie er bij een botsing wordt geabsorbeerd, en dus hoeveel een object stuitert bij een botsing), wrijving en de inertiatensor (die aangeeft hoe gemakkelijk een object om zijn eigen as kan draaien). Bij het aanmaken van een object in DynaMo wordt het nummer van dit object binnen DynaMo, de Dynamo-ID, verkregen. Deze Dynamo-ID zal gebruikt worden bij alle verdere communicatie met DynaMo over dit object.
    c. **Aanmaken SOLID-object:** Van ieder object moet de structuur aan SOLID worden doorgegeven. Voor eenvoudige objecten zoals een balk of een bol kan dit door het aanroepen van een functie; van complexere objecten zal de structuur binnen SOLID handmatig moeten worden opgebouwd. SOLID heeft de structuur van de objecten nodig om botsingen te kunnen detecteren. Bij het aanmaken van objecten in SOLID moet de Dynamo-ID worden meegegeven. Het object krijgt een Solid-ID. Deze wordt door SOLID ook doorgegeven aan DynaMo, zodat DynaMo updates kan doorgeven naar SOLID.
4. **Genereer de VRML-file:** In een client-server architectuur kan nu de VRML-file voor voor het initialiseren van de wereld op de clients worden gegenereerd. Deze clients kunnen dan vervolgens contact zoeken met de Server. In de huidige architectuur wordt deze stap overgeslagen.
5. **Uitvoeren van bewegingen:** Het bouwen van de 3D-wereld is nu voltooid, dus kan de Animation Engine worden gestart. De Animation Engine zorgt voor het uitvoeren van de bewegingen binnen de wereld, en voor de afhandeling van interactie en dergelijke. De Animation Engine wordt in detail beschreven in sectie 4.3.

Appendix C: het maken van een 3D-wereld

In deze appendix wordt de procedure voor het aanmaken van een complete dynamische3D-wereld beschreven door het weergeven van een gedocumenteerd voorbeeld in SmallScript3D. Dit is het voorbeeld uit 5.2.3, waarin 50 ballen boven elkaar geplaatst worden en vervolgens op een vlak vallen. Uitleg over het script is opgenomenen in het commentaar, tussen aanhaalingstekens. Zie ook de algemene uitleg over SmallScript3D in sectie 2.1.1.

```
VeelVallendeBallen    " Dit is de naam van het script "

" Hieronder worden de gebruikte variabelen gedeclareerd. Deze hebben (nog) geen type "
|   dsystem result pos b bal rand aantBal colCol px py pz vloer vec teller showMes
    showMesAt AantMes x1 z15 box1 toucher time1 interp |

" Hier worden DynaMo en SOLID geinitialiseerd "
dsystem := self InitDynamo.
result := DL_Object call: 'SolidInit' types: #() withAll: #().

" Er wordt een nieuw DynaMo-punt gemaakt. Dit wordt later gebruikt als positie "
pos := DL_point new.
pos PointNew2: 0.0 with: 10.0 with: -7.9.

" De scenebuilder wordt geinitialiseerd "
b := SceneBuilder vrml.
b color: Color red.

" De 'container' voor de ballen wordt geinitialiseerd "
bal := OrderedCollection new.
aantBal :=50.  " Er vallen 50 ballen "

" De random-generator wordt geinitialiseerd "
rand := Random new.

" Hieronder worden alle ballen geinitialiseerd "
1 to:aantBal do:[:i |  " i is de teller, en loopt van 1 tot aantBal (in dit geval 50) "

      " Stel de kleur van de bal in op een willekeurige kleur. "
      b color: (Color red01:   (-0.5+(rand next*3))/2
                      green01: (-0.5+(rand next*3))/2
                      blue01:  (-0.5+(rand next*3))/2).

      " Stel 'pos' opnieuw in. 'pos' wordt de positie van de volgende bal. Iedere bal
        begint 20 hoger dan de vorige "
      pos Init: 0.0 with: 0.0 with: (20 + 10.0 * i).

      " Voeg een bal toe aan de ballen-collectie. De bal heeft type SmallWorldObject "
      bal add: SmallWorldObject new.

      " Initialiseer de zojuist toegevoegde bal op positie pos. In deze stap wordt ook
        de DynaMo-companion van de bal aangemaakt "
      (bal at:i) NewObject: pos.

      " Stel px, py en pz in op de coördinaten van pos. "
      px := pos GetX.
      py := pos GetY.
      pz := pos GetZ.

      " Voeg de positie en de oriëntatie van de bal toe aan zijn interpolators "
      (bal at:i) posInt add: px,py,pz.
      (bal at:i) orientInt add: 0,0,1,0.

      " Maak het SmallScript3D-object van de bal aan "
      (bal at:i) ss3d: (b add: (Sphere center: px,py,pz radius:2)).

      " Voeg de bal toe aan SOLID "
      self SolidAddSphere: (((bal at:i) companion) selfint) posX: px posY: py posZ: pz
                                                            radius: 2.0.
      " Stel de inertiatensor en de massa van de bal in "
      (bal at:i) inertiatensor: 10.0 Y: 10.0 Z: 10.0.
      (bal at:i) mass: 5.0.
].    " Einde van de loop: alle ballen geinitialiseerd "

" Stel 'pos' in op de positie van de vloer "
pos Init: 0.0 with: 0.0 with: -2.0.

" Creer de vloer, een SmallWorldObject "
vloer := SmallWorldObject new.

" Initialiseer de vloer op positie 'pos'. De vloer is een statisch object. De
  companion van de vloer wordt in deze stap aangemaakt "
vloer NewStaticObject: pos.

" Stel px, py en pz in op de coördinaten van pos. "
px := pos GetX.
```

```
py := pos GetY.
pz := pos GetZ.

" Stel de kleur in op groen en maak vervolgens het SmallScript3D-object van de vloer "
b color: Color green.
vloer ss3d: (b add: (Box origin: (px-100),(py-100),(pz-2)
                         corner: (px+100),(py+100),(pz+2))).

" Voeg de vloer toe aan SOLID "
self SolidAddBox: ((vloer companion) selfint) posX: px posY: py posZ: pz extX: 200.0
                                              extY: 200.0 extZ: 4.0.

" Maak een nieuwe vector aan, initialiseer deze met een negatieve z-waarde "
vec := DL_vector new.
vec VectorNew2: 0.0 with: 0.0 with: -3.0.

" Gebruik deze vector om de zwaartekracht in te stellen "
dsystem Set_gravity: vec selfInt.


" Einde van de DynaMo- en SOLID-initialisatie; Initialiseer ne de Animation Engine "
teller := 0.
tellerMax := 5000             " DynaMo zal 5000 iteraties uitvoeren "

" De volgende drie items bepalen de naukeurigheid van de VRML-interpolatie "
showMes := 0.
showMesAt := 20. " Iedere 20e stap worden alle pos. en orient. bijgewerkt "
AantMes := 0.

" Test of de teller kleiner dan tellerMax is "
[teller < tellerMax]

" Zolang dat zo is: voer de loop van de Animation Engine uit "
whileTrue:
        [" Voer nu ‘dynamics’ uit. In dynamics berekent DynaMo alle posities en
           oriëntaties in het volgende frame van de animatie "
         dsystem Dynamics.

        " Verhoog de update-teller "
        showMes := showMes+1.
        " Kijk of de pos. En orient. bijgewerkt moeten worden "
        (showMes = showMesAt)
        ifTrue: [" Zo ja: reset de tellers "
                showMes := 0.
                AantMes := AantMes+1.

                " En voeg voor alle ballen in de ballen-collectie ... "
                1 to:aantBal do:[:i |
                " de nieuwe positie ... "
                self DynamoGetObjectPosition: ((bal at:i) selfdyna)
                                                Interpolator: ((bal at:i) posInt).
                " en de nieuwe oriëntatie toe aan de interpolators "
                self DynamoGetObjectOrientation: ((bal at:i) selfdyna)
                                                  Interpolator: (bal at:i) orientInt .
                ]. " Einde loop: alle ballen bijgewerkt "

        " Geef een bericht weer op de standaard-uitvoer van Smalltalk "
        Transcript show: 'Dynamics '; show: teller printString; show: 'compleet'; cr.
        ]. " Einde van de ifTrue

        " Verhoog de teller "
        teller := teller+1.
]. " Einde van de Animation Engine "

" Sluit de DynaMo-dll: deze is niet meer nodig "
self CloseDll.

" Maak een nieuwe SmallScript3D-kubus aan. Deze wordt de ‘knop’ voor het starten van
de animatie in de VRML-file "
box1 := Box origin: 60,0,0 corner: 61,1,1.
b group:[
        b color: Color blue.
        b add: box1. " box1 is toegevoegd aan SmallScript3D als blauwe kubus "
        " Koppel de kubus aan een tochsensor "
        b add: (toucher := b newTouchSensor)].

" Voeg een tijdsensor toe aan SmallScript3D, met tijdsduur: 20 seconden. "
b add: (time1 := (b newTimeSensor) cycleInterval: 20).
```

```
" Start de tijdssensor als er op de blauwe kubus box1 wordt geklikt "
b whenTouching: toucher start: time1.

" Maak een Ss3d-interpolator-colectie aan "
interp := OrderedCollection new.

" Doe voor alle ballen in de ballencollectie: "
1 to:aantBal do:[:i |
        " Voeg een nieuwe positie- en een oriëntatie-interpolator toe aan ss3d. Gebruik
          hierbij de interpolators van de kubus "
        b add: (interp add: (b newPositionInterpolatorFrom:  (bal at:i) posInt)).
        b add: (interp add: (b newOrientationInterpolatorFrom: (bal at:i) orientInt )).

        " Koppel de interpolators aan de timer en de touchsensor "
        b whenFractionChanging: time1 rotate: ((bal at:i) ss3d) using:
                                                (interp at: (2*i)).
        b whenFractionChanging: time1 translate: ((bal at:i) ss3d) using:
                                                (interp at: (2*i-1)).
        ].

" Voeg twee lichtbronnen toe aan SmallScript3D "
b add:(Light at:100,100,100 lookingAt: 0,0,0).
b add:(Light at:100,-100,100 lookingAt: 0,0,0).

" Voeg een viewpunt (camera) toe aan SmallScript3D "
b add:((Camera at: 150,100,125 lookingAt:0,0,25) label: ' StandaardView ' ).
b navigationInfo examine.

" Schrijf alle informatie uit SmallScript3D weg naar een VRML-file "
b writeToFile: 'VeelVallendeBallen' useIndentation: true.
```

Referenties

[1] Virtue-site
http://www.virtue.tue.nl

[2] Website Cebra
http://www.cebra.tue.nl

[3] Website PhilemonWorks
http://www.philemonworks.com

[4] DynaMo
Bart Barenbrug, “Designing a class library for interactive simulation of rigid body dynamics”
Technische Universiteit Eindhoven, april 2000
http://www.win.tue.nl/~bartb/dynamo/

[5] SOLID
Gino van den Bergen, “Collision Detection in Interactive 3D Computer Animation”
Technische Universiteit Eindhoven, maart 1999
http://www.win.tue.nl/~gino/solid/
http://www.dtecta.com/

[6] Oriëntatie en omrekening Matrix naar Axis-angle
http://www.sct.gu.edu.au/~anthony/info/graphics/matrix.hints

[7] VRML-documentatie: The Annotated VRML 97 Reference
http://www.best.com/~rikk/Book/

[8] Perl
Randal L. Schwartz, Erik Olson, Tom Christiansen, “Learning Perl on Win32”;
O'Reilly, Augustus 1997