Sammanfattning

Ett gränssnitt för Internetbaserad VR. Tack vare den teknologiska utveckling som har skett under sista åren när det gäller mjuk- och hårdvara för persondatorer har det blivit möjligt att förena Internet och virtuell verklighet (VR) i Internetbaserad VR. Internetbaserad VR används för att framställa virtuella världar på Internet där användare kan förflytta sig genom tredimensionella scener och interagera med 3D-objekt. Dessa scener beskrivs med standard språket VRML (Virtual Reality Modelling Language).

Tyvärr råder stor brist på metoder för att göra dessa världar mer intuitiva och användarevänliga att hantera. Detta talar för att det behövs nya former och metoder för 3D-interaktivitet. I det här examensarbetet föreslår vi en interaktionsmodell som är inriktad mot att ge betraktaren en realistisk upplevelse av en fysiskt styrd virtuell verklighet. Resultatet har blivit en implementation av ett 3D-gränssnitt med stöd för fysikalisk simulering och direkt interaktion. Med hjälp av gränssnittet kan användare förflytta tredimensionella objekt samtidigt som fysikaliska fenomen som gravitation, friktion och kollision mellan objekten approximativt simuleras. Ett verktyg för att konvertera enkla VRML scener så att de kan manipuleras genom vår interaktionsmodell har också implementerats.

Abstract

An interface for Internet based VR. The technological advances in personal computer soft- and hardware have made it possible to combine Internet and Virtual Reality (VR) in Internet based VR. Internet based VR is a way to visualize virtual worlds where the user may navigate in a 3D scene and interact with 3D objects. These scenes are described with the standard language VRML (Virtual Reality Modelling Language).

Unfortunately, there is a lack of methods about how to make these virtual worlds more intuitive and easy to use. We propose an interaction model that gives the user a realistic experience of a physically constrained virtual reality. The result is a 3D interface with support of physical simulation and direct interaction. With it the user can translate 3D objects while gravitation, friction and collisions between objects approximately are simulated. One tool for converting from one simple VRML scene in order to could be manipulated with our interface has also been implemented.

Innehållsförteckning

1 Inledning

1.1 Bakgrund

1.1.1 Internetbaserad virtuell verklighet

Internet och ”Virtual Reality”, som vanligen översätts till virtuell verklighet (VR), har funnits i mer än trettio år nu. Trots detta var det inte förrän 1994 man började att använda dessa två teknologier tillsammans.

Internet har sitt ursprung i ARPANET-systemet från 1960-talet. Den amerikanska försvars-departementet hade utvecklat nya protokoll för att skydda det militära datornätverket. På 80-talet övertogs ARPANET av universiteten och blev Internet. Genombrottet kom med WWW med Mosaic, den första webbläsaren, som blev tillgänglig för allmänheten 1993. Sedan dess har Internet med sin explosionsartade tillväxt blivit ett av de viktigaste kommunikationsmedierna. Varje dag används Internet av millioner människor för att söka information, produkter och tjänster.

Virtuell verklighet har sitt ursprung från slutet av 60-talet.Virtuell verklighet är en teknik för att skapa en konstgjord datorgenererad verklighet. Användare upplever illusionen av att befinna sig i den virtuella världen. Denna illusion skapas med hjälp av speciell utrustning: vanligtvis ingår en huvudmonterad bildskärm för att titta på världen medan interaktion med objekt sker genom digitala handskar. Tekniken introducerades kommersiellt i 1985 och används idag mest för underhåll, simulering och träning av verkliga aktiviteter.

Tack vare den teknologiska utveckling som har skett under sista åren när det gäller hårdvara för persondatorer har det blivit möjligt att förena Internet och VR i Internetbaserad VR. Den beskriver tredimensionella världar som användare kan gå igenom och påverka. Genom Internet blir dem tillgängliga till allmänheten vilket skapar en helt ny genre för att visualisera och presentera information.

Virtuella världar kan bestå av 3D-objekt, text och multimedia. Navigering och interaktion i den virtuella världen brukar ske via tangentbord och mus. Ett särskilt språk kallat VRML (Virtual Reality Modelling Language) används för att beskriva dessa tredimensionell scener på ett liknande sätt som vi använder HTML (Hyper Text Mark-Up Language) för att framställa webbsidor på Internet. Scenerna visas på en vanlig datorskärm med hjälp av ett tilläggsprogram till de vanliga webbläsarna: Netscape Navigator eller Internet Explorer.

Man kan tillämpa Internetbaserad VR inom många olika område t ex:

- Kemister skulle kunna använda den för att visualisera komplexa molekyler i ett projekt. Dessa modeller skulle bli tillgängliga till alla deltagare oavsett vad de skulle finnas.
- Arkitekter skulle kunna skapa en VR-byggnad och bjuda allmänheten att besöka den virtuella platsen även innan den har börjats att bygga.
- I virtuell e-handel så att man skulle kunna gå omkring för att välja, plocka och betala varor.
- Lärare skulle kunna använda dem för distansutbildning. Solsystemet kunde visas för eleverna genom en forskningsresa i rymden omkring planeterna.
- Konstnärer skulle kunna skapa skulpturer och sälja dem till sina kunder i förväg.

- Man skulle kunna bygga underhållande platser där människor kunde träffas och umgås.

Exempel på specifika applikationer kan hittas i [1]. Men det är klart att det skulle kunna finnas många andra tillämpningar.

1.1.2 Internetbaserad VR –gränssnitt

Ändå är vägen till effektiva virtuella platser ännu mycket lång. Självklart finns det fortfarande begränsningar i teknologi, men det finns också en stor brist på metoder om hur man gör dessa världar/applikationer mer intuitiva och lättanvända.

En faktor som illustrerar detta är att det finns mer än 50,000 virtuella världar publicerade på Internet, men de flesta av dem saknar bra modell för simulering och interaktion.[2, 4] Följden blir att användare ofta upplever frustration och obehag [2].Detta händer eftersom det finns en mängd problem som försvårar utformning och implementering av tredimensionella gränssnitt. Här följer några exempel på dessa:

- För det första saknas generella riktlinjer om hur tredimensionell interaktion skall se ut[2, 4]. Jämför med Apples kända riktlinjer för 2D-fönstersystemet. [5]
- Det är inte möjligt att använda tidigare resultat från manniska-data-interaktion baserade på 2D fönsternssystemet. 3D-miljöer fokuserar ibland på att användare ska kunna förflytta sig och själv hitta det som är intressant för honom/henne istället för de mer tydliga uppgifterna i det 2D baserade gränssnittet. På samma sätt blir representationen av data i sig själv en viktig aspekt av gränssnittet. [2]
- Tidigare resultat från de klassiska VR systemen får inte heller användas. Då är användaren utrustad med digitala handskar istället för mus. Syftet med dessa är att generera ”immersion” och känsla av närvaro (eng. presence)[6] vilket inte är det primära syftet med Internetbaserad VR. [4]
- Internetanvändare sinmatningsenheter är mus och tangentbord. Dessa ger endast 2D data. Trots att det finns några studier i ämnet är det ändå inte klart hur effektiv en konvertering av 2D- inmatning till 3D -data är.[2]
- Specialisering av 3D-applikationer[3,7] försvårar återanvändning av interaktionstekniker mellan olika tillämpningar.
- Det finns psykologiska faktorer som förhindrar användare att beskriva hur han upplever tredimensionella gränssnitt [4]
- Dessa tredimensionella världar tillåter ibland flera användares samtidiga interaktion. [3]
- Till sist saknas det verktyg för att automatisera byggande av dessa gränssnitt.[2,7] Jämför med visuell programmering i Windows miljö som underlättar skapande av fönster och menyer.

Det går inte att lösa dessa problem utan forskning baserad på psykologi, mänskligt beteende och människan-dator-interaktion. Detta talar också för att det behövs nya former och metoder för 3D-interaktivitet. Dessa måste bedömas och utvecklas på ett iterativt sätt.

Tyvärr är många av de nuvarande ansträngningarna riktade på att återge bättre fotorealistisk tredimensionell grafik medan viktiga aspekter kring byggandet av tredimensionella gränssnitt i princip inte har utforskats[2,7].

1.1.3 3D-gränssnitt med stöd till fysikalisk simulering

En interaktionsmodell som skulle göra ett 3D-gränssnitt mer intuitivt baserar sig på hypotesen att det skulle bli lättare för användare att interagera med objekt om de presenterade fysikaliska egenskaper. Resultatet av användares interaktion skulle simuleras på ett sätt som påminner om verkligheten.

Den psykologiska grunden för den här metoden ligger i Gibbsons teori [8] om den ekologiska perceptionen. Enligt Gibbson har vi utvecklat vår perception och kognitiva förmåga i den naturliga miljön. Under hela människans evolution har funnits gravitation och kollisioner mellan objekt – därav vår medfödda förmåga för att uppfatta dessa fenomen. Ju mer vi använder i ett gränssnitt är vår medfödda och sent skaffade förmåga att iakttaga verkligheten, desto mera lätt och intuitivt blir det att använda.[2]

Att helt och hållet simulera interaktion i en virtuell värld på samma sätt som i verkligheten ligger över våra teknologiska möjligheter. Man kan tvivla på att detta någon gång blir möjligt, därför öppnas vägen för lösningar i form av metaforer baserade på vårt sätt att interagera med verkligheten.[2]

1.2 Syfte

I stort sätt kommer detta examensarbetet att visa hur man går till väga för att bygga ett 3D-gränssnitt baserat på fysiska egenskaper av tredimensionella objekt och hur man kan automatisera dess implementering.

I examensarbetet byggs en prototyp till en fysisk simuleringsmotor och en modell för simulering studeras. Tanken är att utveckla en simuleringsmotor för fysiska egenskaper som är inriktad mot att ge betraktaren en realistisk upplevelse av en fysiskt styrd virtuell verklighet. Simuleringen behöver alltså inte vara fysiskt korrekt så länge den upplevs realistisk. Detta antagande ska utnyttjas för att ställa upp en modell för simulering som bygger på kollisionsdetektering. Inledningsvis ska gravitation, friktion och tröghet implementeras.

Integrerad med simuleringsmotorn ska betraktarens personvybaserade navigeringsfunktioner utökas med metoder för att manipulera föremål i den virtuella världen med hjälp av mus och tangentbord. Operationer som ska stödjas är bl. a att kunna förflytta föremål.

Ett viktigt krav på simuleringsmotorn är att den på ett enkelt sätt ska gå att integrera med befintliga beskrivningar av virtuella världar. Det ska vara enkelt att bygga HTML-sidor med simulering och VRML integrerat. Befintliga VRML -beskrivningar av virtuella världar ska enkelt kunna uppdateras för simulering.

Tänkbara användningsområden för vår kombination av interaktion och simulering är visualisering av arkitektur / hus med möjlighet att röra sig i byggnaden och möjlighet att möblera om. Virtuella utställningar är ett annat tänkbart användningsområde, liksom virtuella affärer.

1.3 Om uppläggningen av rapporten

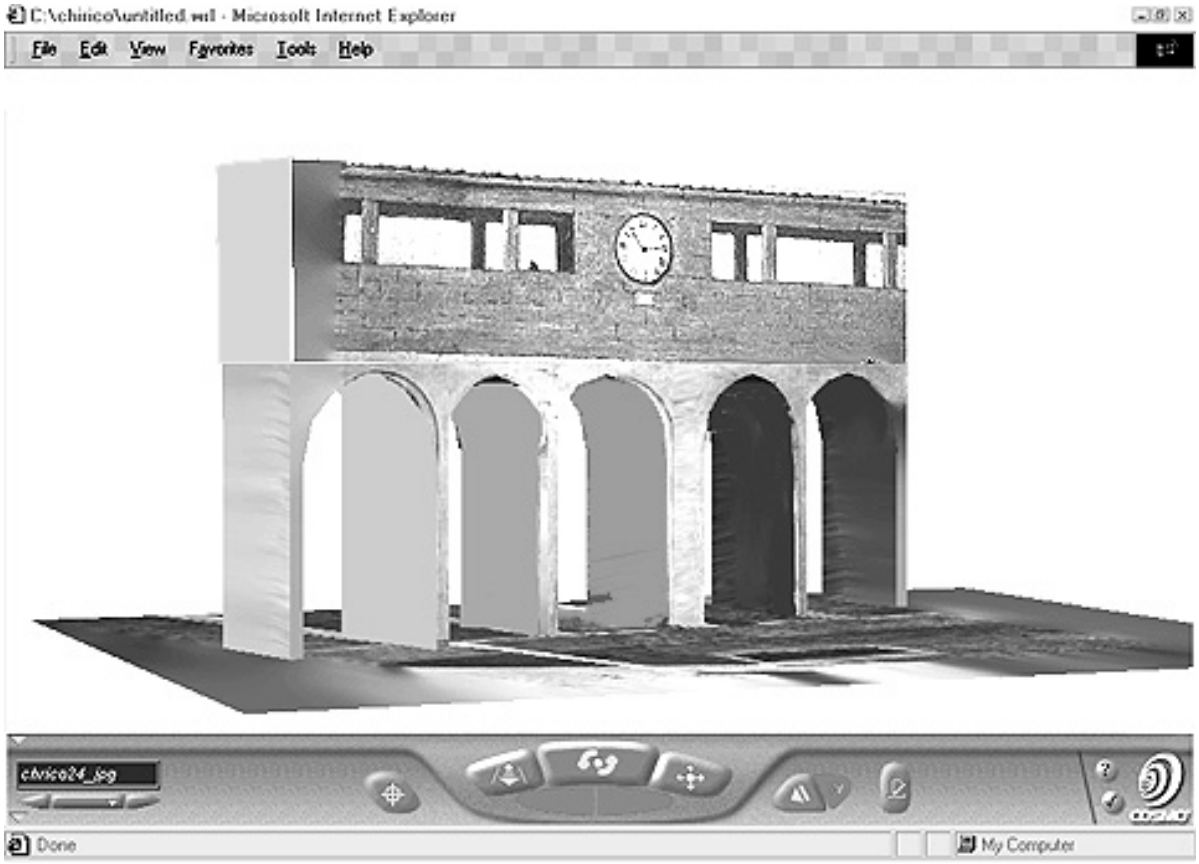
Den här rapporten är organiserad på följande sätt. I kapitel 2 introduceras VRML med en historisk översikt av språket och en beskrivning av VRML:s egenskaper som är viktiga för projektet. I kapitel 3 förklaras generellt hur programmeringspråket Java och VRML kan fungera tillsammans för att skapa dynamiska virtuella världar. I kapitel 4 beskrivs vår interaktionsmodell och hur den har implementerats. I kapitel 5 behandlas konvertering mellan enkla VRML-scener och scener där objekt kan manipuleras genom vår interaktionsmodell. Ett exempel som visar användning av motorn beskrivs i kapitel 6. Rapporten avslutas med en diskussion kring resultaten i kapitel 7.

2 VRML (Virtual Reality Modelling Language)

I detta kapitlet ges en introduktion till VRML (Vitual Reality Modelling Language) . Vi beskriver särskilt de av språkets egenskaper som är viktiga för att förstå det här projektet. Om man är intresserad att veta mer om VRML kan man hitta en detaljerad beskrivning i VRML:s specifikationsdokument [9].

VRML, som uttalas ”vermal”, är det standardfilformatet för att beskriva virtuella världar på Internet. Dessa världar kan bestå av 3D-objekt, text och multimedia. Det finns också möjlighet att lägga till animerade och interaktiva objekt.

Dessa tredimensionella platser kan upplevas med hjälp av ett tilläggsprogram till de vanliga webbläsarna. Exempel på tilläggsprogram är Cosmo Player från Computer Asociates Co. [10] och Contact från Blaxxun Co. [11].I figur 2.1 visas en VRML -scen med hjälp av Cosmo Player.



Figur 2.1 En VRML scen visas med Cosmo Player på Internet Explorer.

2.1 Historisk bakgrund

VRML:s uppkomst dateras till 1994. Allt började när Mark Pesce och Tony Parisi visade en webbläsare för tredimensionella scener på den första WWW (World Wide Web) mässan. Intresset var stort och efter mässan samordnades en diskussionsgrupp för att utveckla en 3D analogi till HTML .

I oktober samma år röstade diskussionsgruppen fram ett förslag till språket, som var baserat på Open Inventor-formatet från Silicon Graphics Co. Förslaget blev känd som VRML 1.0. Med hjälp av VRML 1.0 kunde man beskriva 3D-objekt och hyperlänkar på tredimensionella scener. Ändå saknades det viktiga funktioner för att kunna skapa animationer, interaktivitet och multimediaintegration.

Med tanke på att vidareutveckla och tillägga nya mekanismer till språket bildades VAG (VRML Architecture Group).VAG arbetade fram ett antal krav avseende språkets funktionalitet. Därefter presenterades sex företag sina förslag för att fullfölja dessa krav. VAG valde förslaget från Silicon Graphics Co. Det kallades ”Moving Worlds” och blev officiellt som VRML 2.0 i januari 1996. VRML 2.0, i sin nuvarande version, antogs som standard med namnet VRML 97 och det specificeras i ISO/IEC CD 14772 [9].

2.2 VRML:s beskrivning

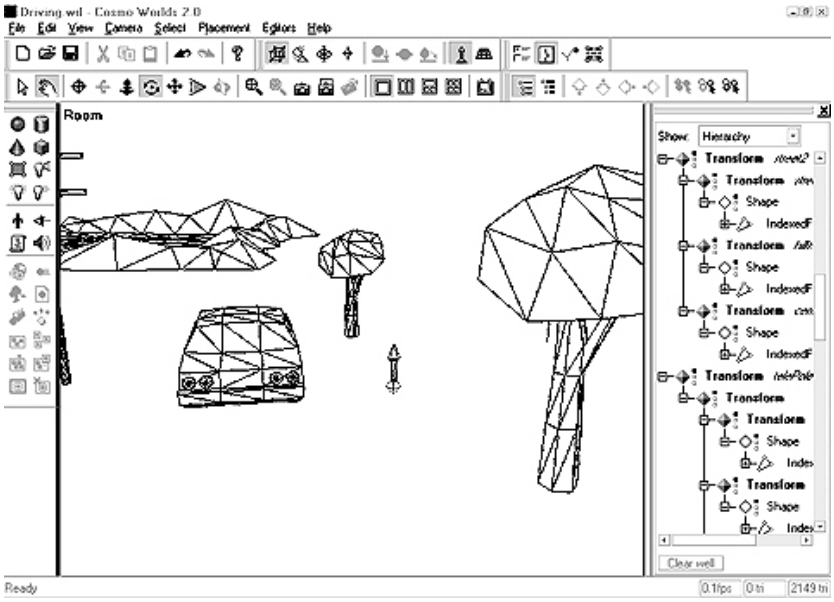
Språket innehåller bland annat följande egenskaper:

- Användning av enkla textfiler för att beskriva scener.
- Byggande av en scengraf för placering av 3D-objekten.
- Interaktiva scener med hjälp av händelser och sensorer.
- Mekanismer för att infoga scener i en scen.
- Egendefinierade typer för inkapsling och återanvänding av data.

2.2.1 VRML-filer

VRML-filer har tillägget `.wrl` (fr. eng. world). Språket är teckensbaserat dvs filer innehåller endast text och behöver inte kompileras. Därför kan man skriva sina egna VRML-filer med vilken texteditor som helst till exempel Microsofts Notepad.

Emellertid är VRML-filer vanligtvis genererade med hjälp av en modellerare. En modellerare är en applikation där man kan skapa en VRML-scen utan att behöva skriva kod. I stället kan man lägga till, modifiera och ta bort scenens element med hjälp av ett grafiskt gränssnitt. En av de mest använda modelleraren är Cosmo Worlds från Computer Asociates Co.[12] (bild 2.2) Man kan också använda kända program för att skapa tredimensionell grafik som 3D Studio Max från Kinetix Co.[13] och TrueSpace4 [14] från Caligari Co. Dessa program kan konvertera tredimensionella scener från egna format till VRML formatet.

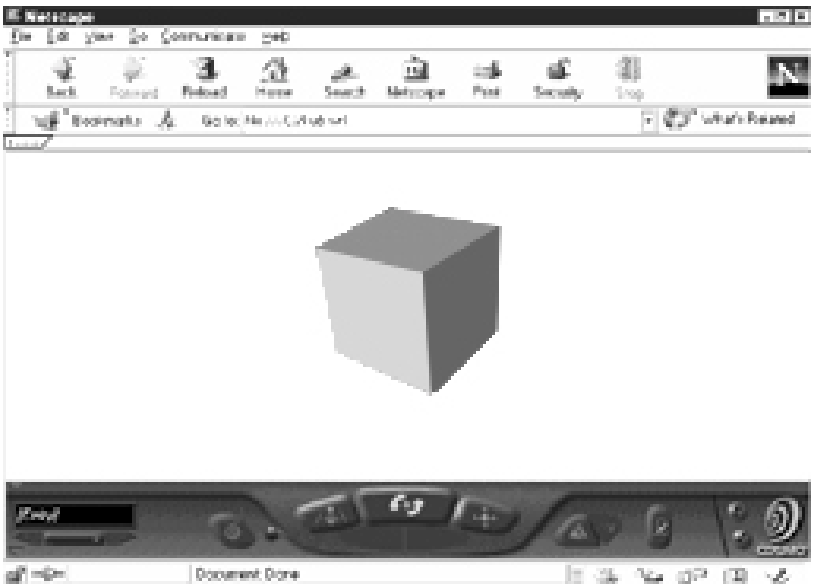


Figur 2.2 Cosmo Worlds användaregränssnitt

a)

```
#VRML V2.0 utf8
# En blå kub

Shape{
appearance Appearance {material Material {
                          diffuseColor 0 0 1} }
geometry Box{ size 1 1 1   }
      }
```

b)



Figur 2.3 a) En VRML-fil som beskriver en scen. Scenen består endast av en blå kub **b)** Scenen visas med Cosmo Player.

Ett exempel på en VRML fil visas i figur 2.3 a).Första filens linje är filhuvudet som bland annat anger filens VRML- version och vilken standard som används för att formatera texten: utf8 (fr. eng. universal character set). Som i andra programmeringsspråk är det möjligt att lägga egna kommentar till filen. Dessa föregås av '#' symbolen.

Den andra delen av filen beskriver en VRML scen. För detta används alla element som finns hos ett vanligt grafiskt språk: vypunkter, transformer, 3D-geometriska primitiver, ljussättningar etc. Dessa element beskrivs i VRML med hjälp av noder (`nodes`).

2.2.2 Noder

Språket definierar 54 typer av noder med olika attribut och funktioner. En typ av nod kan till exempel beskriva en geometri hos ett objekt, en annan typ kan beskriva ljudeffekt, en annan scenens ljussättning och så vidare. En nod deklareras med sitt typ

Varje nod lagrar sina parameter, som är kallade fält (`fields`) och händelser (`events`). Fälten anger egenskaper hos en nod. Händelserna är egentligen meddelande som noden kan sända eller ta emot. En nod kan också ha ett namn på scenen för att kunna referera direkt till den. För detta används ordet `DEF`.

```
DEF <nodens namn> <nodens typ> { fält händelser }
```

Figur 2.4 Nodens syntax

2.2.3 Fält och fältsvärde

Det finns två slags fält: publika (`exposedField`) och privata (`field`).Ett publikt fält innehåller ett värde som kan bli läst och ändrat utifrån noden. Detta inte tillåts i de privata fälten. Värdet som tilldelas till ett fält kallas för fältvärde. Det är viktigt att veta att ett fältvärde kan också vara en annan nod eller en grupp av noder.

Det finns 20 olika typer av fältvärde till exempel:

- `SF3Vec`: representerar en tredimensionell flyttalsvektor `x y z` som kan användas för att ange de X, Y och Z -koordinater för placering av ett objekt

- `SF3Rot`: representerar en 4-dimensionell flyttalsvektor: `x y z a` som kan användas för att beskriva en rotation omkring (x, y, z) axel med vinkeln lika med a .

Ett fältet deklareras först med sitt namn.(fig. 2.5) För att skriva dem delar man på enkla (SF) och multipla fältvärde (MF). Multipla fältvärden skrivs som en lista av värde inom ”[ ]” parenteser. Om fältvärdet är enkelt får man låta bli att skriva dessa parenteser.

```
<fältets namn > [<fältsvärde> …]
```

Figur 2.5 Fältets syntax

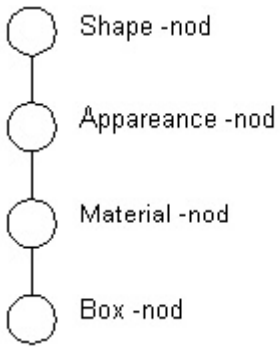
För att resumera allt hittills kan vi gå tillbaka till figur 2.3. Där ser vi en scen som innehåller tre noder: Första nod är av typ `Shape` och används för att skapa ett 3D-objekt, en kub i det här fallet. För att beskriva formen och utseendet hos objektet har `Shape` -noden två fält kallade `appearance` och `geometry`.

- `appearance`-fältet innehåller en nod av typ `Appareance` som anger utseende hos objektet och tillskrivs ett värde till sitt `material`-fältet i form av en nod av `Material`-typ. Med hjälp av `Material`-noden kan till exempel objektets färg specificeras. Färgen har angivits här med det RGB -värdet i det `diffuseColor`-fältet.

- `geometry` -fältet ansvarar för 3D-objektets geometri. Det innehåller en nod av typ `Box` och representerar en låda. `Box`-noden har ett fält med namn `size.` Det anger bredd, höjd och djup hos lådan:`1 1 1`. En VRML -enhet representerar en meter.

2.2.4 Scengraf

Som står tidigare kan vissa noder innehålla andra noder i sina fält. Detta skapar en hierarkisk placering mellan noder i scenen. Denna hierarkin beskrivs vanligtvis med en graf kallad scengraf. En scengraf som motsvarar till scenen med den blåa lådan visas i figur 2.6. På scengrafens representation använder man cirklar för att symbolisera noder och linjesegment för att visa nodernas position i hierarkin.



Figur 2.6 En scengraf som motsvarar scenen i figur 2.3

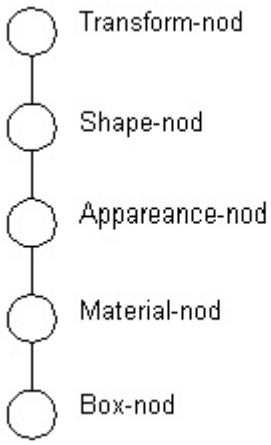
2.2.5 Placering av objekt.

Man använder `Transform`-noder för att kunna placera ett objekt på scenen. `Transform`-noder definierar ett lokalt koordinatsystem för alla noder som ligger i ett fält kallat `children`. Eftersom varje `Transform`-nod kan innehålla andra `Transform`-noder som i sin tur definiera var sitt lokala koordinatsystem skapas en sammanläkning av olika transformationer för att placera ett objekt.

Det enklaste sättet att placera ett objekt är att använda endast en `Transform`-nod. Vi ska anta att vi vill placera den blåa kuben på scenens XYZ -globala koordinater lika med (2,3,2). För detta följs följande steg:

- Först läggs en `Transform`-nod till scenen. Detta skapar ett lokalt koordinatsystem.
- Sedan translateras den lokala koordinatsystemets origo till globala koordinater (2,3,2). Detta anges med hjälp av `Transform`-nodens fält `translation`
- Slutligen i det `Transform`-nodens fält `children` lägger vi den `Shape`-noden med sina underliggande noder. Då kommer att placeras kuben i den lokalkoordinatsystems origo dvs (2,3,2).

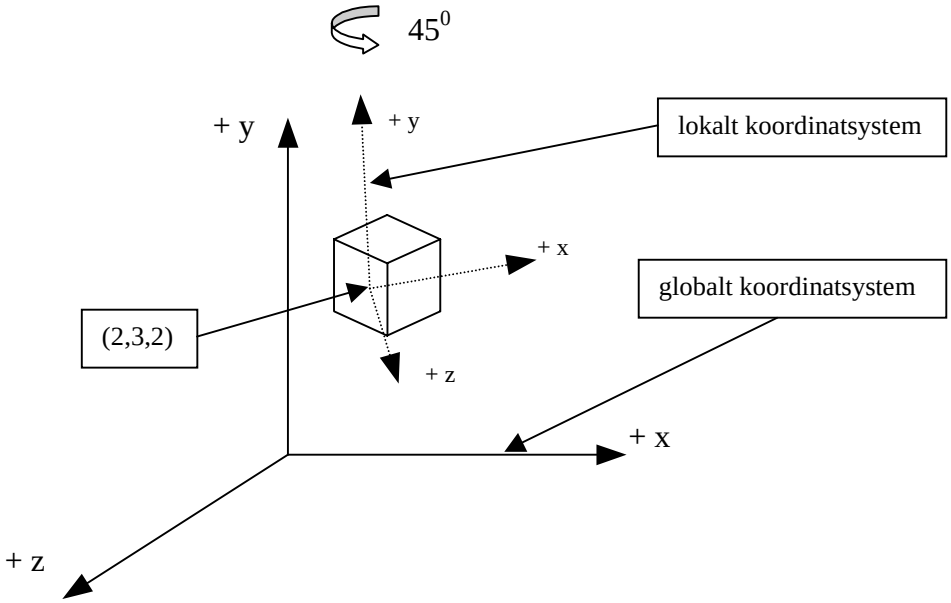
Scengrafen visas i figur 2.5.



Figur 2.5 Scengrafen för placering och orientering av en kub

På samma sätt som `translation`-fält anger den lokala koordinatsystemens origo, finns ett fält som anger dess orientering. Fältet har namnet `rotation` och med hjälp av det kan vi ange kubens orientering i scenen. Detta representeras grafiskt i figur 2.5 . Lägg märke till att koordinatsystemen är högervridna i VRML.

Inom koden som finns i figur 2.6 har avsiktligt inte angivits `diffuseColor`-fältvärdet.(jämför med figur 2.3). Om ett fält inte specificeras på en scen antas fältets fördefinerade värde (`fields defaut values`). Fördefinerade fältvärdet av `diffuseColor` -fältet är det RGB värdet 1 1 1 som är den vita färgen. Därför kommer att visas en vitt kub i scenen. De fördefinerade fältvärdena för varje fält och nod kan hittas i VRML:s specifikation [8].



Figur 2.6 Global och lokal koordinatsystem för placering av lådan i scenen

```
#VRML V2.0 utf8
# tranlation och rotation av en kub
# vinkel 0.785 radianer = 45 grader


Transform {
          translation 2 3 2
          rotation    0 1 0 0.785
          children    [ Shape {
                        material  Material{ }
                        geometry  Box  { size 1 1 1 }
                                }
                      ]
          }
```

Figur 2.7 VRML-fil som beskriver en scen där translateras och roteras en kub

2.2.6 Händelser och kopplingar

En av de mest användbara VRML:s egenskaper är möjlighet att ändra nodernas fältvärde med hjälp av händelser. Det finns två typ av händelser:

- Inhändelse (`eventIn`) : Definierar vilket fältvärde ska ändras när en nod får en händelse.
- Uthändelse (`eventOut`): Definierar vilket meddelande en nod ska sända när den får en händelse.

```
eventIn    <händelsens typ> <händelsens namn>
eventOut   <händelsens typ> <händelsens namn>
```

Figur 2.8 Händelsens syntax

Varje nod har sina egna ut- och inhändelser. Dessutom är alla publika fält en sammansättning av fältvärde, ut och inhändelser. Man har tillgång till dessa fältets händelser med referenser i form:

- `set_<namn på fält>` för inhändelser.
- `<namn på fält>_changed` för uthändelser.

Händelser används för att sända och ta emot värde mellan noder. När en nod får en händelse tillhörande till ett publikt fält, ändras sitt värde till det inkommande värdet. För att sända och ta emot händelser upprättas en koppling (`ROUTE`) mellan en uthändelse från en nod till inhändelse i den andra noden

```
ROUTE  <noden1 namn>.<fälts namn._changed /uthändelse >
TO     <noden2 namn>.<set_.fälts namn /inhändelse >
```

Figur 2.8 Kopplings syntax

Handelser och kopplingar används vanligtvis för att tillåta användare interagera med objekt som finns på scenen.

2.2.7 Interaktion i VRML

I VRML finns det speciella noder kallade sensornoder som kan registrera musens rörelse och generera händelser. Dessa händelser har information om t. ex. var musens markör befinner sig. Genom att skicka händelser till andra noder kan vi ändra objekts position .

Även om det finns nio typer av sensornoder använder vi mest i det här projektet en nod av typ `PlaneSensor`. En `PlaneSensor`-nod kan registrera när man klickar på ett objekt, drar ett objekt med musen eller släpper musens vänstra knapp.

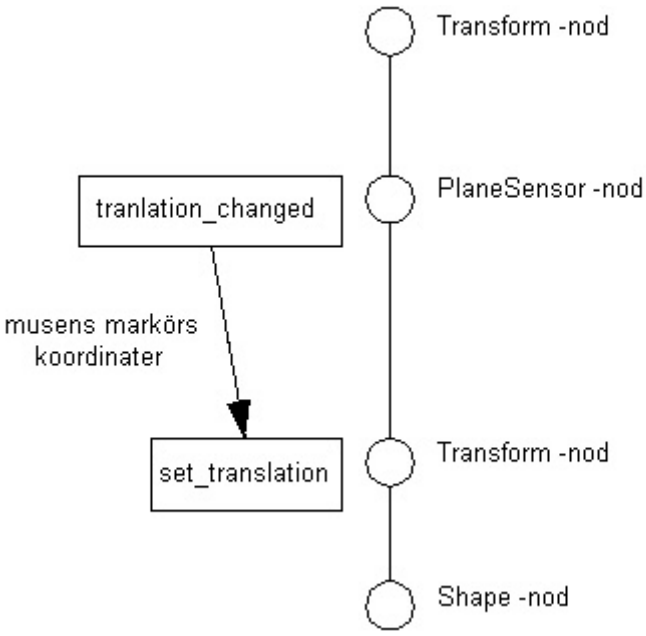
Objektet, som finns beskrivet i en `Shape` -nod, och `PlaneSensor`-noden måste ligga i ett gemensamt koordinatsystemet för att generera dessa händelser. När objektet dras genererar `Planesensor`-noden en uthändelse med ett värde som motsvarar musens markörs koordinater. Dessa koordinater tillhör till ett plan som ligger parallellt till det lokala XY -planet på det gemensamma koordinatsystemet där objektet och `PlaneSensor` -noden finns. Denna uthändelsen kallas `translation_changed.`

I figur 2.9 visas en scen där man använder en `PlaneSensor`-nod för att dra en låda med musen. Den sista linjen på filen anger att det finns en koppling mellan händelsen från `translation_changed` på den `PlaneSensor`-noden till `set_translation` på den `Transform` -noden. På det sättet skickar vi musens koordinater till `Transform`-nodens `translation`-fält . När inhändelser kommer ändras `translation`-fältvärdet. Resultatet blir att vi förflyttar det lokala koordinatsystemet och detta upplevs som om vi drar lådan med musen.

På scengrafer ska vi representera in- och uthändelser med fyrkanter och kopplingar mellan noder med en pil mellan dem. En scengraf som motsvarar scenen visas i figur 2.10.

```
#VRML V2.0 utf8
# Interaktiv scen

Transform {
          DEF theSensor PlaneSensor { }
          children [
          DEF theBoxTrans Transform{
                    children[
                         Shape { geometry Box {size 2 2 2}}
                           ]
                                 }
                   ]
          }

ROUTE theSensor.translation_changed
TO    theBoxTrans.set_translation
```

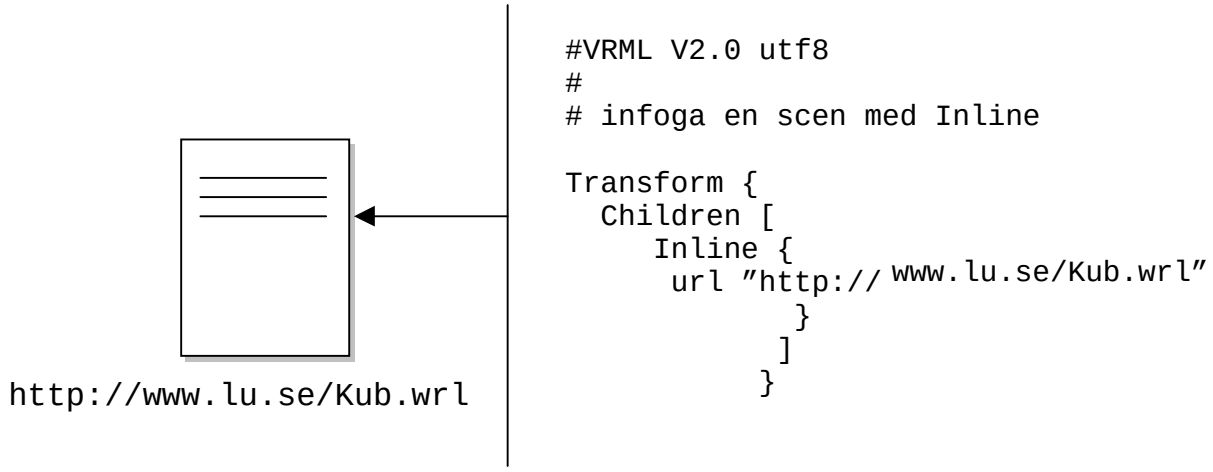
Figur 2.9 Interaktiv scen där man drar en kub med muspekaren



Figur 2.10 Scengrafen för placering och orientering av en kub.

2.2.8 Att infoga andra scener till en scen.

En stort fördel som VRML har, är att kunna infoga till en scen andra scener som kan finnas lagda på Internet. Dessa importerade scener ritas bara när det behövs vilket optimerar renderingen (återgivandet).



Figur 2.11 Användning av en `Inline` -nod för att infoga en scen inom andra. Scenen till höger infogar en annan scen som beskrivs i en annan fil

För att infoga andra scener till en scen används en `Inline`-nod. I en `Inline`-nod anges den filens url adress på det `url`–fältetet . Scenen till höger i figur 2.11 föreställer samma scen som i figur 2.3 men nu med hjälp av en `Inline`-nod.

2.2.9 Egendefinerade typer.

Även om det finns 54 nodtyper får man definiera sina egna nya typer av nod. Den typ man konstruerar måste ha

- Ett nytt namn.
- En gränssnitt som består av alla fält, in och uthändelser son den nya typen kommer att innehålla. Man måste också ange varje fälts typ, fältvärdes typ och fördefinerade värde.
- En intern struktur till vilken man kan relatera dessa fält.

För detta används en `PROTO`-deklaration (fr. engelska prototyp). Syntaxen visas i figur 2.12.

```
PROTO ny_typs_namn [

fältets_typ fältsvärdets_typ fältsvärdes_namn fördefinerade_värde

händelser

 ]

{ Noder Kopplingar }
```

Figur 2.12 Egendefinerade typer(`PROTO`) -deklarations syntax

Ett exempel visas i figur 2.13. Scenen består av tre kuber som kan dras med musen. De ligger respektive på koordinater (1, 1, 1), (2, 2, 2) och (3, 3, 3). För detta har vi relaterat fältet kallat `thePosition` till `Transform`-nodens `translation`-fält i den interna deklarations struktur (används ordet `IS`).

Efter att man har definierat en egen typ med en `PROTO`-deklaration kan den nya typen användas transparent i samma filen. En deklaration kan också refereras från en annan fil med hjälp av en `EXTERNPROTO`-deklaration. En `EXTERNPROTO`-deklaration ger information om den nya typen och var filen med `PROTO`-deklarationen finns.

```
#VRML V2.0 utf8
# Definierar en ny typ av nod kallad InterBox

PROTO InterBox    [
exposedField  SFVec3f thePosition  0 0 0
]
{
Transform {
          translation IS thePosition
          children [
              DEF theBoxTrans Transform {
                                    children [
                                          Shape {
                                          geometry Box { size 1 1 1}
                                                 }
                                       ]
                                     }
              DEF theSensor PlaneSensor { }
                                                        ]
                                  }
ROUTE theSensor.translation_changed
TO theBoxTrans.set_translation
}
# Nu kommer den själva scenens beskrivning
Transform {
          children  [
                  InterBox  { thePosition 1 1 1 }
                  InterBox  { thePosition 2 2 2 }
                  InterBox  { thePosition 3 3 3 }
]
}
```

Figur 2.13 Scenen föreställer tre lådor som kan dras med musen. Interbox är en egendefinerade nodtyp. Fältet `thePosition` representerar placering av lådor relativ det global koordinatsystemet

Detta har blivit en kort beskrivning på VRML. Språket är egentligen mycket större och jag hänvisar till specifikationen för den som vill läsa mer. Det finns också böcker som förklarar det mesta om språket [15 ,16]. På nästa kapitel kommer vi att beskriva hur VRML kan integreras med programmeringsspråket Java för att skapa dynamiska scener.

3 VRML och Java

3.1 Java

Java är ett modernt programmeringsspråk utvecklat av Sun Microsystems. Det är ett fullständigt objektorienterat språk som har blivit enormt populärt till stor del tack vare dess ”applets”, en minitillämpning som kan köras på en webbsida. Att förklara språkets egenskaper ligger utanför ramen för det här rapporten. Jag hänvisar till en av de många böcker som finns om Java [17] och till Suns webbsida [18] där man kan finna inledande kurser på Java.

3.2 Vad är EAI ? (The External Authoring Interface)

EAI är ett viktigt tillägg till det VRML språket. Det beskriver ett sätt att styra en VRML-scen från ett externt program.[19] Även om programmet kan vara implementerat i vilket språk som helst, har EAI använts främst med Java. Som det externa programmet används en Java applet.

För att kommunicera med scenen finns ett speciellt paket av Java klasser. Paketet följer vanligtvis med det VRML-tilläggsprogrammet till webbläsaren. Paketet innehåller alla Java klasser och metoder för att bl. a.:

- Skapa och ta bort noder från en scen
- Läsa och ändra publika fält from en specifik nod.
- Sända inhändelser till en specifik nod.
- Ta emot uthändelser från scenen.

Dessutom kan man använda alla Java klassernas bibliotek, t ex kan användare via själva applet trycka på en knapp, implementerad med hjälp av AWT-paketet [20], och samtidigt skicka en händelse till en nod.

Vi beskriver EAI-mekanismen genom ett exempel där vi följer samtliga steg för att upprätta kommunikation mellan en Java applet och en VRML scen. Vi ska använda samma scen som på sidan 16 och skapa en koppling mellan den `PlaneSensor`-noden och position av lådans, inte genom en VRML-koppling utan via Java appleten :

- Först importerar man i appletens klass det `Vrml.external.*` paket som behövs för att använda EAI.

- Sent skapar man en referens till själva scenen. För detta används metoden `Browser.getBrowser().`

- Därefter refererar man till noder som man är intresserad av med metoden `getNode(nodens_namn).` Nodens namn definieras med DEF. I det här fallet referera vi till de två noderna som har namnen: `MinPlaneSensor` och `LådansTransform`.

- Med nodens referens har man tillgång till nodens fältvärde. Då får vi ändra direkt på dem om de är publika. För detta finns en direkt korrespondens mellan VRML och Javas datatyper. T. ex. inom metoderna konverteras från `SF3Vec` som används för att ange tredimensionella koordinater i scenen till Javas vektorer: `float [3]` och tvärtom.

- Slutligen behövs det en speciell metod för att kunna ta emot uthändelser från scenen. Det är den så kallad `callback()` metoden. För att veta från vilken nod genereras uthändelser initieras nodens fält med `advise( )`metoden. Dessa två metoder finns deklarerade i gränssnitet `EventOutObserver`, och appleten måste implementera dem.

- Scenen och appleten bäddas in tillsammans i en HTML -sida.

Koden från appleten och den HTML sidan följer:

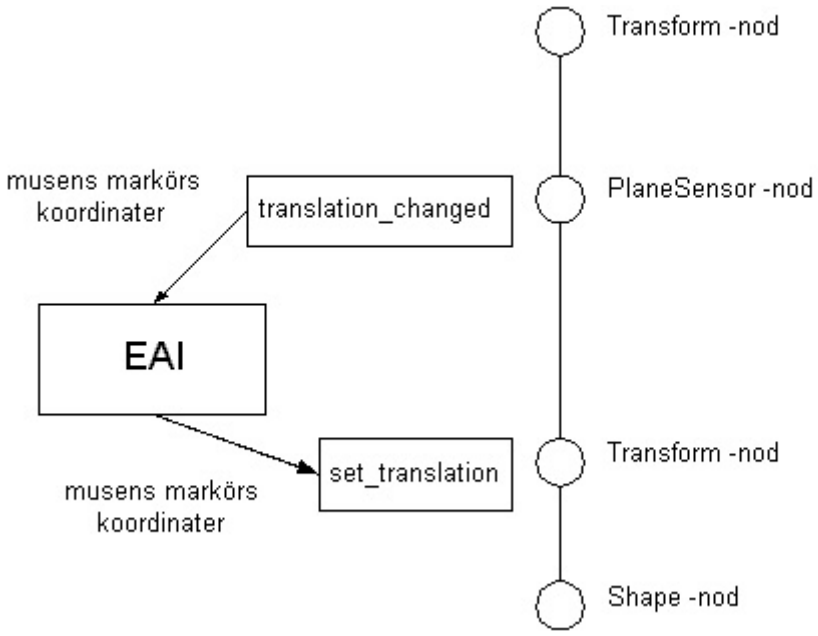
```
 // eaiExempel.java

import java.applet.*;
import vrml.external.*;
import vrml.external.field.*;
import vrml.external.exception.*;

class eaiExempel extends Applet implements EventOutObserver {

  Browser browser=null;
  Node eaiSensor=null;
  Node eaiTransform=null;
  EventOutSFVec3f mouseChanged=null;
  EventInSFVec3f  setPosition=null;

  public void start(){

  browser=Browser.getBrowser(this);
  eaiSensor=browser.getNode("theSensor");
  eaiTransform=browser.getNode(”theTransform”);
  setPosition=(EventInSFVec3f)eaiTransform.getEventIn("set_translation ");
  mouseChanged=(EventOutSFVec3f )eaiSensor.getEventOut("translation_changed");
  mouseChanged.advise(this,new Integer(1999));

  }

 public void callback(EventOut who, double when, Object which){

  Integer whichNum=(Integer)which;

  if (whichNum.intValue()==1999){
   float [] newPosition=new float[3];
   newPosition=mouseChanged.getValue();
   setPosition.setValue(newPosition);
    }
  }


}
```

Figur 3.1 Koden av den Java appleten

```
<html>
<body>
<embed  src = "scen.wrl" ></embed>
<applet code= "eaiExempel.class" mayscript></applet>
</body>
</html>
```

Figur 3.2 Den HTML -filen

Det som vi får är samma scen som i figur 2.9 men nu utan en direkt koppling (ROUTE) mellan noder. (Jämför fig. 2.10 och fig. 3.3). Detta öppnar möjligheten att ändra värde som kommer från en scen innan vi skickar det tillbaka till någon annan nod. Exemplet är förmodligen ett av de enklaste. Om man är intresserad kan man läsa mer om EAI i [21].



Figur 3.3 Scengrafen som visar hur man kan använda EAI för att skapa en koppling mellan två noder

3.3 EAI :s problem

Vi har valt att använda EAI i det här projektet eftersom det är den enda mekanismen som finns idag för att styra en VRML scen genom ett gränssnitt (AWT) och samtidigt kunna referera till samtliga noder. EAI är en kraftfull mekanism för att skapa dynamiska scener. Tyvärr finns det problem med dess stabilitet. [22] Webbläsaren kraschar ofta när det finns många händelser och scener är för stora. Dessa problem kommer förmodligen aldrig att åtgärdas. I stället kommer en ny standard kallad X3D att ersätta VRML och EAI inom ett par år

3.4 X3D: framtiden

Nästa standard för Internetbaserad VR kallas X3D (eXtensible 3D). Den blir förmodligen implementerad i Java. Den kommer att stödjas av specialiserad hårdvara så att renderingen blir avsevärt förbättrad. Man förväntar också att X3D blir stabilare än VRML och EAI. Lösningen ligger i ett mindre språk med en kärna och externa moduler som kan kopplas till den. Antagligen kommer det inte att behövas att man laddar ner tilläggsprogrammet för att titta på scener. Mycket mer information om X3D kan hittas på X3D:s officiella webbsida [23].

4 3D-gränssnitt med stöd för fysikalisk simulering

I det här kapitlet beskrivs hur ett 3D-gränssnitt med stöd för fysikalisk simulering har utformats och byggts. Först har vi implementerat en motorn för att simulera några fysiska fenomen som gravitation, friktion och kollisioner mellan objekt. Därefter har lagts till de funktioner som låter betraktaren förflytta objekt i en VRML-scen.

Motiveringen till att vi vill simulera fysiska fenomen ligger i vår hypotes att det skulle bli lättare för användare att interagera med objekten om dem betedde sig som i verkligheten.

För att implementera simuleringsmotorn har vi valt mellan olika typer av fysiska modeller vilket diskuteras i avsnittet 4.1.Vår modell är baserad på registrering av kollisioner mellan objekten. För att kunna upptäcka dessa kollisioner används en kollisionsdetekteringsalgoritm. Detaljer om algoritmen ges i avsnittet 4.2. Därefter, i avsnittet 4.3 behandlas interaktionsdelen och hur den har integrerats till simuleringsmotorn.

4.1 Fysikalisk Simulering

4.1.1 Fysiska modeller

Grunden för att implementera en simuleringsmotor är formuleringen av en modell som hjälper oss att på ett matematiskt sätt beskriva hur ett system beter sig. Eftersom vi strävar efter att simulera verkligheten använder vi en fysisk modell som beskriver den genom fysikaliska lagar.

Det finns olika slag av fysiska modeller. Det finns dynamiska modeller som baseras på Newtons lagar och som använder dynamiska parametrar för att bestämma systemets tillstånd. Till exempel används modeller för krafter som påverkar objekt för att bestämma hur objektet rör sig. En matematisk beskrivning av dynamiska modeller kan hittas i [24]. Dynamiska modeller kan med stor noggrannhet beskriva alla delar av ett system.
Men dessa karakteriseras av att ha stor beräkningsvolym som resultat av integration och lösning av differentialekvationer. [24] Därför blir dynamiska system ofta oanvändbara i realtidsapplikationer där användare interagerar direkt med föremål och svarstiden måste vara kort. Andra typer av fysiska modeller blir mer eller mindre approximativa men också snabbare: det finns kinematiska modeller som baseras enbart på objekts hastighet [25] och positionsbaserade modeller [26] som kan användas för att bestämma rörelse hos en mängd av objekt som en funktion av deras energi.

Eftersom vi i det här fallet inte är intresserade av att simulera exakt objekts rörelser utan att animera dem på ett approximativt sätt använder vi en modell som kombinerar kinematiska , dynamiska och heuristiska algoritmer. Modellen baseras på objektens hastighet men tar också hänsyn till andra objekts fysiska egenskaper som friktionskoefficient och massa .Vi definierar en tillståndsmodell som hjälper oss att skilja på möjliga fall som kan inträffa när vi uppdaterar objektens position under simuleringen.

4.1.2 Tillståndsmodell

I en tillståndsmodell har varje objekt ett antal tillstånd. För varje tillstånd finns det en funktion som uppdaterar objektets position under simulering. Vi kallar dessa funktioner för uppdateringsfuntioner.

För att simulera realistiska rörelse strävar vi efter att följa några animerings principer beskrivna i [27].Bland dessa finns :

- När objekt flyger beskriver deras rörelse en parabel.
- När objekten kolliderar med varandra syns resultatet av kollisioner mellan dem.
- Friktionskraft påverkar objektens rörelse.

För att simulera dessa effekter definierar vi följande tillstånd för varje objekt: ”flying”, ”grounded” och ”on collision”. Ett objekt finns i ”grounded” tillstånd om det har kontakt med marken eller med ett objekt som ligger på marken. När ett objekt finns i ”grounded ” tillståndet ändras inte objektets position i Y -koordinaten. Objektets hastighet i XZ -planet minskar beroende på friktionskoefficients värde. Ett objekt finns i ”flying” tillstånd om det inte är i ”grounded” tillståndet. Då uppdateras objektets position med hänsyn till gravitationskraften och friktion verkar inte i objekts rörelse. Till sist när det blir en kollision mellan objekt växlar de med varandra hastighetsvärde och riktningar. Båda objekten finns då i ”on collision” tillstånd.

För att kunna bestämma objekts tillstånd under simulering används en kollisionsdetekterings-algoritm. Det blir genom algoritmen som vi kan bestämma om ett objekt är kontakt med marken (grounded) eller inte (flying).Dessutom används algoritmen för att ändra på uppdateringsfunktions parameter när två objekt kolliderar med varandra. Algoritmens detaljer förklaras i nästa avsnittet. Uppdateringsfunktioner kan hittas i Bilagan C.

4.1.3 Objekts representation

I simulering har vi bestämt att varje föremål har fysikaliska egenskaper: hastighet, massa, friktionskoefficienter etc. Dessa objekts fysiska egenskaper används som parameter inom uppdateringsfunktioner.

Representationen av fysiska objekt implementeras dels med egendefinierade typ (`PROTO`) i VRML dels med EAI Java klasser.

I VRML används en `PROTO`-deklaration för att lagra initialvärde av objektens fysikaliska egenskaper. Där också beskrivs objektens geometri. I figur 4.1 visas en preliminär `PROTO` -deklaration för fysikaliska objekt. Jag kallar den preliminär eftersom vi kommer att lägga till ytterligare information. I `PROTO`-deklarationen används fältvärde i form av vektorer för att representera linjär hastighet, krafter och objekts initialposition. Enkla fältvärden används för t ex. lagra information om objektens massa. För att optimera rendering av scener infogas objektens geometri via en `Inline`-nod.

En Java klass representerar också fysiska objekt (`phyNode`).Den innehåller objektens tillstånd, fysiska egenskaper och uppdateringsfunktionerna som metoder. (se Bilaga B) Med hjälp av EAI kan initialvärde på objekts egenskaper läsas från scenen. Detta sker med referenser till fältvärde som förklarades i kapitel 3.

Genom att bestämma objekts tillstånd och använda uppdateringsfunktioner ändras objektens position. Dessa nya värde överförs till scenen via EAI . En `TimeSensor`-nod, som ligger i scenen, är ansvarig för att generera varje tidsteg. Dessa tas emot med `callback` metoden i Java.

```
PROTO PhysicalObject [

exposedField SFRotation objOrientation 0 1 0 0
exposedField SFVec3f    objPosition    0 0 0
exposedField MFString   objURL ""
exposedField SFVec3f    objLinearVelocity 0 0 0
exposedField SFFloat    objMass 1
exposedField SFVec3f    objForce 0 0 0

]

Transform {
                translation  IS objPosition
                rotation     IS objOrientation

               children  Inline { url IS objURL }

          }
```

Figur 4.1 En preliminär `PROTO` deklaration för fysikaliska objekt

4.2 Kollisionsdetektering

I vår simulerings modell är det viktigt att kunna upptäcka kollisioner mellan objekten eftersom vi använder dem för att bestämma deras tillstånd.

Tyvärr finns inte stöd i VRML för att automatiskt upptäcka kollisioner mellan tredimensionella objekt. Därför måste vi själva implementera en mekanism med hjälp av Java och EAI . För detta används en kollisionsdetekterings algoritm.

Det finns vissa egenskaper som en kollisionsdetekteringsalgoritm måste ha för att bli funktionell i vår implementation:

- Den bör ha mindre beräkningstid än $O(n^2)$. Man får denna beräkningstid om varje objekts position jämförs med varandra. Detta är generellt ineffektivt, speciellt i realtidsapplikationer.
- Den måste kunna upptäcka kollisioner efter att användaren har ändrat på objekts position , dvs algoritmen ska inte enbart använda objektens positions initialvärde. Detta är ett krav för att kunna manipulera föremål under simuleringen.

Lin [28] sammanfattar många av kollisionsdetektering algoritm som finns idag såsom presenterar kollisionsdetekteringspaket som I_COLLIDE [29] och SOLID [30].

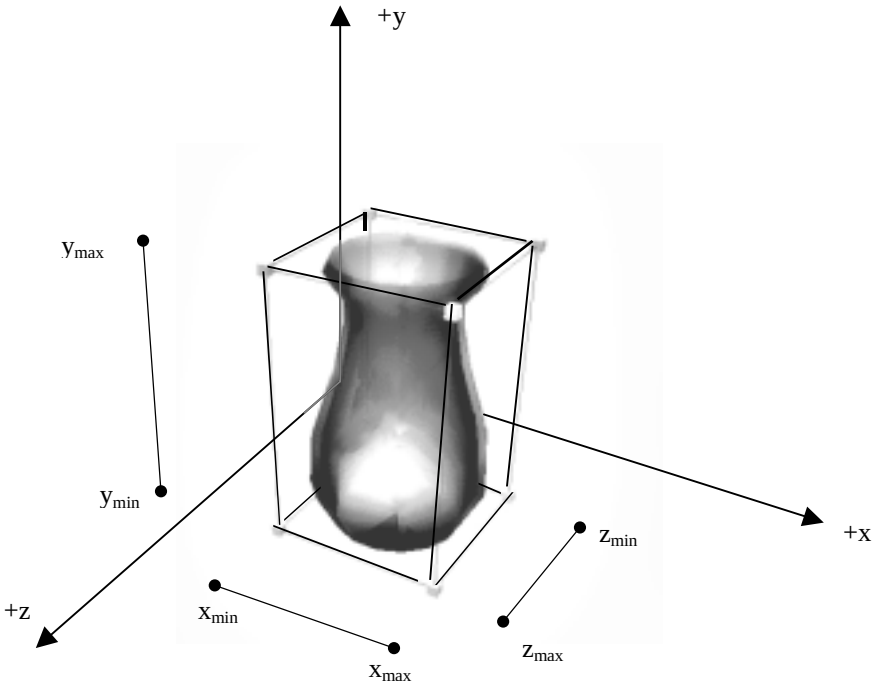
4.2.1 AABB algoritmen

Mest av dessa paket använder först en preliminär approximativ kollsionsdetekteringsalgoritm och sedan använder exaktare och långsammare algoritmer för att upptäcka kollisioner mellan par av objekt. Vi ska använda en algoritm kallad AABB (fr. eng Axel Aligned Boundings Box).[24] AABB algoritmen tillhör klassen av preliminära approximativa algoritm. Den är väldigt snabb och beror inte av initialposition av objekten vilket uppfyller våra krav. Vi använder den med tanke på att skapa en grund till en bättre kollisionsdetekteringsmekanism. Till exempel kan AABB algoritmen förbättras med användning av träd som delar objektens geometri i ett antal av lådor hierarkiskt organiserade[31].

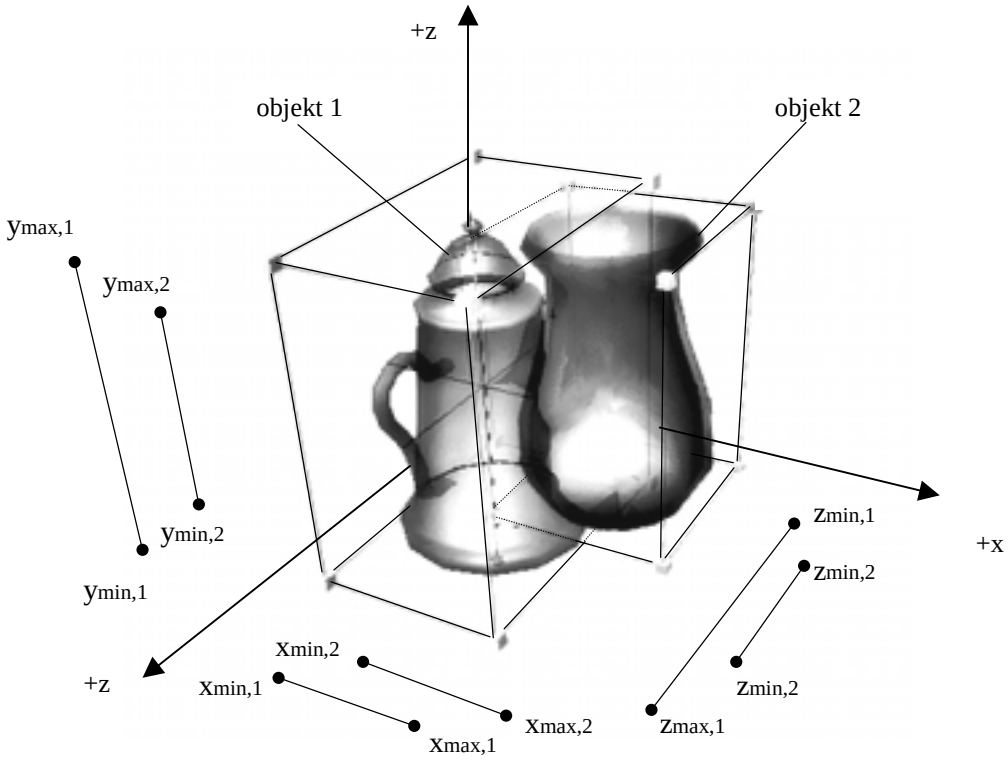
Algoritmens grundtanke är att ersätta objektens geometri med en låda omkring dem (eng. bounding box). Lådan är orienterad på ett sätt att alla sina kanter är parallella med någon av axlarna.

En låda beskrivs med tre intervall $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, $[z_{min}, z_{max}]$ som visas i figur 4.2. Intervallen lagras i tre listor, en för varje axel och sorteras. Därefter kontrolleras listor för att hitta intervall som överlappar varandra. Två lådor kolliderar med varandra om dess intervall överlappar i de tre listorna (se figur 4.3). Då kan vi säga approximativt att objekt har kolliderat. Det är viktigt att lägga märke till att det finns objekt vars geometri beskrivs dåligt av lådor, t ex. objekt med hål eller långa objekt som inte har sina kanter parallella med axlarna.

Orsaken till att vi använder algoritmen i motorn är främst dess korta beräkningstid. Algoritmen utnyttjar scenens sammanhang (eng. coherence), det vill säga att objekt inte ändrar sina positioner så snabbt att listorna förblir nästan sorterade efter varje uppdatering. Det har visats att om man använder `insertion sort` metoden [32] för att sortera listor förväntas en beräkningstid lika med O(n+m) där n är antal av objekt som rör på sig och m är antal av objekt som står stilla.[24]



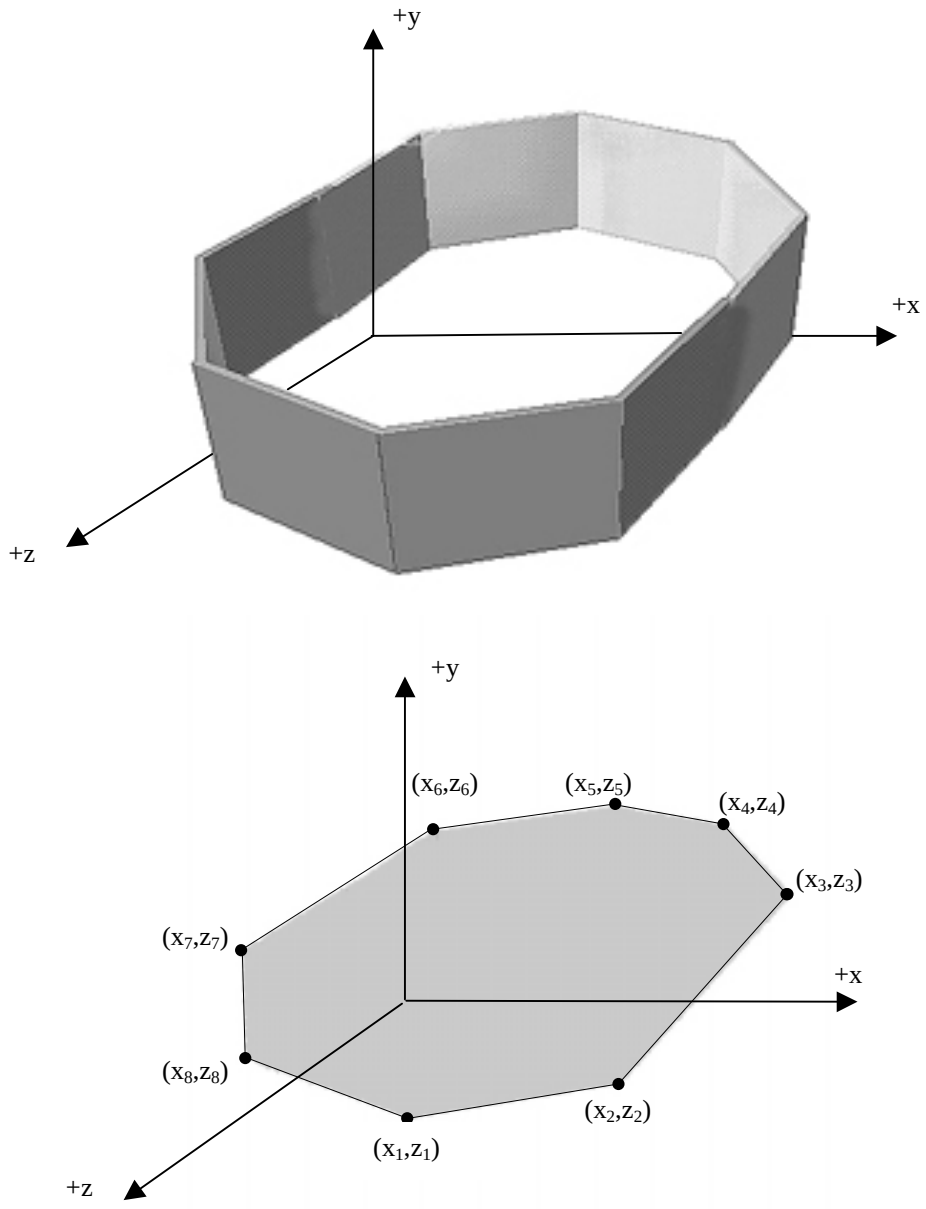
Figur 4.2 Objekt med sin axel-orienterade låda. Bilden visar också lådans representation med intervall.



Figur 4.3 Två objekt kolliderar om de två lådornas intervall överlappar med varandra.

4.2.2 2D Kollisionsdetektering

Som vi innan anmärkte är AABB algoritmen särskilt ineffektiv för att beskriva långa objekt som inte har sina kanter parallella med axlarna. I vissa fall kan det bli en lösning att använda AABB algoritmen tillsammans med en 2D -kollisonsdetektering.(fig. 4.4) Vi introducerar metoden för att ge mer frihet när man vill definiera ett rum där alla objekten måste finnas t. ex. en byggnad. Implementering av algoritmen sker med metoder av Java AW- paketet [20] .



Figur 4.4 2D kollisionsdetektering kan användas för att bestämma ett rum

4.2.3 Kollisionsdetektering och objekts tillstånd

För att kunna bestämma objektens tillstånd kontrollerar vi objektens position vid varje tidsteg. Pseudokoden för att ändra objekts tillstånds visas i figur 4.5.

```
Ändra objekts tillstånd (indata: tidsteg)
{
För varje objekt {
 Uppdatera objekts låda(tidsteg)
}

Sortera x,y ,z intervall listor

För varje objekt {

 Om objektets låda kolliderar med marken => objekts tillstånd =
grounded;
    annars objekts tillstånd =flying;
 }

För varje objekt a som kolliderar med objekt b{

    Om objekt a tillstånd = grounded => b tillstånd = grounded;
    Om objekt b tillstånd = grounded => a tillstånd = grounded;

    Objekt a tillstånd =on collision;
    Objekt b tillstånd =on collision;

    Växla mellan objekt a och objekt b hastighet

     }
 }
for varje objekt{
 Om objektets låda inuti det 2D -definierade rummet uppdatera
objekt(tidsteg); }
}
```

Figur 4.5 Pseudokod för att växla mellan objektstillstånd beroende på kollisioner mellan dem

4.2.4 Implementering

För att kunna använda AABB -algoritmen beräkna en låda omkring varje objekt. Detta gjorts automatiskt under den konvertering som vi beskriver i kapitel 5. Därför antar vi här att lådans position och storlek är kända

På den VRML -sidan läggs till två fält som initialt anger lådans center och storlek i den `PROTO` -deklaration av fysikaliska objektet (se Bilaga A). Därefter läser den `phyNode` -klassen information om lådan och skapar den.

I Java instansieras varje låda med ett `Box` -objekt. Som komponenter av den `Box` -klassen ingår `Segment` -klassen för att representera varje x, y z intervall. Extremvärdena för varje intervall lagras i `Vertex` -objekt. Alla dessa Java klasser representeras i Bilaga B.

Lådor uppdateras med varje tidsteg. När detta händer uppdateras också intervall. Intervallen finns lagrade på en lista för varje axel som sorteras och kontrolleras för att hitta objekten som kolliderar. Ett objekt från en klass kallad `simColl` är ansvarigt för att anropa metoder som uppdaterar dessa lådor, sorterar listor, och ändrar objektens tillstånd.

Slutligen vad gäller 2D-rummet, där objekten måste finna, sås implementeras det med `Zone` -klassen. Vi använder enbart information av lådans sida som är parallell med marken för att kontrollera objektens position i rummet.

4.3 Interaktionsdel

För att implementera interaktionsdelen har vi begränsat interaktion till att betraktaren ska kunna förflytta föremål. I det här avsnittet beskrivs hur interaktionsdelen har utformats och implementerats. Vi behandlar också hur man integrerar simulering och interaktion genom att vidareutveckla tillståndsmodellen.

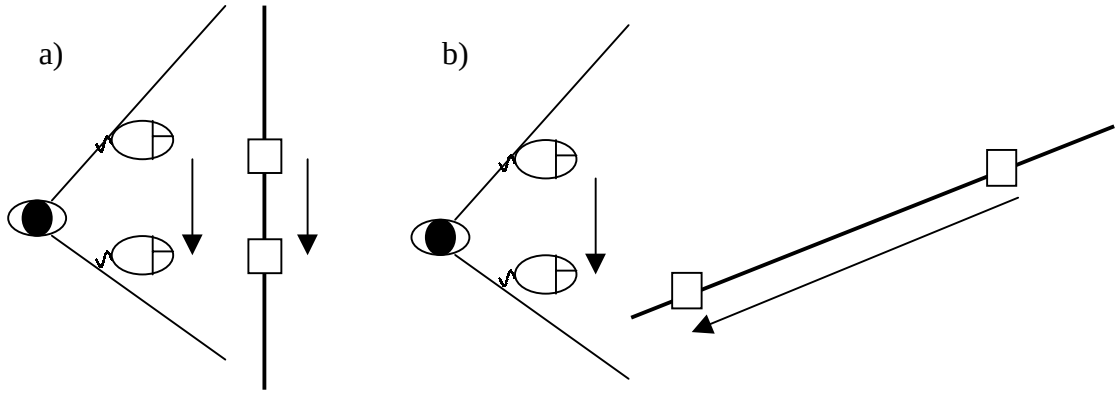
4.3.1 Direkt interaktion

Direkt interaktion är att kunna direkt påverka objekt i en 3D -miljö utan användning av 2D element som menyer, knappar, checkboxer etc. Det är uppenbart att direkt interaktion är mer intuitiv än andra form av interaktion och på något sätt påminner oss vårt sätt att manipulera reella objekt i verkligheten. I inledningen angavs några olösta problem kring utformning och implementering av ett 3D -gränssnitt. Många av dessa är relaterade till direkt interaktion. Därför är det är viktigt att vara medveten om att det inte finns några riktlinjer eller metoder som specificerar hur direkt manipulation måste utformas eller implementeras för att den ska vara effektiv[4, 7].

Första problem som vi behandlade var att definiera på vilket sätt inmatningsenheter skulle användas. Detta är ett problem som behandlas i [33] när de gäller om rotering av objekt .Vi kommer att föreslå en egen modell för förflytta föremål. I VRML används endast mus för att manipulera objekt. Det finns inte möjligheter att registrera tangenttryck. Man kan endast använda piltangenter för att förflytta sig i scenen.

Som är känd är musen en 2D -inmatningsenhet. Detta betyder att vanligtvis översätts musens markörposition till ett koordinatpar. Dessa koordinater tillhör till ett plan som vanligtvis kan definieras, t ex med hjälp av knappar (”hot keys”) som i gränssnitt i kända modelleringsprogram som t. ex. TrueSpace 4 [14], eller Studio Max 3D [13] .

Vårt sätt att hantera musens inmatning är annorlunda .Vi inkluderar informations om användares position och orientering. Vi tror att manipulation underlättas om musens koordinater tillhör till ett plan som ligger ortogonalt till vypunkten.(figur 4.6 a). Annars skulle små rörelser av musen översättas i långa rörelser av objekt såsom visas i figur 4.6 b). Detta skulle komplicera manipulation av objekt i scenen. Dessutom liknar denna form av interaktion det sätt vi rör reella objekt med handen.



Figur 4.6 Betraktare drar ett objekt med musen. a) Objektet rör sig på ett plan som är ortogonalt till vypunkt. b) Om planet är inte ortogonalt till vypunkten kommer musens små rörelse att tolkas som långa rörelse av objektet

Därefter bestämde vi om hur objekt skulle förflyttas. Det verkade jobbigt att med enbart musen förflytta föremål långa avstånd. Därför implementerades två arbetssätt som en metafor för att manipulera objekten med handen.

- För att placera ett objekt på korta avstånd klickar man med musen på objekt, och med musens knapp nertryckt drar man långsamt musen och förflyttar objekt.

- För att placera ett objekt på långa avstånd klickar man på ett objekt och då ”plockar” man det. Sedan kan betraktare använda piltangenter för att förflytta sig tillsammans med objektet i scenen och ”släppa” det när han/hon vill.

- För att göra det första arbetssättet ännu mer realistiskt införde vi alternativet att man kan ”kasta” objektet om man drar musens snabbt och sent släpper musens knapp.

Tyvärr måste vi ändå använda knappar för att växla mellan dessa arbetsätt. Det finns två knappar: ”get” och ”drop”. Om man trycker på ”get” kommer man att plocka nästa objekt som man klickar på. Om man trycker på ”drop” är man tillbaka i det första arbetssättet och

man kan släppa ett objekt som var plockat om man klickar på det. Andra varianter om hur man växlar mellan dessa arbetsätt har också testats.

På det sättet kan man i varje moment förflytta objekten i ett plan som ligger ortogonalt till vypunkten eller förflytta sig tillsammans med dem.

Dessutom kan man manipulera mer än ett objekt samtidigt. Om vi lägger ett objekt ovanpå andra som vi har plockat kan vi förflytta båda. Vi kan också förflytta andra objekt via kollisioner mellan dem.

4.3.2 Implementering

För att förflytta objekt i korta avstånd använder vi en `PlaneSensor` -nod. Problemet är att så snart som användare förflyttar sig i scenen och vypunkten ändras hamnar vi i en situation som i figur 4.6 b). Detta händer eftersom musens position översätts till ett par av koordinater som tillhör till det XY -planet i det lokala koordinatsystemet där sensorn och objektet finns. Det lokala koordinatsystemet roterar inte automatiskt så att planet XY-blir ortogonalt till vypunkten.

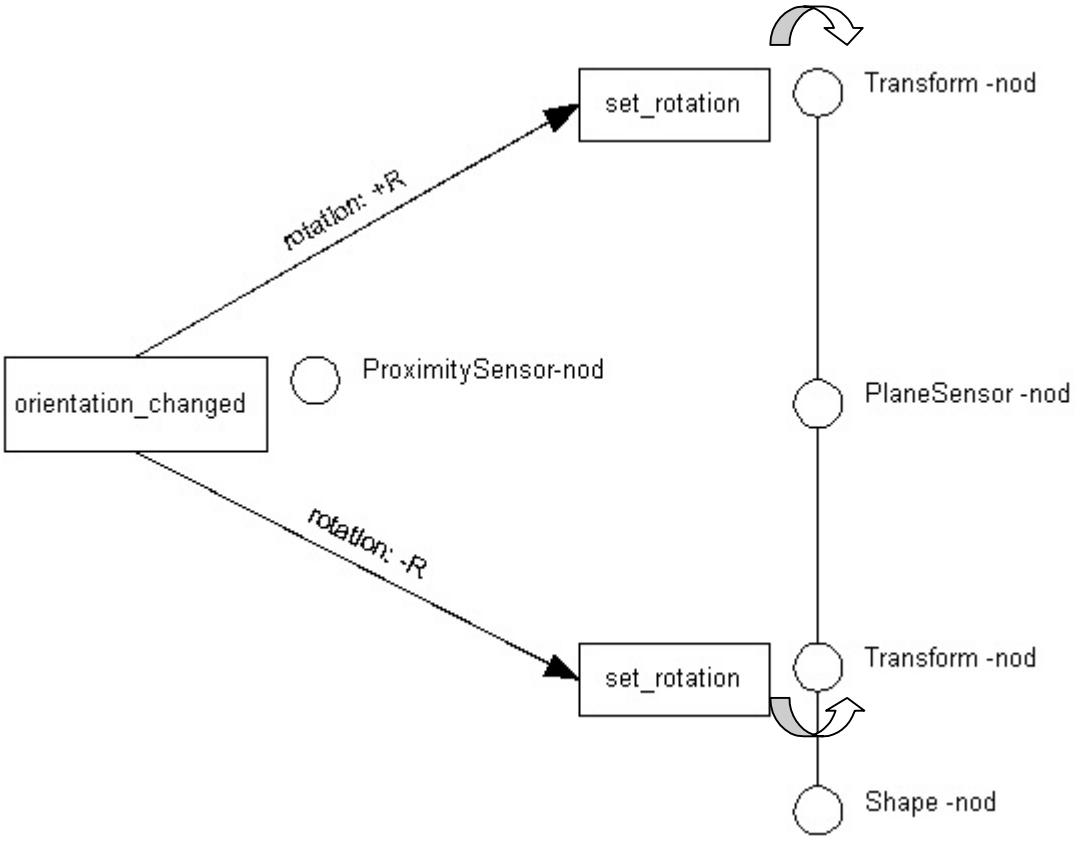
Vår lösning ligger i en sammansättning av två rotationer: en för att orientera `PlaneSensor` -nodens koordinatsystem så att det lokala XY-planet ligger ortogonalt mot vypunkten och en annan för att rotera objekt tillbaka eftersom första rotationen kommer också att påverka objektet.(Se sid 15 där det förklaras hur både objektet och sensorn måste ligga i ett gemensamt koordinatsystemet)

För att uppföra dessa två rotationer med hjälp av EAI måste vi veta när vypunkten ändras. För detta används en `ProximitySensor`-nod. Den genererar uthändelser varje gång som betraktare förflytta sig i scenen. Händelsevärden är den nya positionen och den nya orienteringen. Dessa läses från scenen med hjälp av den EAI:s `callback()` metoden.

Båda rotationerna representeras i en scengraf i figur 4.7. Det första rotation representeras med en pil omkring den gemensamma `Transform`-noden. Den andra pilen representerar rotation för att rotera objekt tillbaka. Både rotationerna blir osynliga för betraktaren.

I lösningen använder vi två inbädda koordinatsystem och därför får vi lokala koordinater från sensorn. Dessa koordinater måste konverteras till scenens globala koordinatsystem. Anledningen är att vi kontrollerar alla objektens globala position i kollisionsdetekteringsmekanismen.

Slutligen för att inte behöva rotera alla `Transform`-noder och alla objekt som finns i scenen används en `TouchSensor`-nod. Den registrerar om musens markör finns i en geometri. Då utför vi båda rotationer enbart i de objekt som kommer att manipuleras. För att inte översvämma scenen med händelser fungerar denna sensor enbart efter att en operation har avslutats.



Figur 4.7 Två rotationer för att orientera sensors planet

För att kunna ”plocka” objekten och förflytta dem på scenen använder vi samma ProximitySensor -noden. Vi summerar helt enkelt en ändring av betraktarens position till objektets position.

Alla dessa sensornoder och inbäddade koordinatsystem ingår i den interna strukturen för `PROTO` deklarationen för fysikaliska objekt. (Bilaga A)

I Java delen av implementationen finns en klass kallad `actHandler` som ändrar objektens fältvärde och tar emot uthändelser från sensorn noder. Därefter kontrolleras uppdatering av objekt för att upptäcka kollisioner. För detta använder vi lådornas axel-listor som i kollisionsdetekteringsalgoritmen vid simulering. Detta skapar ett synkroniseringsproblem som löses med trådar och Javas synkroniserade metoder.

4.4.3 Kollisionsdetektering och interaktion

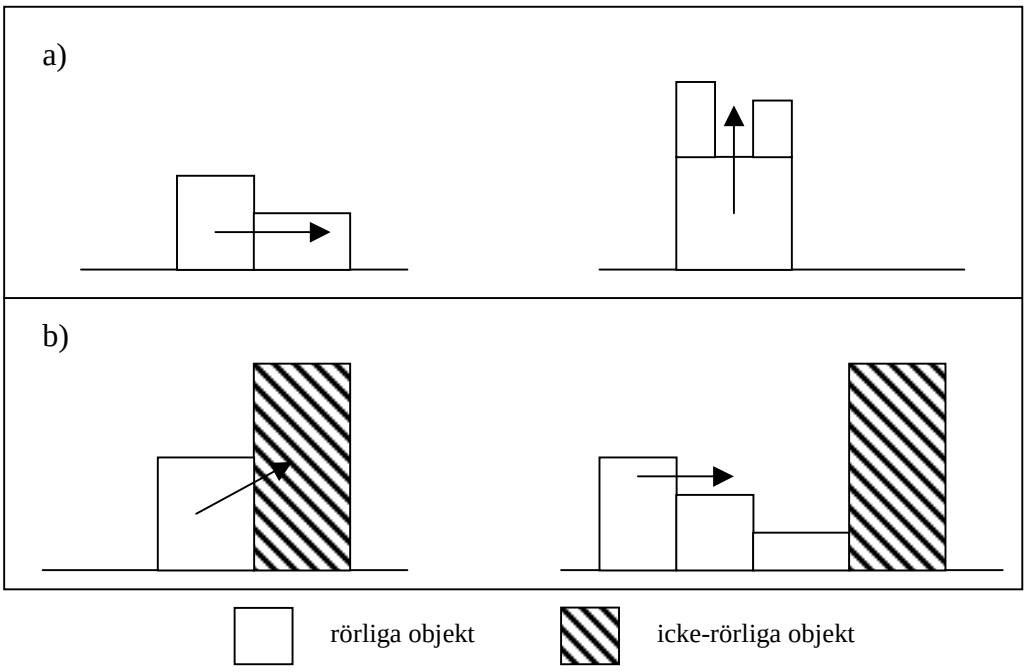
Placeringen av objekt via interaktion kontrolleras med kollisionsdetekteringsalgoritm som bestämmer om rörelse tillåts. Syftet är att förhindra att objekts går igenom varandra när man manipulerar dem och också tillåta att transmittera interaktion mellan objekt. Till exempel när två objekt finns intill varandra och vi förflyttar ett objekt om det kolliderar med ett annat vill vi att båda objekten rörs om rörelsen är möjligt.(fig 4.8 a).

I det här sammanhanget introducerar vi en uppdelning mellan objekt: rörliga objekt som kan manipuleras och icke-rörliga objekt som inte kan manipuleras. Till dem sista tillhör till exempel väggar, golvet, träder etc. Om ett rörligt objekt kolliderar med en icke-rörlig objekt kommer dem att inte kunna förflyttas. Inte heller om objekten utgår från 2D -rummet som vi har definierat för 2D- kollisionsdetekterings mekanismen.

För att hitta ett sätt att bestämma när objekten kan förflyttas eller inte definierar vi en giltig translation som en translation som uppfyller :

- Det objekt som förflyttas är rörligt
- Objektet förblir inom 2D –rummet efter att ha utfört translationen.
- Som resultatet av translationen sjunker inte objektet ner i marken.
- Om objektet kolliderar med andra objekt är translationen också giltig för dem.

Vi tittar på några exempel i figur 4.8. Pilarnas origo finns i objekt som manipuleras och visar translationens riktning . I figur 4.8 a) har vi giltiga translationer eftersom objekts som kollideras också kan förflyttas. Däremot i figur 4.8 b) har vi två ogiltiga translationer, eftersom translationer inte blir giltiga för de icke-rörliga objekten.:



Figur 4.8 Giltiga och icke-giltiga translationer

På samma sätt som vid simulering uppdaterar vi först objektets låda. Därefter kontrollerar om translationen är giltig för alla objekt som kolliderar. Det sista objekt som rör sig som resultatet av translation blir det första som uppdateras. Hela resonemanget leder oss till en pseudokod som beskriver vårt sätt att bestämma om ett objekt kan förflyttas eller inte.(figur 4.9)

```
boolean Uppdatera vid manipulation (translation, objekt)

Vektor giltigVector;
Int i=0;

Om objektet icke-rörlig return False;

Uppdatera objektets låda (translation);

Om objektets låda utanför den 2D zone => return False
Om objektets sjönk i marken => return False


Sortera tre listor med intervall

För varje objekt ”A” som kolliderar med objektet{

giltigVector [i++]= Uppdatera vid manipulation (rörelse,A)
}

Om alla element i giltigVector == True =>

      uppdatera objekt (translation) och return True;

      Annars => läggs lådan tillbaka och return False
```

Figur 4.9 Pseudokod för att bestämma om ett objekt kan förflyttas via interaktion

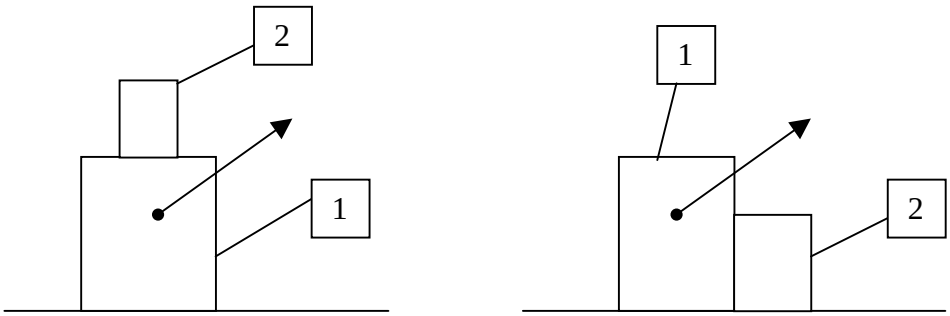
Denna pseudokod implementeras i `actColl` Java klassen. Lägg märke till att intervallens listor är samma som används vid simulering. Därför synkroniseras kollisionsdetekterings mekanismen och skydds metoder där listorna sorteras.

4.3.4 Simulering och interaktion

Vi vill inte simulera gravitation i objekt som manipuleras, (tänk på att du har något i handen som inte kan hållas eftersom gravitationen förhindrar det).Därför inför vi en ny tillstånd kallad ”locked”. Ett objekt är ”locked” om musens markör finns över objektets geometri och musens knapp är nedtryck. Då uppdateras inte objektet via simulering och inte heller objektets lådan. Objektet är inte längre ”locked” så snart som vi släpper musens knapp och flyttar musens markör.

Eftersom ett objekt som är plockat för att förflyttas långa avstånd inte heller borde påverkas av simulering inför vi tillstånd ”selected”. Tillståndet används också för att veta vilka objekt måste förflyttas om betraktare också gör det.

Slutligen måste vi också klara att överföra interaktion. Ett objekt som förflyttas via överförings av interaktion måste kontrolleras för att det ska simuleras eller inte. Det finns nämligen två fall (fig. 4.10) :



Figur 4.10 Två fallet för att överföra en translation

I fallet a) är vi intresserad att objekt 2 inte sjunker in i objekt 1 som manipuleras. Därför får gravitationens effekt inte simuleras för båda . I fallet b) är vi intresserad att simulera gravitation med objektet 2. Därför införs ett ny tillstånd ”loaded” som karakteriserar objekt 2 i fallet a) men inte objekt 2 i fallet b). Kriterium är att objekt som manipuleras måste finnas helt under objekt till vilket interaktions överförs för att blir ”loaded”. Samtliga tillstånd finns representerade i en diagram i Bilaga B.

I nästa kapitel ska beskrivas ett verktyg för att konvertera mellan enkla VRML-scener och scener som innehåller objekt med fysikaliska egenskaper så att de kan visas och manipuleras genom vårt gränssnitt.

5 Konvertering

5.1 Allmänt

För att underlätta användning av motorn har det byggts en ”översättare” som kan förändra strukturen i en vanlig VRML-scen till en annan där föremålen beskrivs genom `PROTO` – deklarationen av fysiska objekt.

Översättare har skrivits med hjälp av CyberVRML97 Java paketet skrivet av Satoshi Konno och som kan hittas i [33]. CyberVRML97 är en VRML-parser som kan användas för att få specifik information om en scen som till exempel: objekts geometri, placering av objekten i scenen, typer av nod etc. Det är också med hjälp av CyberVRML97 som de axel-orienterade lådorna för den AABB algoritmen beräknas.

Översättare skapar också alla filer som beskriver objektens geometri i enskilda filer för `Inline` -noden. Det återstår enbart att specificera objekts initialegenskaper eller använda de predefinierade värden som finns på den `PROTO` –deklarationen.

Programmet gör följande steg vid konvertering:

1. Skapar en fil där den konverterade scenen kommer att beskrivas och vi kallar den phyScen.wrl
2. Skapar en referens till den VRML-filen där scenen beskrivs.
3. Scenen traverseras för att hitta Transform –noder.
4. För varje `Transform`-nod :

- Skapas en fil där läggs information om objektens geometri i `Transform`-noden. Filen har samma namn som nodens namn (specificeras genom `DEF`).
- I phyScenen.wrl -filen skapas en instans av PROTO-deklarationen där skrivs information om noden: initial placering av objektet och storlek, och centrum för den axel-orienterade lådan omkring objektets geometri. Där skapas också referensen till filen som har objektets geometri via den `Inline`-mekanismen inuti PROTO -deklarationen.

5. Scenen traverseras igen och alla andra noder som inte är av `Transform`-typen kopieras direkt från scenens filen till phyScenen.wrl

6 Slutligen läggs information om klockan och `ProximitySensor`-noden som behövs för simuleringen och interaktionen respektivt.

5.2 Exempel

Vi visar ett enkelt exempel. Antag att vi har en scen som består av två objekt en kub och en boll och som finns beskriven i en fil kallad scen.wrl .VRML -filen visas i figur 5.1 .

```
#VRML V2.0 utf8

DEF boll Transform {
 translation          -3 2 -2
  children Shape {geometry Sphere {radius 2 }}
}

DEF kub Transform {
 translation          3 1 -2
  children Shape { geometry Box {size 2 2 2 }}
}
```

Figur 5.1 Scen.wrl som består av en boll och en kub

Efter konvertering genereras tre filer: kub.wrl, ball.wrl och phyScen.wrl. Dessa filer visas respektive i figurer 5.2 , 5.3 och 5.4. Medan Kub.wrl och Boll.wrl beskriver geometrin för objekt i scenen, phyScen.wrl har information för `PROTO`-deklarationen för fysikaliska objekt och referar till de andra filer.

```
#VRML V2.0 utf8

Transform {
children [ Shape {geometry Sphere {radius 2.0}]
}
```

Figur 5.2 Ball.wrl beskriver bollens geometri

```
#VRML V2.0 utf8

Transform {
children [ Shape {geometry geometry Box { size 2.0 2.0 2.0 }]
}
```

Figur 5.3 Kub.wrl beskriver kubens geometri

```
#VRML V2.0 utf8

EXTERNPROTO PhyNode [
eventOut       SFVec3f    theSensorPos
eventOut       SFBool     theSensorActive
exposedField   SFVec3f    theTranslation
exposedField   SFVec3f    theScale
exposedField   SFRotation theRotation
exposedField   SFVec3f    theDimensions
exposedField   SFVec3f    theCenter
exposedField   SFVec3f    theVelocityLinear
exposedField   SFFloat    theMass
exposedField   SFVec3f    theForce
exposedField   SFVec3f    theSensorOffset
exposedField   SFBool     moveable
exposedField   MFString   theURL
exposedField   SFRotation thePRotation
exposedField   SFVec3f    thePTranslation
exposedField   SFFloat    Selected
exposedField   SFBool     theTouch
eventOut       SFBool     theOver ]
"Phy.wrl#PhyNode"

DEF ROOT Group{children [


DEF bollPhy   PhyNode{
PTranslation -3.0 2.0 -2.0
boxDimensions 4.0 4.0 4.0
boxCenter -3.0 2.0 -2.0
URL "Boll.wrl"
}


DEF boxPhy PhyNode{
PTranslation 3.0 1.0 -2.0
boxDimensions 2.0 2.0 2.0
```

Referens till den PROTO deklaration för fysiska objekt

Bollen

Kuben

```
boxCenter 3.0 1.0 -2.0
URL "Kub.wrl"
}
]}


DEF Timer1_0 TimeSensor {
enabled   TRUE
startTime 50
stopTime   0
cycleInterval 2000
loop  TRUE}

DEF Start1_1 Script {
eventIn      SFBool     start
eventOut     SFTime     startTime
url "vrmlscript: function start(value,ts){startTime=ts
}


DEF PS ProximitySensor {
enabled  TRUE
center 0 0 0
size        90000 90000 90000

ROUTE Start1_1.startTime TO Timer1_0.set_startTime
```

Simuleringsklockan och ProximitySensor -noden

Figur 5.4 Filen phyScen.wrl som kan läsas av motorn

P sättare för att automatiskt konvertera den till en scen som består av objekt med fysikaliska egenskaper som kan simuleras och manipuleras i vår motor.

5.3 Begränsningar

Det finns ett antal begränsningar om vilka element som tillåts finnas i scenen. De viktigaste är:

- Man kan endast använda `Transform`-noder för att gruppera objekten.
- Det bör inte finnas någon VRML mekanism som påverkar objektens rörelser, detta gäller speciellt till kopplingar och sensorer.
- Det är viktigt att notera att varje objekt måste finnas i en enskild `Transform`-nod för konvertering. Om två objekt finns i en gemensam `Transform`-nod kommer dem att antas vara ett enda objekt. Detta är långt från en nackdel eftersom det låter oss sammanställa olika objekt med varandra, men lägg märke till att en enda låda för kollisionsdetekteringsmekanismen kommer att finnas omkring dem.

I nästa kapitel ska vi demonstrera det tredimensionella gränssnittet med stöd för fysisk simulering genom ett exempel.

6 Exempel

I det här kapitlet beskrivs ett exempel där vår interaktionsmodell använts för att manipulera föremål i en VRML scen. Scenen föreställer en utställningshall där användare kan förflytta möbler från och till olika rum. Det finns också ett antal objekt som kan läggas ovanför möbler.

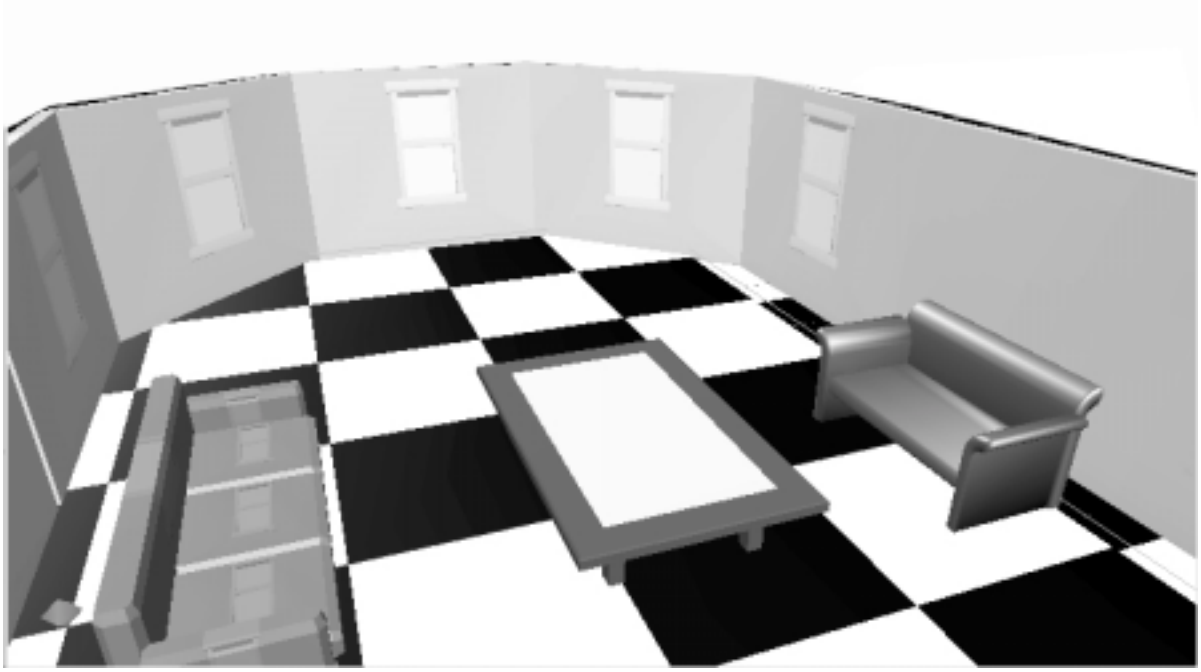
Scenen är byggd med hjälp av Cosmo Worlds[12] .Sedan har konverteraren använts för att ändra scenens struktur så att objekt kan manipuleras.

6.1 Scenens delar

I scenen finns följande objekt:
- två lampor
- två böcker
- två soffor
- en vas
- ett bord
- en bokhylla

Dessutom har lagts därefter :
- 28 rektanglar för att representera en byggnad
- 9 rektanglar för att representera golvet

Vi ska presentera endast en liten del av scenen (fig. 6.1) för att visa funktioner som finns.

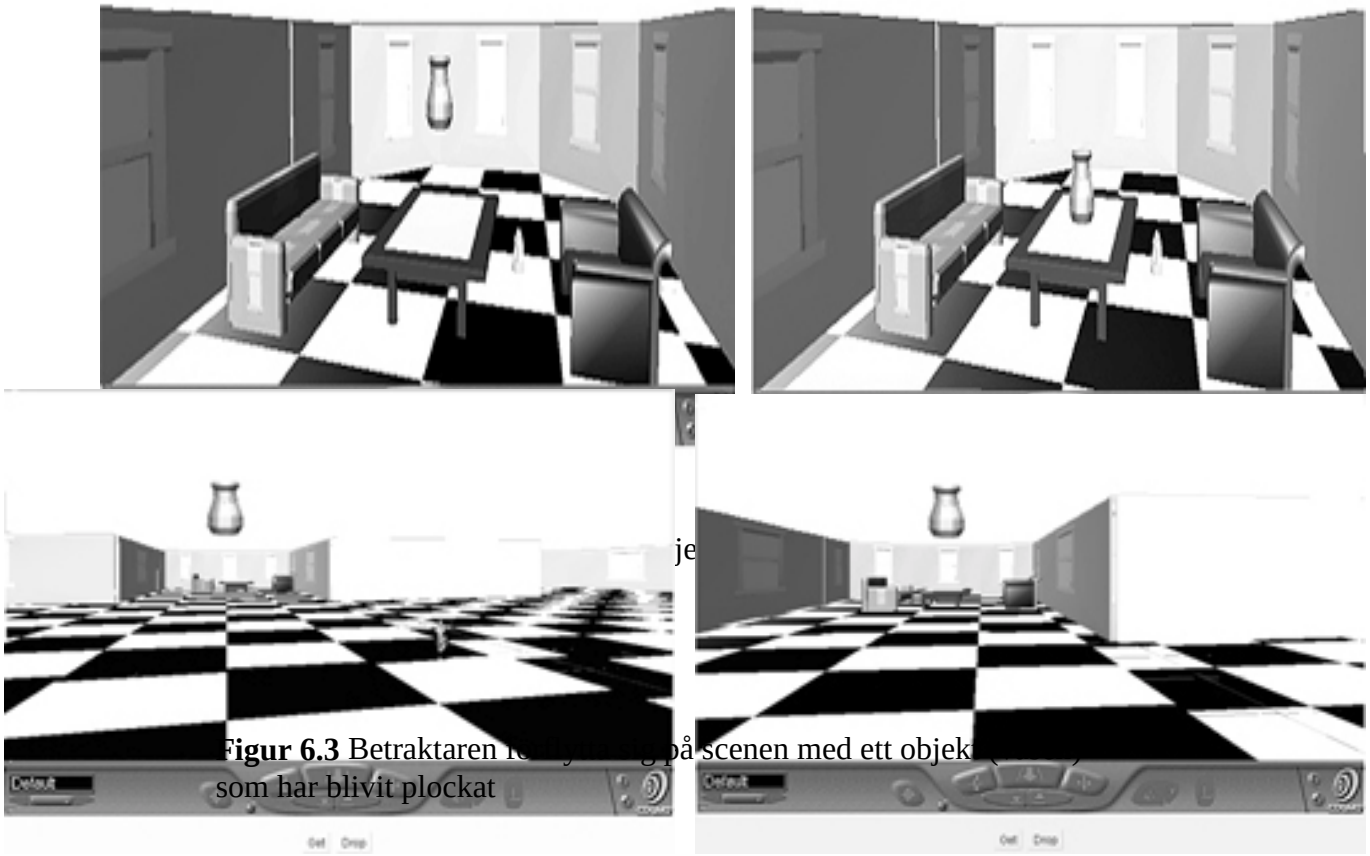


Figur 6.1 Exemplets delscen. Ett rum med två soffor och ett bord.

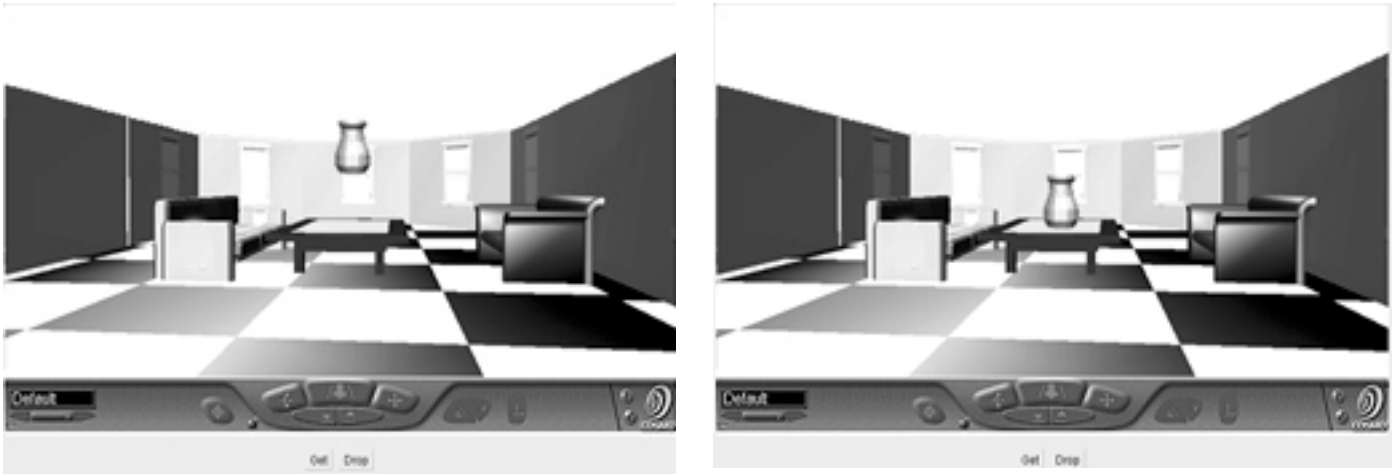
6.2 Miljö

- Scenen visades på en PC dator Pentium II 400 Mhz med 64 Mb, AGP 3D grafiskt kort.
- Som tilläggsprogram användes Cosmo Player [10]
- Webläsare: Netscape v 4.5 och Internet Explorer 5.
- Operativsystem: Windows 98.

6.3 Bildspel



Figur 6.3 Betraktaren [illegible] på scenen med ett objekt [illegible] som har blivit plockat



Figur 6.4 Betraktaren släpper ett objekt som var plockat.

1)

2)

3)

4)

Figur 6.5 Överföring av interaktion: 1) Användare förflyttar ett objekt (soffa) 2, 3, 4) Alla andra objekt som kolliderar med soffan också förflyttas tills translation blir icke-giltig p.g.a. kollisionen med väggen.

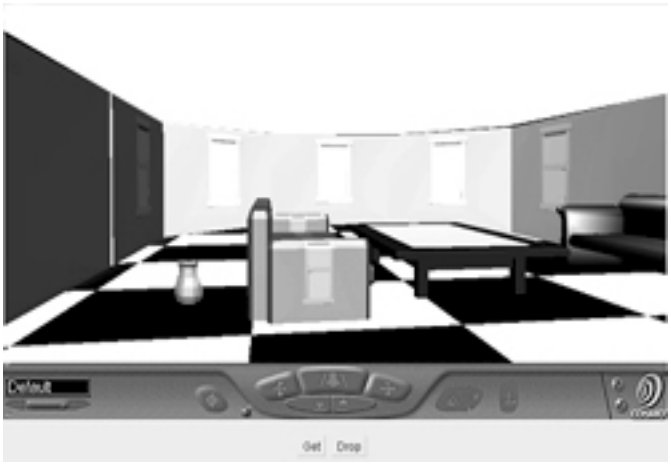
1)

2)

3)

4)

5)



6)

Figur 6.6 1-4) Att kasta ett objekt: bilder visar simulering av ett objekt som kastas .Rörelsebanan blir en parabel. 5-6)Visar simulering av friktion efter att objekt har uppnått marken. 6) Objektet ligger stilla.

6.4 Körningsresultat

Exemplet kan visas med Nescape v. 4.5 och Cosmo Player. Användare kan förflytta objekten, plocka dem, släppa dem etc. Svarstiden blev acceptabel. Däremot kraschar det Internet Explorer 5 när man trycker på ”drop ” knappen.

Det har visat att det största implementationsproblemet ligger i dess stabilitet. Att exemplet visas korrekt med Netscape v. 4.5 får oss att tänka att dessa problem är relaterade till webbläsaren och VRML-EAI mekanismen.

7 Resultat och diskussion

7.1 Allmänt

Resultatet av examensarbetet har blivit en implementering av ett 3D-gränssnitt med stöd för fysikalisk simulering och direkt interaktion. Med hjälp av gränssnittet kan användare förflytta tredimensionella objekt samtidigt som fysiska fenomen som gravitation, friktion och kollision mellan objekten approximativt simuleras. För att implementera simuleringsmotorn har använts en tillståndsmodell baserad i kollisionsdetektering mellan 3D -objekten. Vi har implementerat axel-orienterade lådor (AABB) algoritmen för att upptäcka kollsioner mellan tredimensionella föremål. Sedan har simuleringsmotorn integrerats med funktioner för att låta användare förflyta objekten i en VRML scen. Simuleringsmotorn och den interaktiva delen av gränssnittet har implementerats med Java (EAI). Dessutom har ett verktyg för att konvertera enkla VRML-scener så att de kan manipuleras genom vårt gränssnitt också implementerats.

7.2 Problem och brister

I examensarbetet används en relativt ny teknologi som fortfarande är under utveckling och som har sin största brist vad gäller stabilitet. Detta har märkts under implementationsfasen när webbläsaren ofta kraschade. Till slutet blev simuleringsmotorn beroende inte bara av typ av plattform utan även av webbläsarens version. Detta gjorde det svårt att göra en riktig bedömning av gränssnittet, t ex att veta hur många som var maximalt antal objekt som kunde finnas samtidigt i en scen utan att avsevärt minska svarstiden eftersom den mest berodde på datorns konfiguration och webbläsaren.

Vi har fokuserat på applikationens svarstid och använt snabba metoder och algoritmer. Detta har infört en approximativt faktor i simuleringen och beräkning av kollisioner mellan objekt som förmodligen kunde minskas med hjälp av bättre algoritmer.

Tyvärr kunde många av problemen relaterade till kommunikation mellan VRML och EAI inte undvikas. Speciellt de som uppstår mellan inkommande händelser från scenen och den EAI:s `callback` –metoden.

Andra former av interaktion som t ex rotation av de 3D-objekten och simulering av vinkelhastighet skulle ha tillagts. Tyvärr har problemen med stabilitet och brist på tid gjort att detta inte varit aktuellt.

7.3 Förslag

Jag tycker att gränssnittet behöver testas för att göra en bedömning av det .Det skulle hjälpa till avsevärt att upptäcka och korrigera utformningsproblem och på så sätt öka dess effektivitet. Vi har inte hunnit med detta och återigen har stabilitetsproblemen gjort det svårt att låta andra personer ge sin bedömning.

När det gäller simulering skulle man kunna tillämpas någon form av ”culling” [35] för att endast hantera objekt som är synliga.

Rotation av objekt skulle vara också intressant att tillägga till motorn. För att implementera rotationen av objekten skulle bland annat krävas en mer sofistikerad kollisionsdetekterings-algoritm än den som har använts här.

När det gäller interaktionsdelen skulle det vara intressant att implementera en begränsning för de funktioner som är relaterade till att välja och plocka objekten. Eftersom vi inte får plocka objekt om de ligger långt ifrån oss skulle det vara lämpligt att begränsa dessa funktioner till ett viss avstånd. För att informera användare om detta avstånd skulle kunna användas halvt transparenta grafiska element Alternativet är att använda ”gummiband” handen som kan plocka objekten utan hänsyn till avståndet mellan dem och betraktaren men som efter plockning kan placera dem nära till honom/henne.

Interaktion kunde också förbättras med införande av hjälpmarkeringar för betraktaren. Jag tycker att det viktigaste skulle vara att använda objektens skugga för att användare kan uppfatta djupet [7].

7.4 Interaktion och framtiden för Internetbaserad VR

Framtiden för Internetbaserade VR ligger i en stabilare lösning som vi redan har beskrivit, som är baserad på en mindre kärna som ska ge stöd till externa moduler. Dessa externa moduler kommer att implementera vissa typer av funktionalitet för en scen beroende på scenens syftet.[23]

X3D kommer delvis att ge stöd till implementering av kommersiella applikationer som i dag är mycket ovanliga [36]. Även om främsta orsaken verkar ligga i VRML:s stabilitet och komplexitet [36], är också problemet att bygga användbara 3d-gränsnitt en faktor som, enligt min uppfattning, kommer att påverka utvecklingen av effektiva tredimensionella applikationer på webben.

I detta sammanhang spelar detta examensarbete rollen att visa hur en typ av tredimensionell interaktion kan utformas och implementeras, inklusive en konverterare. Detta skulle kunna användas som riktlinjer för att bygga framtida X3D-baserade moduler.

Interaktions modellen som vi har byggt kunde implementeras med en mer stabil teknologi som den kommande X3D förmodligen blir och användas för implementering inom e-handel, arkitekturprojekt, husmöblering, spel etc. Vi hoppas därför att detta projekt blir ett steg på vä'gen till en allmän användning av 3D-visualisering på Internet.

Litteraturförteckning

Vad gäller hänvisningarna till WWW-sidor avser de alla tidpunkten slutet av september 1999.

[1] VRML Galleries and Worlds, http://home.hiwaay.net/~crispen/vrmlworks/worlds.html,

[2] Dr. Ping Dai, Dr. Gerhard Eckel, Dr. Martin Göbel m. f. ”Virtual Spaces VR Projection System Technologies and Applications”, 1997, http://viswiz.gmd.de/~eckel/publications/eckel97c/Tutorial.V4.html

[3] Volker Paelke, ”Visual Presentation Techniques in 3D Interaction”, http://www.c-lab.de/~vox/smart3D/differences.html

[4] Johnson, Chris ”On the Use of Pseudo-3D Images in Human Computer Interaction” , http://www.dcs.glasgow.ac.uk/~johnson/papers/validation/3dstripped.html

[5] Macintosh Human Interface Guidelines, http://developer.apple.com/techpubs/mac/HIGuidelines/HIGuidelines-2.html

[6] Mathew Lombard, Theresa Ditton, ”At the Heart of It All:The Concept of Prescence”, Department of Broadcasting, Telecomunications, & Mass media Temple University, http://www.ascusc.org/jcmc/vol3/issue2/lombard.html

[7] Ivan Popupyrevk, Research in 3D user interfaces, Looking forward, The IEEE Computer Society,s Student Newsletter,1995, Vol 3, N2 sid. 3-5. http://www.mic.atr.co.jp/~poup/research/publish/lookfwd.html

[8] McAndrew, Francis T., ”Environmental Psychology” Brooks/Cole Publishing Company, California, 1993, sid. 30-50.

[9] The Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997, http://www.vrml.org/technicalinfo/specifications/vrml97/index.htm

[10] Cosmo Player v. 2.1, http://www.karmanaut.com/cosmo/player/

[11] Blaxxun Contact v. 4.2, http://www.blaxxun.com/products/contact/index.html

[12] Cosmo Worlds, http://www.cosmosoftware.com

[13] Caligari Technologies, Truespace, http://www.caligari.com

[14] Kinetix, 3D Studio Max, http://www.ktx.com/3dsmaxr3

[15] Andrea L. Ames,David R. Nadeau, John L. Moreland, ”The VRML 2.0 Sourcebook”, John Wiley & Sons, Inc.,1997.

[16] Jed Hartman, Josie Wernecke, Silicon Graphics ”The VRML 2.0 Handbook: Building Moving Worlds on the Web”, Addison-Wesley, 1996.

[17] Per Holm, Objektorienterad programmering och Java, Studentlitteratur, 1999.

[18] Java Tutorial, http://java.sun.com/docs/books/tutorial/

[19] Chris Marrin, Proposals for a VRML 2.0 Informative Annex, External Authoring Interface, Silicon Graphics Inc. http://www.marrin.com/vrml/eaiwg/ExternalInterface.html

[20] Sun Microsystems Inc, Abstract Window Toolkit (AWT), http://www.javasoft.com/products/jdk1.1.2/docs/guide/awt/index.html

[21]Anthony Steed, The VRML External authoring Interface, University College London, http://www.cs.ucl.ac.uk/staff/A.Steed/eai/summary.html,

[22] Bernie Roehl, Justin Couch, Cindy Reed-Ballreich, Tim Rohaly, Geoff Brown, ”Late Night VRML2.0 with Java”, Ziff-Davis Press, 1997.

[23] Extensible 3D (X3D) Task Group, Web 3D Consortium, http://www.vrml.org/fs_x3d.htm

[24] Andy Witkin, Dave Baraff, Michael Kass, “An Introduction to Physically Based Modeling”, SIGGRAPH '95 course, http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/pbm.html

[25] James K. Hahn, Realistic animation of rigid bodies.In John Dill, editor, Computer Graphics (SIGGRAPH ’88 Proceedings), volume 22 sid. 299-308, 1988.

[26] Viktor J. Milenkovic, Position-Based Physics: Simulating the Motion of Mainly Highly Interacting Spheres and Polyhedra. Computer Graphics Proceedings, Annual Conferences Series, 1996.

[27] Volker Paelke, Cartoon Animation Principles, Visual Presentation Techniques in 3D Interaction, http://www.c-lab.de/~vox/smart3D/VPTin3Dinterfaces.html

[28]Ming C Lin, Stefan Gottschalk, “Collision Detection between geometric models:a survey”, University of North of Carolina.

[29] John Cohen, Ming C. Lin,Dinesh Manocha, Brian Mirtich, M. K. Ponamgi, John Canni, I-Collide, University of North Carolina, http://www.cs.unc.edu/~dm/I_COLLIDE.html

[30] Gino van den Bergen , SOLID: Software Library for Interference Detection, http://www.win.tue.nl/cs/tt/gino/solid/

[31] Gino van den Bergen, Efficient Collision detection of Complex Deformable Models using AABB Trees, Department of mathematics and Computing Science, Eindhoven University of Technology, 1998. http://www.win.tue.nl/cs/tt/gino/solid/#papers

[32] Russel Winder, Grabam Roberts, ”Developing Java Software”, John Wiley & Sons, Inc., 1998 sid. 603-605.

[33] Michael Chen,”A Study in Interactive 3D-Rotation Using 2-D Control Devices”Computer Graphics, volume 22,Number 4, August 1988.

[34] Satoshi Konno, CyberVRML97 for Java http://www.cyber.koganei.tokyo.jp/vrml/cv97/cv97java/index.html

[35] Stephen Chenney, Jeffrey Ichnowski, David Forsyth ”Efficient dynamics Modelling for VRML and Java”, University of California at Berkeley.

[36] Tony Parisi, X3D Market Requirements,Web 3D Consortium, http://www.vrml.org/news/x3d/mrd09.html

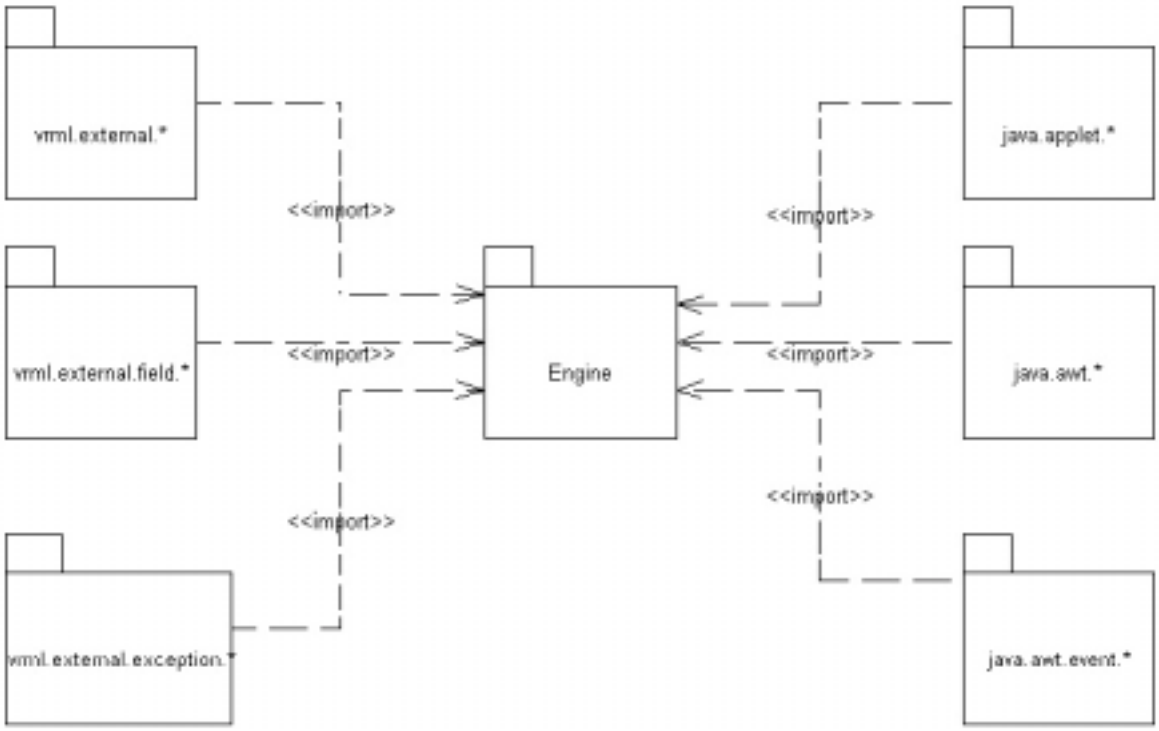
Bilaga A PROTO-deklarationen för fysikaliska objekt

```
#VRML V2.0 utf8


# This is the "Physical Node" where to store all that needs


PROTO PhyNode [

exposedField SFRotation thePRotation 0 1 0 0
exposedField SFVec3f    thePTranslation 0 0 0
exposedField SFRotation theRotation 0 1 0 0
exposedField SFVec3f    theTranslation 0 0 0
exposedField SFVec3f    theScale 1 1 1
exposedField MFString   theURL ""
exposedField SFVec3f    theDimensions 0 0 0
exposedField SFVec3f    theCenter 0 0 0
exposedField SFVec3f    theVelocityLinear 0 0 0
exposedField SFRotation theVelocityAngular 0 0 1 0
exposedField SFFloat    theMass 0
exposedField SFVec3f    theForce 0 0 0
exposedField SFVec3f    theSensorOffset 0 0 0
exposedField SFBool             theTouch TRUE
exposedField SFFloat    Selected 1
eventOut SFVec3f theSensorPos
eventOut SFBool  theSensorActive
eventOut SFBool  theOver
exposedField SFBool  moveable TRUE]

{

DEF parentTransform Transform {
 translation      IS thePTranslation
 rotation         IS thePRotation

 children         [

                  DEF planeSensor PlaneSensor {
                  enabled IS moveable
                  offset IS theSensorOffset
                  translation_changed IS theSensorPos
                  isActive IS theSensorActive
                 DEF obTran Transform { translation  IS theTranslation
                                        rotation     IS theRotation
                                        children    Inline {url IS theURL}}
                                                TouchSensor { enabled IS theTouch
                                                              isOver  IS theOver

                                                }]}
}
```

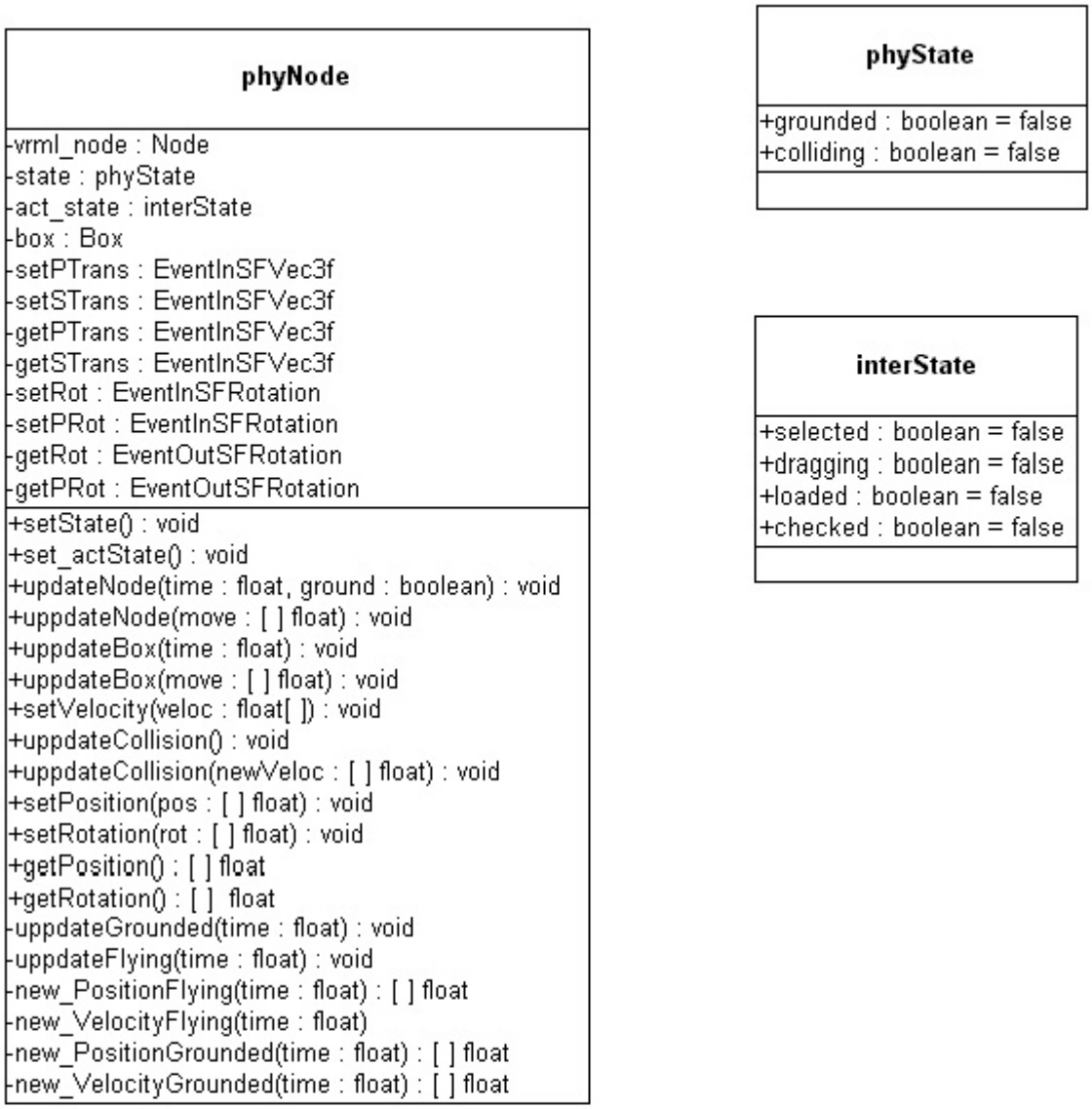
Bilaga B Diagram för Java klasser

Vi följer UML (Unified Modeling Language) regler för att rita diagram. En beskrivning på UML:s regler kan laddas från http://www.rational.com.

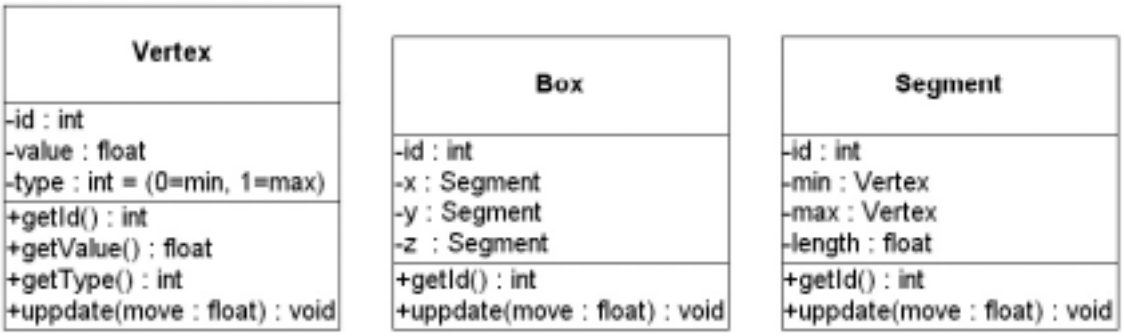
- **Motorn (Engine) paketet**



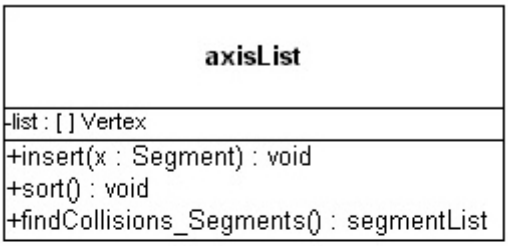
- **Representation av fysiska objekt och interaktionshanterare**

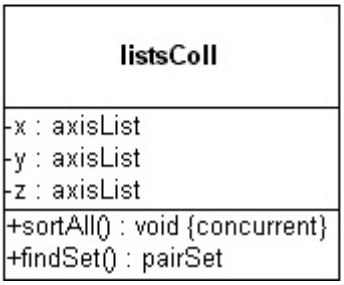


- **Lådor för AABB kollsionsdetekteringsalgoritm**

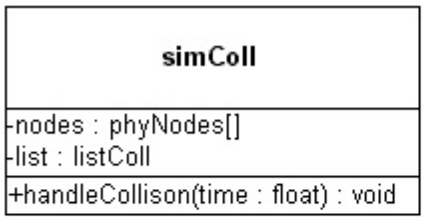


- **Intervallens listor**

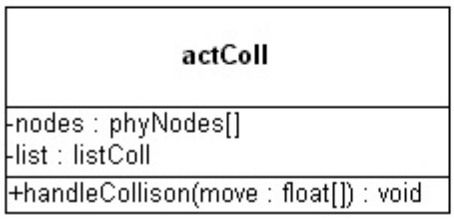




- **Kollisionsdetekterings mekanismer**

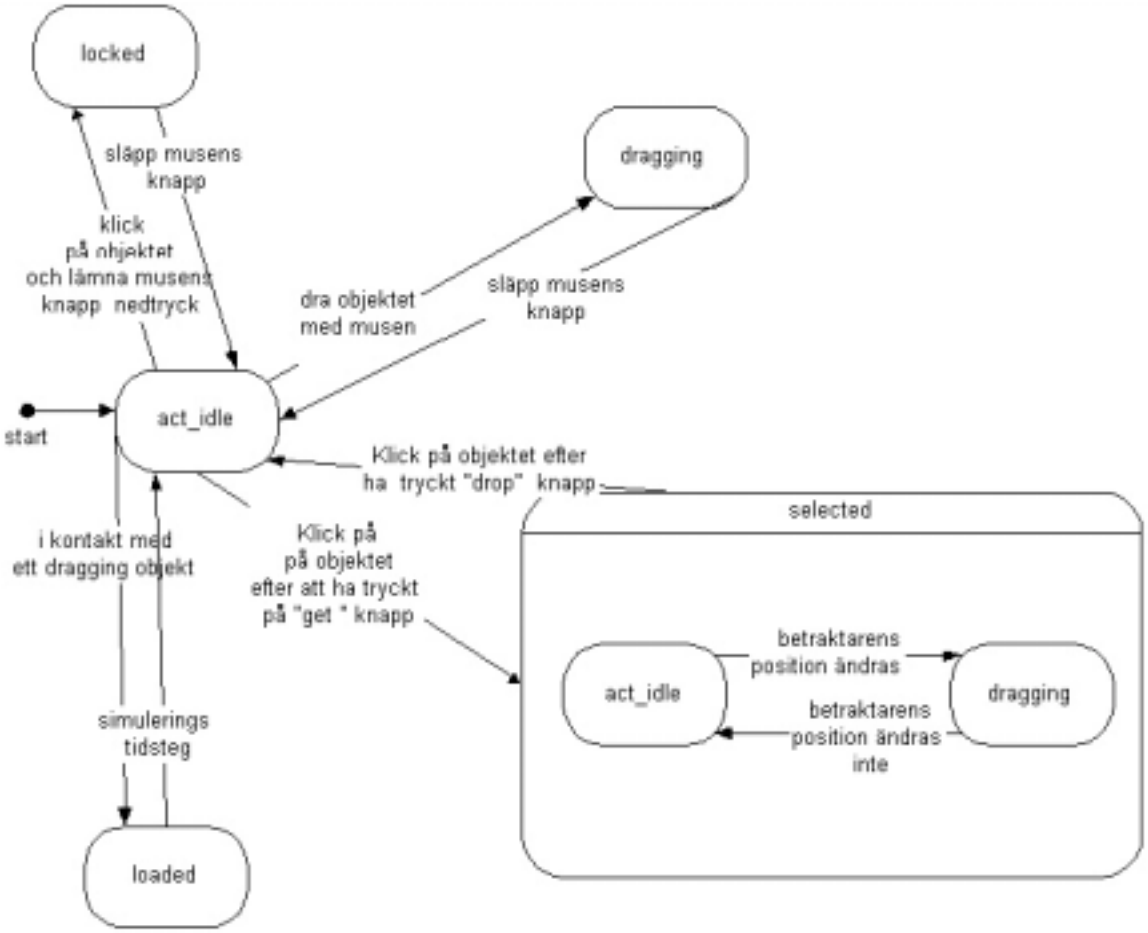


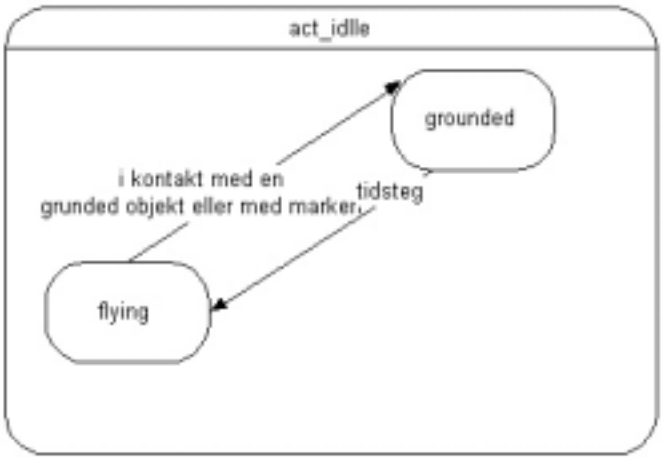
{Implementerar pseudokoden som finns i sida 30}



{Implementerar pseudokoden som finns i sida 36}

- **Objekts tillstånd**





Bilaga C Uppdateringsfuntioner

- Varje objekt kan vara i ”flying” eller ”grounded” tillstånd.(och andra interaktiva tillstånd)

- Varje objekt har en position som beskrivs med globala koordinater på scenen (x,y,z).(m)
  Innan uppdatering: (x_0, y_0,z_0)
  Efter uppdatering: (x_1,y_1,z_1)

- Varje objekt har en linjär hastighet som en tredimensionell vektor (Vx,Vy,Vz).(m/s)
  Innan uppdatering: (Vx_0, Vy_0, Vz_0)
  Efter uppdatering: (Vx_1,Vy_1,Vz_1)

- Till varje objekt tilldelas en tredimensionell vektor (Fx,Fy,Fz) som representera krafter som verkar på objektet. (N)

- Varje objekt har en massa :m (kg)

- Det finns en kofficient Kf för att simulera effekt av friktion mellan alla objekt, inklusive marken)

Pseudocode:

void **Uppdate** (tidsteg dt){

if Objekt_interact_tillstånd !=dragging || selected || locked {

Om objekt_tillstånd==grounded{

(Vx_1,Vy_1,Vz_1)= get_NewVelocityGrounded();
(x_1,y_1,z_1)=get_NewPositionGrounded((Vx_1,Vy_1,Vz_1),dt);
Uppdate_obj_inVRMLscene(Objekt, (x_1,y_1,z_1));
}
Om objekt_tillstånd==flying {

(Vx_1,Vy_1,Vz_1)=get_NewVelocityFlying((Vx_1,Vy_1,Vz_1),dt);
(x_1,y_1,z_1)=get_NewPositionFlying(dt);
Uppdate_obj_inVRMLscene(Objekt,(x_1,y_1,z_1));
}
}
}

(Vx_1,Vy_1,Vz_1) **get_NewVelocityGrounded**(){

$Vx_1=Vx_0*K_f+(Fx/m)*dt;$
$Vy_1=0;$
$Vz_1=Vz_0*K_f+(Fz/m)*dt;$
If Vx_1<Min_Velocity => $Vx_1=0;$
If Vz_1<Min_Velocity => $Vz_1=0;$

Returnera (Vx_1,Vy_1,Vz_1);}

(Vx_1,Vy_1,Vz_1) **get_NewVelocityFlying**(dt){

$Vx_1=Vx_0+(Fx/m)*dt;$
$Vy_1=Vy_0+(Fy/m)*dt;$
$Vz_1=Vz_0+(Fz/m)*dt;$

Returnera (Vx_1,Vy_1,Vz_1);
}

(x_1,y_1,z_1) **get_NewPositionGrounded**((Vx_1,Vy_1,Vz_1),dt){

$x_1=x_0+Vx_1*dt;$
$y_1=y_0;$
$z_1=z_0+Vz_1*dt;$
Returnera (x_1,y_1,z_1);
}

g : tyngdfaktorng= 9.82 N/kg;

(x_1,y_1,z_1) **get_NewPositionFlying**((Vx_1,Vy_1,Vz_1),dt){

$x_1=x_0+Vx_1*dt;$
$y_1=y_0+Vy_1*dt-g*(dt^2);$
$z_1=z_0+Vz_1*dt;$
Returnera (x_1,y_1,z_1);
}