Collision detection in Collaborative Virtual Environments

John Opdenakker

6 juni 2005

Samenvatting

Collaborative Virtual Environments (CVE's) zijn drie-dimensionale grafische werelden die er voor zorgen dat verscheidene geografisch verspreide gebruikers met elkaar kunnen communiceren door middel van een combinatie van visuele en auditieve voorstellingen.

Door de inburgering van het internet is nu wereldwijde samenwerking aan simulaties mogelijk. Binnen de gedeelde grafische wereld waarin de gebruikers (voorgesteld door avatars) met elkaar interageren, streeft men naar zoveel mogelijk realisme. Dit houdt onder andere in dat gebruikers niet door elkaar of andere objecten heen kunnen bewegen en dat iedere gebruiker altijd de meest recente status van de virtuele omgeving heeft. Deze problemen worden respectievelijk opgelost door een geschikt algoritme dat voor iedere gebruiker snelle collision detection realiseert en een goede synchronisatie van het netwerk.

Het eerste deel van dit werk bestaat uit een overzicht van bestaande collision detection algoritmes met hun voor- en nadelen en hun toepasbaarheid in specifieke applicaties. Daarna gaan we dieper in op de eigenschappen waaraan een Collaboratieve Virtuele Omgeving moet voldoen en worden pro's en contra's voor het gebruik van bepaalde netwerk protocollen en architecturen gegeven. De aandacht wordt ook gevestigd op efficiënt gebruik van het netwerk en de problemen die zich kunnen stellen bij de integratie van collision detection in zo een genetwerkte omgeving. Tot slot worden de resultaten uit het praktijkgedeelte van de thesis overlopen en geïllustreerd.

Woord vooraf

Er zijn een aantal mensen die ik graag zou bedanken voor de hulp en steun gedurende de periode dat ik aan de thesis gewerkt hebt. Allereerst wil ik Prof. dr. Wim Lamotte bedanken voor de algemene coördinatie van mijn thesis. Mijn dank gaat ook uit naar mijn begeleiders Pieter Jorissen en Tom van Laerhoven voor de hulp en feedback op problemen betreffende de tekst en implementatie. Last but not least, wil ik de mensen die me dierbaar zijn danken voor het geloof in mijn kunnen en de steun tijdens de moeilijke momenten. In het bijzonder zijn dit mijn ouders en mijn vriendin, maar ook zeker niet te vergeten mijn vrienden die voor me klaar stonden.

Inhoudsopgave

Inleiding

Collision detection

Collision detection vormt een fundamenteel probleem in computeranimatie, computer graphics, physically-based modeling en robotics. Al deze Virtual Reality toepassingen voorzien een computer-gegenereerde wereld waarin de gebruikers kunnen interageren met objecten en virtuele agents. Bovendien moeten deze applicaties hoge en constante frame-rates bieden. Het volstaat niet om afbeeldingen gedetailleerd te renderen om realisme te garanderen in deze applicatiedomeinen. Fysische interacties in een simulatie kunnen botsingen veroorzaken. Deze interacties moeten nauwkeurig gemodelleerd worden door de botsingen en contactpunten op te sporen, zelfs als de betrokken objecten groot en complex zijn. Om te voorkomen dat objecten interpenetreren, wordt een collision detection algoritme gebruikt.

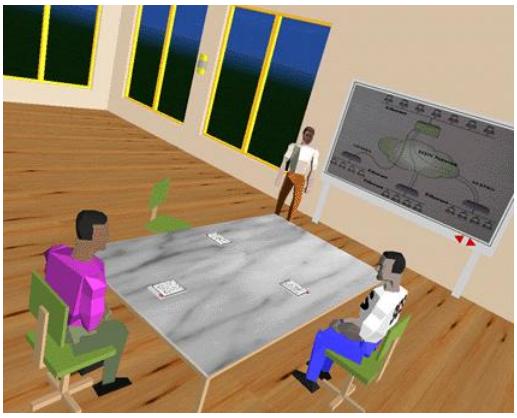
In real-time applicaties vormt collision detection vaak een bottleneck. Tegenwoordig is real-time computeranimatie geëvolueerd tot het modelleren van scenes met een groot aantal complexe objecten. Bijgevolg moeten alle problemen die betrekking hebben op fysische modellering verder onderzocht worden om aan deze nieuwe benodigdheden te voldoen. In het bijzonder het probleem van collision detection voor vervormbare objecten. De meeste bestaande oplossingen kunnen niet triviaal uitgebreid worden, omdat ze sterk gebaseerd zijn op de aanname dat de vorm van het object vast is.

De evolutie in software applicaties loopt parallel met de ontwikkeling van grafische kaarten en processoren. Deze evolutie is zeer goed vast te stellen door bijvoorbeeld huidige computer games te vergelijken met die van vroeger. De eerste games leken allemaal op elkaar en waren niet veel meer dan een tweedimensionale omgeving bestaande uit simpele objecten en een simpel character dat zich er in kon voortbewegen. In uitzonderlijke gevallen bestond er een multiplayer mode via een intern netwerk. Tegenwoordig is de diversiteit veel groter: Real Time Strategy (RTS), First Person Shooters (FPS) en Role Playing Games (RPG) zijn maar enkele voorbeelden van populaire soorten games. Bijna al deze games voorzien complexe 3D graphics en kunnen zowel via een intern netwerk als via internet gespeeld worden. Om ze real-time te runnen is er niet enkel nood aan efficiënte graphics algoritmes, maar moet ook de netwerkcommunicatie real-time verlopen. In figuur 1 is de evolutie van de graphics in het spel 'Prince of Persia' goed te zien [20].

Figuur 1: Ter illustratie: Een screenshot van de eerste prince of persia uit 1989 links en rechts een screenshot uit 'Prince of Persia : The Sands of Time' van 2003 [20].

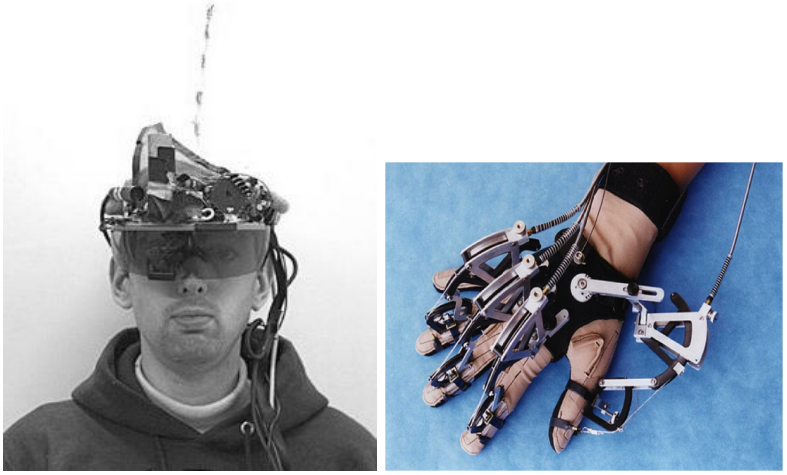
Genetwerkte virtuele omgevingen

Een genetwerkte virtuele omgeving (NVE) is een software systeem waarin verschillende gebruikers met elkaar interageren in real-time. Deze gebruikers kunnen verspreid zijn over de volledige wereld [82].



Figuur 2: Een conferentie in een virtuele omgeving met 3 mensen die gelijktijdig verbonden zijn met het DIVE systeem. De mensen worden voorgesteld door avatars en kunnen interageren, praten en bewegen in een gedeelde omgeving [10].

Voor de eerste NVE's moeten we teruggaan tot de jaren 80. In 1983 startte men met de ontwikkeling van een gedistribueerde militaire virtuele omgeving voor het Amerikaanse leger, genaamd SIMNET (simulator networking). In het begin van de jaren 90 kwam er met DIS (Distributed Interactive Simulation) een opvolger voor SIMNET [55]. Een aantal jaren later waren de eerste genetwerkte games een feit (bijvoorbeeld Doom en Counter Strike). DIVE in 1992 (figuur 2) en Paradise in 1993 waren de eerste NVE's die gebruikt werden voor academisch onderzoek [49]. Recenter is het gebruik van chatsystemen met 3D grafische browsers. Uit de geschiedenis van NVE's blijkt dat zulke systemen moeilijk te ontwikkelen zijn aangezien het tezelfdertijd gedistribueerde systemen, 3D grafische applicaties en real-time interactieve programma's zijn.



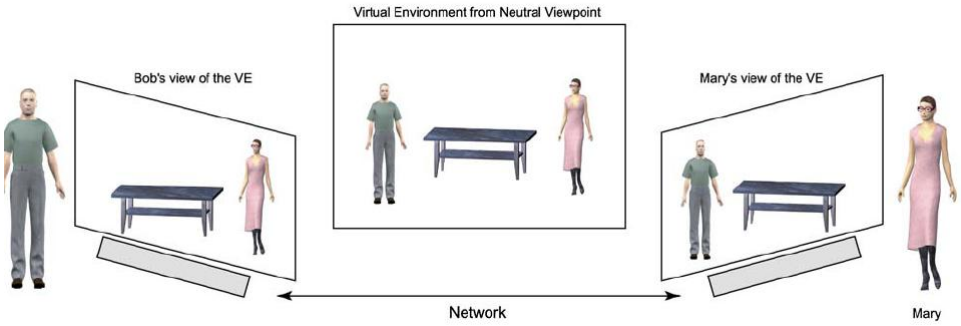
(a) Head mounted device [10]. (b) Data glove [7].

Figuur 3: Voorbeelden van devices gebruikt voor VR toepassingen.

Hedendaagse VR toepassingen zijn op te splitsen in twee categorieën: volledig immersieve omgevingen en half-immersieve desktop omgevingen. Volledig immersieve omgevingen maken gebruik van head-mounted displays (figuur 3(a)) en motion trackers. De bewegingen en oriëntatie van de gebruiker worden gedetecteerd door de motion trackers en de display wordt overeenkomstig geupdate. Manipulatie van objecten in de wereld wordt vaak mogelijk gemaakt door de gebruiker een dataglove (figuur 3(b)) te laten dragen. In desktop VR omgevingen hebben de gebruikers een beeldscherm dat de 3D graphics van de virtuele wereld toont. Interactie met de virtuele wereld gebeurt door middel van toetsenbord, muis, joystick, etc [42].

Het idee van collaborativiteit in een NVE is zeer eenvoudig. Twee of meer gebruikers kunnen een virtuele omgeving waarnemen op hun workstations. Ze hebben ieder een lokale kopie van deze omgeving. Als een gebruiker acties uitvoert op een workstation, worden deze via het netwerk naar de andere workstations gezonden om synchronisatie tussen alle kopieën van de virtuele omgeving te garanderen. De gebruikers maken deel uit van de VE met hun eigen unieke viewpoint en worden voorgesteld door een grafische belichaming (avatar). Dit maakt het mogelijk dat iedere gebruiker de andere gebruikers met hun acties kan waarnemen [54]. Figuur 4 laat de viewpoints van 2 gebruikers met verschillende avatars zien.

De mogelijkheid om collaboratief te werken vanop verschillende locaties is een groot voordeel voor vele bedrijven en instanties. Werknemers kunnen dikwijls veel goedkoper opgeleid worden en afstand vormt niet langer een barriere. Onderling overleg tussen mensen met hetzelfde beroep wordt mogelijk via teleconferencing. Dit is bijvoorbeeld zeer nuttig in de medische sector, waar artsen hun kennis kunnen delen of tot een second opinion kunnen komen. Een van de nieuwste trends is collaboratieve e-commerce. Op die manier hebben kleinere bedrijven de mogelijkheid om zich te groeperen en te concurreren tegen multinationals.



Figuur 4: Viewpoints van de gebruikers Bob en Mary die beiden door een avatar worden voorgesteld [54].

De rest van deze thesis is opgedeeld in 3 hoofdstukken die respectievelijk collision detection algoritmes, de netwerk aspecten van collaboratieve virtuele omgevingen en de gemaakte implementatie bespreken.

Hoofdstuk 2 beschrijft aan welke vereisten collision detection algoritmes moeten voldoen met het oog op real-time collision detection. In het bijzonder wordt er besproken waarom er een opdeling in een groffe fase en een accuratere fase van collision detection gemaakt wordt en hoe verschillende soorten bounding volumes hierbij gebruikt kunnen worden.

Hoofdstuk 3 beschrijft wat collaboratieve virtuele omgevingen zijn en de voor- en nadelen van verschillende netwerkarchitecturen en protocollen om zo een omgeving te creëren. Bovendien worden er oplossingen geformuleerd om de gevolgen van netwerk delays zo goed mogelijk voor de gebruiker te verbergen. Tenslotte worden ook de problemen die zich voordoen bij het integreren van collision detection in zo een omgeving ter sprake gebracht.

Hoofdstuk 4 beschrijft de proefondervindelijke resultaten en de aanpak van de implementatie van een CVE met collision detection.

Hoofdstuk 1

Collision detection

1.1 Inleiding

Als verschillende objecten bewegen in een simulatie is er een kans dat ze interpenetreren. Dit is meestal een ongewenste toestand, in het bijzonder als het doel van de simulatie is om een realistische wereld te modelleren. Collision detection zorgt er voor dat objecten die gaan colliden tijdig opgespoord worden om zo interpenetraties te voorkomen.

De toepasbaarheid van een collision detection algoritme is afhankelijk van een aantal factoren. Een van deze factoren is de voorstelling van de input objecten van het algoritme. De meeste algoritmes leggen eisen op waaraan de input geometrieën moeten voldoen. Sectie 1.2 verschaft inzicht in de verschillende manieren waarop objecten kunnen voorgesteld worden en de voor-en nadelen van iedere voorstellingswijze.

Om praktisch uitvoerbaar te zijn moet een collision detection algoritme een aanvaardbare tijdscomplexiteit hebben. Wat aanvaardbaar betekent, hangt dikwijls van de specifieke toepassing af. Maar het is duidelijk dat het paarsgewijs testen van alle objecten in een simulatie zelden tot een aanvaardbare uitvoeringssnelheid zal leiden. Bij een groot aantal objecten vormt deze aanpak duidelijk een bottleneck. Om dit probleem te omzeilen voorzien alle collision detection algoritmes van praktisch nut een opsplitsing in twee fases.

De eerste fase, ook wel broad phase genoemd, voert een aantal tests uit om interfererende objecten te identificeren in de volledige workspace [48] (sectie 1.3). Meestal worden deze testen uitgevoerd tussen bounding volumes van de betrokken objecten i.p.v. tussen de objecten zelf. Het verlies van accuraatheid door overlaps te testen tussen bounding volumes in plaats van tussen de objecten zelf wordt gecompenseerd door een toename in efficiëntie van de overlap tests. In sectie 1.3.1 worden enkele frequent gebruikte bounding volumes, hun voor- en nadelen en toepasbaarheid in bepaalde situaties, besproken.

De tweede, narrow phase, voorziet accuratere tests die interfererende delen van objecten opspoort. Ieder object wordt voorgesteld als een hiërarchie van bounding volumes. Door deze hiërarchieën onderling te testen op overlappingen kan

nauwkeuriger bepaald worden welke delen van de objecten interfereren. Voorbeelden van bounding volume hiërarchieën zijn sphere trees, OBB trees en C-trees (sectie 1.4.2).

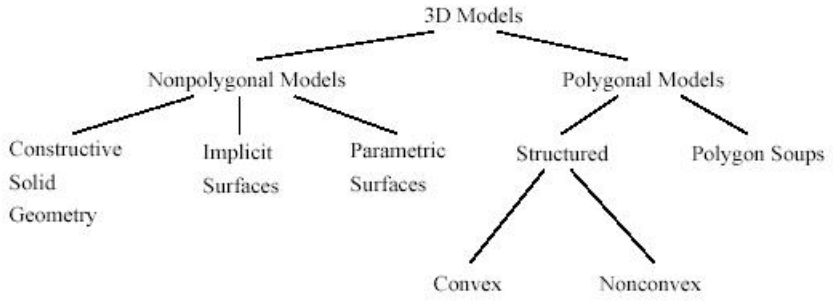
Sectie 1.5.1 beschrijft een aantal collision detection algoritmes die gebruikt kunnen worden indien de input enkel uit convexe objecten bestaat. Indien de input uit meer algemene objecten bestaat, zijn de algoritmes uit sectie 1.5.2 aangewezen. Deze twee soorten van algoritmes hebben het nadeel dat ze vervorming van objecten niet ondersteunen. Een algoritme dat wel in staat is om met vervormbare objecten te werken wordt gepresenteerd in sectie 1.5.3.

Een andere belangrijke factor met het oog op collision detection is collision scheduling. Een aantal frequent gebruikte collision detection queries en methodes om te bepalen wanneer deze queries aangeroepen moeten worden, staan beschreven in sectie 1.6.

Een overzicht van een aantal free license collision detection libraries is terug te vinden in sectie 1.7. Tot slot volgt in sectie 1.8 een bespreking hoe collision detection in enkele games wordt gerealiseerd.

1.2 Model voorstellingen

3D modellen kunnen op verschillende manieren worden voorgesteld. Hier bespreken we enkele van de voornaamste voorstellingswijzen. Een mogelijke indeling van 3D model voorstellingen wordt weergegeven in figuur 1.1.



Figuur 1.1: Mogelijke taxonomie van 3D model voorstellingen [63].

1.2.1 Polygonale modellen

Een verzameling van veelhoeken of vlakken die een object vormt, wordt een polygonaal model genoemd. Polygonale modellen zijn de meest voorkomende modellen in computer graphics en modeling. Omdat ze wiskundig zo simpel zijn, is een snelle rendering mogelijk. Polygonen vormen een soort basis voor alle computer modellen. Bijna iedere modelvoorstelling kan met willekeurige precisie geconverteerd worden in een polygonale mesh. Dit is noodzakelijk als de complexiteit van het model in kwestie interactieve rendering door de graphics

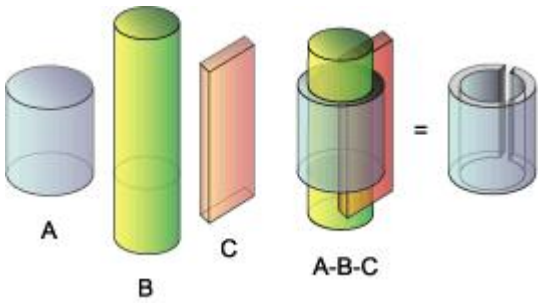
hardware onmogelijk maakt. De meest algemene klasse van polygonale modellen is de polygon soup. Het is een verzameling van polygonen die niet geometrisch verbonden zijn en waarvan geen topologische informatie gekend is. Als de polygonen een gesloten geheel vormen, dan spreken we van een solid model. Verder wordt er nog een onderscheid gemaakt tussen convexe en niet-convexe solid objecten [66, 63]. In figuur 1.1 is duidelijk deze onderverdeling van polygonale modellen te zien.

1.2.2 Niet polygonale modellen

Constructive Solid Geometry

Constructive Solid Geometry (CSG) is een frequent gebruikte techniek om modellen op een constructieve wijze voor te stellen. Een impliciet model wordt gevormd uit eenvoudige primitieve objecten door booleaanse operaties (unie, intersectie en verschil) te combineren en gebruik te maken van zogenaamde blending functies. Mogelijke primitieve vormen zijn cylinders, kegels, blokken, etc. Iedere primitieve wordt geparametriseerd zodat er verschillende instanties met specifieke eigenschappen (positie, oriëntatie, etc.) van kunnen gemaakt worden. Hiërarchische groepering van primitieven is ook mogelijk. Rigide bewegingen [1] en schalering veroorzaken transformaties van groepen of instanties van primitieven. De getransformeerde instanties kunnen ook gecombineerd worden door de booleaanse operaties: unie, intersectie en verschil [79, 63].

Het vormen van complexe objecten door eenvoudigere op een bepaalde manier te combineren is erg intuïtief. CSG is vooral geschikt voor het ontwerpen van objecten in CAD. Sommige operaties, zoals het reconstrueren van objecten aan de hand van hun gesamplede data, zijn moeilijk te beschrijven met CSG. Het is ook niet altijd vanzelfsprekend om een gepaste grens- of oppervlak voorstelling te berekenen uit CSG modellen [40, 63].



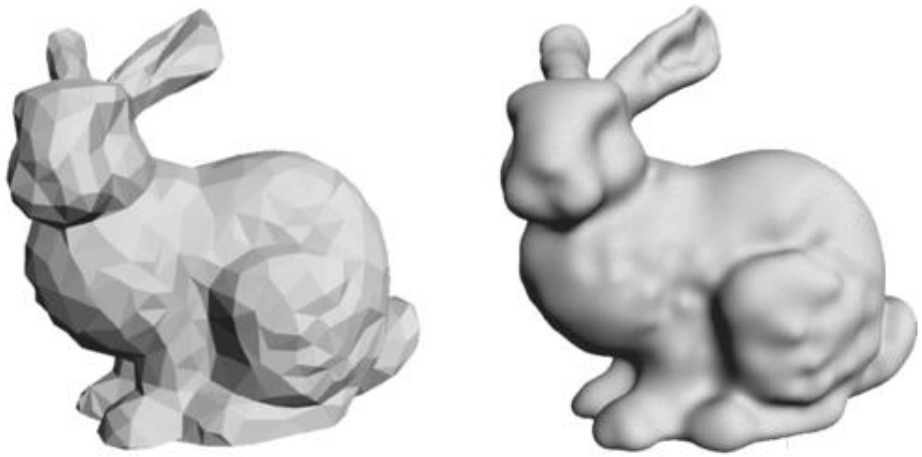
Figuur 1.2: Constructive Solid Geometry [6].

Impliciete oppervlakken

Veel gladde en vervormbare objecten zijn moeilijk of onvoldoende nauwkeurig voor te stellen door middel van combinaties van primitieve basisvormen. Im-

[1] Rigide bewegingen bestaan uit een combinatie van rotaties en translaties.

pliciete oppervlakken maken het mogelijk om ook complexere oppervlakken te renderen zonder ze eerst te moeten trianguleren [37]. Om impliciete oppervlakken te definieren worden impliciete functies gebruikt. Een impliciete functie beeldt een punt in een drie dimensionale ruimte af op een reëel getal. De functie $f(x, y, z) = 0$ definieert het oppervlak. Alle output waarden van f die kleiner of gelijk zijn aan 0 liggen binnen, respectievelijk op dat oppervlak. Resultaten groter dan 0 impliceren dat het betreffende punt buiten het oppervlak ligt [63, 84].



Figuur 1.3: Het verschil in gladheid is duidelijk te merken: aan de linkerkant is een polygonaal oppervlak gebruikt en aan de rechterkant een interpolerend impliciet oppervlak [88].

Blending functies worden gebruikt om de overgang tussen de verschillende primitieven, die gegenereerd zijn door de impliciete oppervlakken, gladder te maken. Enkele veel gebruikte blending functies zijn de Gaussiaanse en de polynomiale [84].

Parametrische oppervlakken

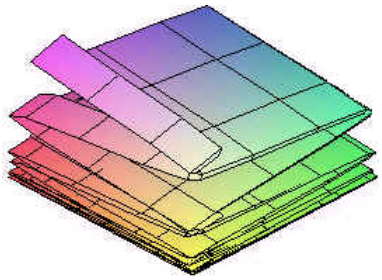
Net zoals impliciete oppervlakken en CSG modellen behoren de parametrische oppervlakken tot de niet polygonale modellen (figuur 1.1). Parametrisch oppervlakken zijn afbeeldingen van een deelverzameling van een vlak naar een drie dimensionale ruimte. Ze stellen geen complete solide modellen voor, maar geven een beschrijving van de grens van het oppervlak. Dit maakt dat ze gemakkelijker gepolygonaliseerd en gerenderd kunnen worden dan impliciete oppervlakken [63].

De punten (u,v) uit een twee dimensionale ruimte invullen in:

$\vec{r}(u, v) = x(u, v)\vec{i} + y(u, v)\vec{j} + z(u, v)\vec{k}$

geeft een verzameling positievectoren voor de punten op het oppervlak dat geparametriseerd wordt. De parametrische functies die zo een oppervlak definieren worden genoteerd als:

$x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$ [44].
Als $x = \sin u$, $y = \cos v$ en $z = e^{2\times \sqrt[3]{u}+2\times \sqrt[3]{v}}$ dan krijgen we het resultaat zoals in figuur 1.4 staat afgebeeld.



Figuur 1.4: Parametrisch oppervlak [44].

Non-Uniform Rational B-Splines (NURBS) zijn een veel gebruikte soort van parametrische oppervlakken. NURBS bieden een intuïtieve methode om oppervlakken te creëren met een minimale opslag van data. Ze zijn erg geschikt voor het modelleren van gladde overgangen in de modellen [80].

1.2.3 Gebruikte modellen

In de rest van deze tekst worden enkel algoritmes besproken die als input polygonale modellen hebben. Het gebruik van niet-polygonale modellen valt buiten het bestek van de thesis.

1.3 Broad phase collision detection

In een omgeving waarin n verschillende objecten aanwezig zijn, moet collision detection uitgevoerd worden tussen $O(n^2)$ verschillende objectparen. De meeste efficiënte collision detection systemen gebruiken 2 fases om de kosten van de detectie te beperken. De eerste, zogenaamde broad phase, stap verwerpt de meeste objectparen door een triviale test gebaseerd op bounding boxes, bounding spheres, octrees, e.d. uit te voeren. De paren die deze fase overleven worden doorgegeven aan de tweede, zogenaamde narrow phase, stap. Deze sectie beschrijft een aantal frequent gebruikte bounding volumes en technieken die bounding volumes gebruiken om een grote hoeveelheid objecten vroegtijdig te cullen. Ook bij narrow phase collision detection (zie sectie 1.4) zullen bounding volumes gebruikt worden.

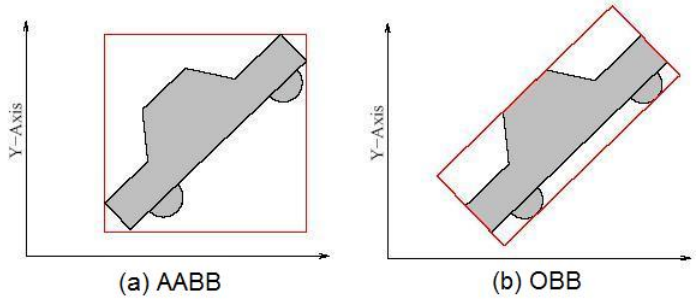
1.3.1 Bounding volumes

Het testen van collisions tussen twee objecten bestaat uit het onderling testen van alle vlakken van beide objecten. Om de collision tests sneller uit te kunnen voeren, worden dikwijls omhullende volumes gebruikt om ieder object voor te

stellen. Deze vereenvoudigde volumes zijn uiteraard minder precies dan de objecten zelf en worden vaak gebruikt als een snelle test om uit te maken of meer gedetailleerde tests noodzakelijk zijn. Er zijn verschillende soorten bounding volumes bekend. Afhankelijk van de vorm van de objecten zal er een geschikt bounding volume gekozen worden.

Bounding boxes

Er zijn twee soorten bounding boxes die vaak in bestaande applicaties gebruikt worden voor collision detection: axis-aligned bounding boxes (AABB's) en oriënted bounding boxes (OBB's). AABB's kunnen niet met de objecten mee roteren. Bijgevolg zijn de boxen en overlap regio's dikwijls te groot. OBB's kunnen wel met het object mee roteren, maar zijn niet geschikt voor dynamische modellen [5]. OBB's impliceren gecompliceerde, dure overlap tests. De

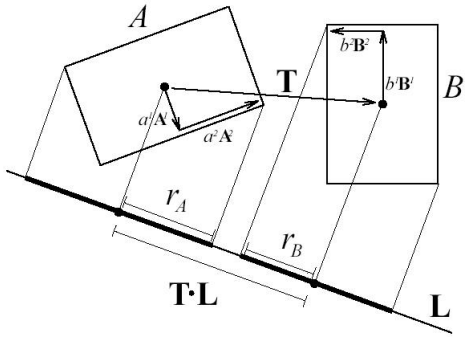


Figuur 1.5: 2D Axis Aligned Bounding Box (AABB) en 2D Oriented Bounding Box (OBB) omhullen een volledig object [38].

specifieke vereisten van de applicatie bepalen of deze kost tolereerbaar is. Als veel polygonen axis-aligned zijn is het interessanter om de minder geheugenintensieve AABB's te gebruiken en zal er weinig verschil in performantie zijn [69].

Separating-axis overlap tests voor bounding boxes

Paren van bounding boxes rechtstreeks testen op overlappingen is meestal een zware operatie. Een snellere methode is het projecteren van de boxen op een as in de ruimte. Voor een 3D box zijn er 15 zulke assen. Deze assen zijn orthogonaal ten opzichte van een vlak of een zijde van beide polytopen. Iedere box vormt een interval door de projectie op een as. Als deze intervallen niet overlappen dan wordt deze as een **separating axis** genoemd voor de boxen en zijn ze disjoint. Als één van de 15 assen een scheidende is, is overlapping uitgesloten. Overlappen de intervallen wel dan moeten verdere tests uitsluitsel bieden [52].



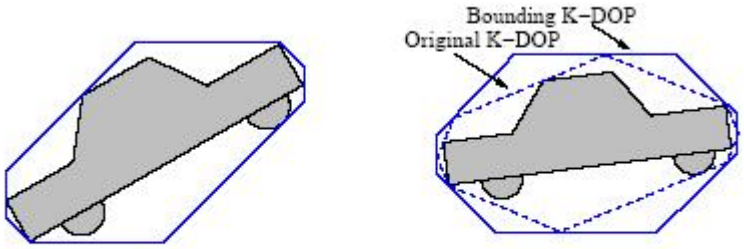
Figuur 1.6: L is een scheidende as voor de OBB's A en B [52].

Discrete Orientation Polytopes (k-DOPs)

Een k-DOP is een convexe polytoop met, gegeven een verzameling van k 3D vectoren, voor alle k zijn vlakken normalen die naar buiten gericht zijn ten opzichte van deze k vectoren. Een AABB is een speciaal geval van een k-DOP voor een gegeven verzameling van k=6 normalen evenwijdig met de coördinaatassen [9].

k-DOPs houden de kost voor overlap tests tussen primitieven in een hiërarchie zo laag mogelijk door tight fitting bounding boxes te gebruiken. Om te verzekeren dat de oriëntatie van ieder bounding vlak beperkt is tot de k discrete richtingen, moet de grens herberekend worden als het object roteert. k-DOPs berekenen gedurende animaties is een erg dure operatie.

Het is mogelijk om de k-DOPs van de objecten vooraf te berekenen en bij rotaties nieuwe er rond te creëren [57]. De resulterende k-DOPs bakenen het origineel gegarandeerd af, zij het niet zo strak [38].



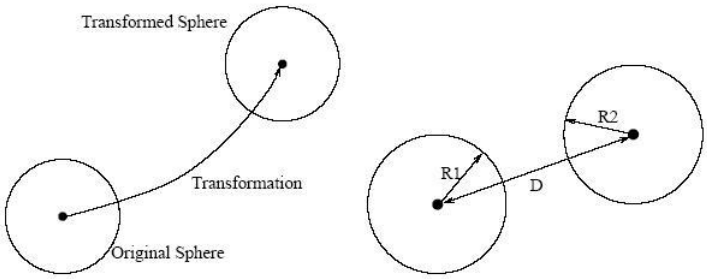
(a) Original 8-DOP (b) Updated 8-DOP

Figuur 1.7: 8-DOPs om de grens van een auto te benaderen [38].

Bounding spheres

Een bounding sphere is een hypothetische bol die een object volledig omsluit en gedefinieerd wordt door een 3D coördinaat (centrum) en een scalaire straal die de maximum afstand van het centrum tot elk punt van het object definieert [72].

Bounding spheres zijn invariant voor rotaties, waardoor een simpele transformatie van hun centrum volstaat bij rotatie [38]. Intersectie tussen twee spheres kan snel getest worden. Twee stationaire spheres A en B raken elkaar als de afstand tussen de middelpunten van de 2 cirkels, zoals gegeven in vergelijking 1.1, kleiner of gelijk is aan de som van hun stralen. Vermenigvuldigingen vergen minder rekenkracht dan kwadraten. Bijgevolg nemen we bij het berekenen van de afstand het kwadraat van de som van de stralen in plaats van de vierkantswortel [89, 5].



(a) Update (b) Intersection

Figuur 1.8: a) Update van bounding sphere bij rotatie. b) Intersectie test tussen 2 spheres [38].

$$Afstand = \sqrt{(A.x - B.x)^2 + (A.y - B.y)^2} \tag{1.1}$$

Swept Sphere Volumes

Deze bounding volumes komen overeen met geometrische vormen samengesteld uit een basis primitieve vorm die naar buiten toe groeit met een offset. Zo een vorm kan gezien worden als de Minkowski som[2] of convolutie van de primitieve vorm en een sphere. Het bounding volume correspondeert dus met de verzameling van punten die door de sphere worden bestreken als zijn centrum wordt bewogen over alle punten van de primitieve vorm. Enkele basis primitieve vormen zijn:

- Point Swepth Spheres: De basis primitieve vorm is een punt. Het resulterende bounding volume is een bol die wordt voorgesteld door een punt en een straal.

[2]De som van verzamelingen A en B in een vectorruimte is gelijk aan $\{a + b : a \in A, b \in B\}$[14]

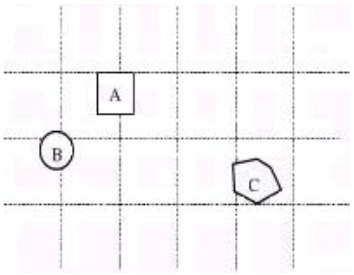
- Line Swepth Spheres: De basis primitieve vorm is een lijnsegment. Het resulterende bounding volume vormt een cylinder voorgesteld door een lijnsegment, centrum en straal.
- Rectangle Swepth Spheres: De basis primitieve vorm is een willekeurige rechthoek in 3D. Het resulterende bounding volume is een afgeronde box voorgesteld door de rechthoek, zijn centrum en een straal [58].

Een rectangle swepth sphere voorziet, net zoals OBB's, kwadratische convergentie naar de onderliggende geometrie. De convergentiesnelheid van een line swepth sphere ligt tussen deze en lineaire convergentie verkregen met spheres. De kost van de overlap tests hangt af van de betrokken basis primitieven. Line swepth spheres zijn geschikt voor lange dunne gebieden; rectangle swepth spheres voor vlakke gebieden [38].

1.3.2 Bounding box methodes

Uniforme grid

Om te voorkomen dat er voor n bounding boxes $O(n^2)$ intersectie tests uitgevoerd worden, kan men een ruimte verdelen in een reguliere grid van cellen (figuur 1.9). Voor iedere bounding box wordt bepaald welke cellen ze overlapt. Enkel bounding boxes die een overlap regio hebben[3] komen nog in aanmerking voor intersecties [5].



Figuur 1.9: Uniforme grid.

Het is belangrijk om een geschikte celgrootte te kiezen. Als de cellen te groot zijn, behoren veel objecten tot dezelfde cel en blijft de kost voor collision detection $O(n^2)$. Als de cellen te klein zijn neemt ieder object veel cellen in. Updaten van objecten wordt dan een dure operatie [59].

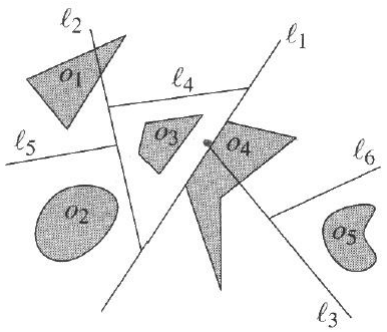
Sweep-and-prune

Het sweep and prune algoritme is gebaseerd op het sorteren van de bounding boxes van de objecten. Een methode om 3D bounding boxes te sorteren is door de dimensie te reduceren. We projecteren iedere box op de x,y en z assen. Een paar bounding boxes kan overlappen als en slechts als hun intervallen overlappen

[3] De overlap regio wordt bepaald door de gemeenschappelijke cellen van de bounding boxes.

in alle drie de dimensies. We construeren drie lijsten, één voor iedere dimensie. Iedere lijst houdt de waarde van de eindpunten van de intervallen bij die overeenkomen met die dimensie. Door deze lijsten te sorteren, kunnen we bepalen welke intervallen overlappen. In het algemeen zal het sorteren $O(nlogn)$ tijd in beslag nemen. Verdere reductie van de tijdscomplexiteit is mogelijk door de gesorteerde lijsten van het voorgaande frame bij te houden en enkel de waarden van de interval eindpunten te veranderen. Aangezien er relatief kleine bewegingen zijn tussen frames zullen de lijsten al bijna gesorteerd zijn. Insertion sort vergt dan slechts lineaire tijd [4, 43].

1.3.3 Opsplitsing van de ruimte in binaire subruimtes



Figuur 1.10: Binary space partitioning. Vlakken delen de object ruimte recursief op [15].

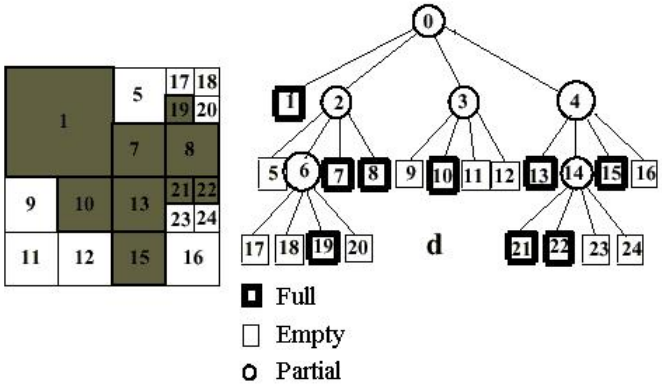
Er bestaat een eigenschap die zegt dat er altijd een vlak te vinden is tussen 2 objecten, op voorwaarde dat deze objecten convexe polyhedra zijn die niet intersecteren. Het aantal intersectie tests kan drastisch gereduceerd worden door de ruimte recursief te partitioneren en deze eigenschap uit te buiten [5].

1.3.4 Hiërarchische methodes

Quadtrees

Een quadtree is een hiërarchische datastructuur. De wortelknoop in de boom is een vierkant dat de volledige omgeving omvat. Vervolgens wordt er recursief gepartitioneerd, totdat ieder vierkant een geschikte uniforme deelverzameling van de input bevat. Quadtrees zijn enkel toepasbaar op 2D omgevingen [47].

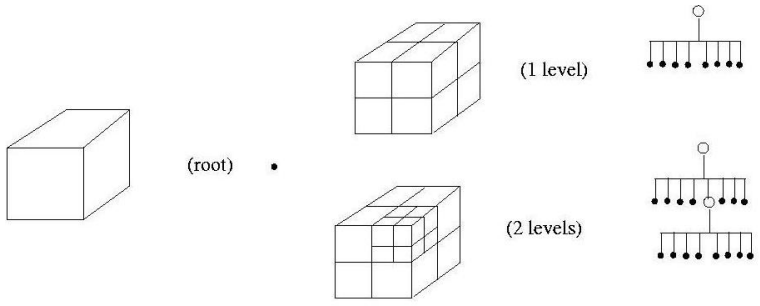
De performantie van collision detection wordt drastisch verbeterd door enkel de objecten in dezelfde quadranten te testen ten opzichte van elkaar.



Figuur 1.11: Quadtree [18].

Octrees

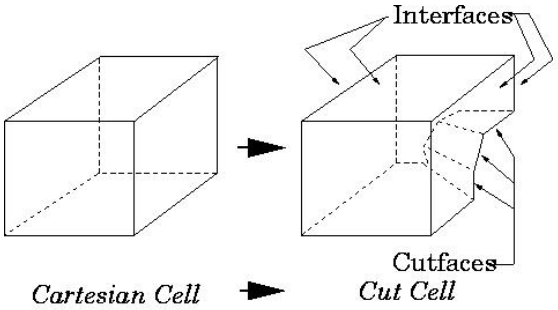
Een octree is een boom om drie dimensies te indexeren. Iedere knoop heeft ofwel 8 ofwel geen kinderen[4] [17]. Behalve de wortel heeft iedere cel in de boom een ouder. Elke cel van een octree komt overeen met een kubus in de fysische ruimte. Ieder kind stelt een octant van zijn ouder voor (figuur 1.12). De berekeningscellen van de grid bevinden zich in de bladeren van de boom. Deze cellen worden geclassificeerd als zijnde cartesische- of cut-cellen (figuur 1.13). Ze bevatten de toestandsvectoren, ruimtelijke afgeleiden van de toestanden, volume, tijdsstap, en alle andere informatie die nodig is [41].



Figuur 1.12: Octree data structuur. De boom wordt twee keer verfijnd. Eerst wordt de wortel opgesplitst in 8 cellen. Elk van deze 8 cellen stelt een octant van het domein van de wortel voor. Op analoge wijze wordt in de volgende stap één van de kinderen van de wortel opgesplitst in 8 cellen [41].

Het zijn de cut-cellen die er voor zorgen dat octrees moeilijk zijn in het gebruik. Ze zijn moeilijk te berkenenen met Cartesische methodes. In het algemeen is een cut-cel het geometrisch verschil tussen de basis cartesische cel en de input geometrie. Het berekenen van algemene intersecties is een zeer moeilijk probleem.

[4] de bladknopen hebben geen kinderen



Figuur 1.13: cut-cel model [41].

Met het oog op collision detection kunnen octrees, net zoals quadtrees, gebruikt worden om vroegtijdige culling te doen.

1.4 Narrow phase collision detection

Narrow phase collision detection analyseert de objectparen die niet verworpen zijn tijdens de broad phase. Deze fase omvat meer verfijnde, maar ook tragere, algoritmes om met zekerheid te bepalen of bepaalde objecten al dan niet intersecteren.

1.4.1 Binary Space Partitioning trees

Een binary space partition tree (BSP) is een datastructuur die een ruimte verdeelt in convexe hulls door middel van hypervlakken[5] [2].

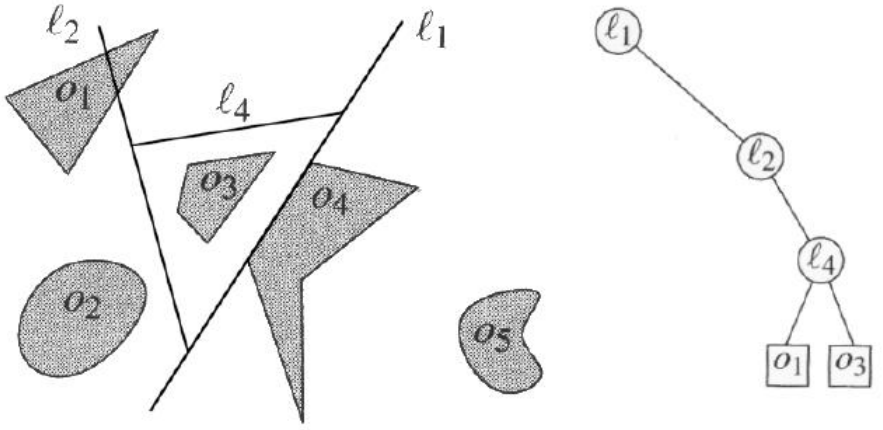
BSP is een techniek die al geruime tijd in de game industrie gebruikt wordt, in het bijzonder in first-person shooters. BSP trees bieden niet alleen efficiënte geometrie culling. World-object collision detection is zonder veel extra werk mogelijk. Het BSP tree algoritme deelt een ruimte op in binaire subruimtes om vroegtijdige geometrie culling te doen (sectie 1.3.3). Er wordt een boom opgebouwd die deze recursieve opsplitsingen voorstelt. De diepte is afhankelijk van de gewenste nauwkeurigheid. Voor de objecten die overblijven na de culling, volstaat het een scheidend vlak te vinden om te bepalen dat ze niet intersecteren. De boom wordt recursief doorlopen en er wordt getest of scheidende vlakken het bounding volume van het object intersecteren. De Cartesische vlakvergelijking maakt het mogelijk om te bepalen aan welke kant van een vlak een punt ligt. Als

$$ax + by + cz + d > 0 \tag{1.3}$$

[5]Een hypervlak kan beschreven worden met een lineaire vergelijking van de vorm:

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b \tag{1.2}$$

In een drie dimensionale ruimte is een hypervlak gewoon een vlak.



Figuur 1.14: Er wordt een boomstructuur bijgehouden van de recursieve opsplitsingen [15].

ligt het punt aan de positieve kant van het vlak. Bij gelijkheid ligt het punt op het vlak en indien

$$ax + by + cz + d < 0 \tag{1.4}$$

weten we dat het punt zich aan de negatieve kant bevindt. Concreet kunnen we bepalen of zo een scheidend vlak bestaat door alle punten van het bounding volume in te vullen in de vlakvergelijking. Indien alle punten aan dezelfde zijde van het vlak liggen is intersectie onmogelijk. Alle polygonen van het object controleren, bevordert de nauwkeurigheid [35].

Een groot nadeel is dat een bijkomende copie van de geometrie van de omgeving nodig is voor iedere objectvorm of grootte. Een BSP tree opstellen voor ieder object is onmogelijk. Dit probleem wordt aangepakt in het dynamic plane shifting BSP traversal algoritme [69].

Een van de eerste commerciele spellen die een BSP datastructuur gebruikten is DOOM [11]. Andere voorbeelden zijn Quake 2 [22] en Unreal [26].

Dynamic plane shifting BSP traversal

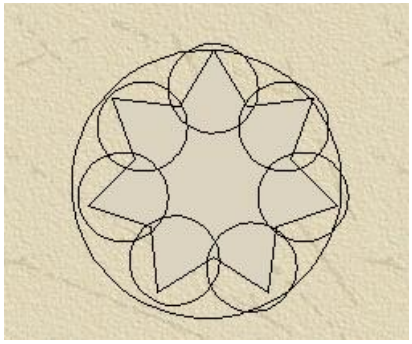
De omgeving wordt voorgesteld met maar één standaard BSP tree. Die boomstructuur wordt geconstrueerd zonder rekening te houden met de vorm van de aanwezige objecten. De vlakvergelijkingen van de knopen worden aangepast bij het doorlopen van de boom. Dit geeft een goede benadering voor collision detection van een willekeurig object met convexe vorm bewegend langs een lineair pad. Het standaard algoritme voor het botsen van een straal met een BSP tree is een recursieve functie die start in de wortel. Als het segment aan één kant van het vlak ligt, dat wordt aangeduid door de knoop, wordt het doorgegeven aan de corresponderende deelboom. Anders kruist het segment het vlak en wordt het opgesplitst. Het eerste deel van het segment wordt gecheckt ten opzichte van de overeenkomstige deelboom. Als het niet botst wordt het tweede deel van het segment gecheckt ten opzichte van de andere deelboom. Als een solid bladknoop

wordt bereikt, geeft het algoritme een botsing. De impact wordt bepaald door het deelsegment van waaruit het blad bereikt wordt [69].

1.4.2 Bounding Volume Hiërarchieën

Bounding volume hiërarchieën worden gebruikt voor collision detection tussen ongestructureerde objecten. Bij het opstellen ervan moet men rekening houden met een aantal aspecten:

- Types van bounding objecten.
- Diepte van de hiërarchie.
- Methode om boomstructuur op te bouwen.



Figuur 1.15: Bounding volume hiërarchie.

De meeste hiërarchieën gebruiken slechts één soort bounding object. De keuze voor een bepaald bounding volume wordt geleid door twee beperkingen:

1. Het bounding volume moet zo strak mogelijk rond het originele model passen. Op die manier zijn het aantal overlap tests en de kost voor het testen van een paar bounding volumes laag.
2. Het testen van twee bounding volumes op overlapping moet zo snel mogelijk gaan, zodat de kost laag blijft.

Rekening houdend met de laaste beperking presteren AABB's en spheres goed. Ze passen helaas niet voldoende strak rond sommige modellen. Aan de andere kant bieden ellipsoiden en OBB's strak passende bounding volumes en dus minder maar duurdere overlap tests. Geen enkele hiërarchische voorstelling garandeert de beste performantie in iedere omstandigheid. In het algemeen werken hiërarchische voorstellingen gebaseerd op spheres en AABB's goed als twee modellen ver van elkaar verwijderd zijn.

Een hiërarchie kan bottom-up of top-down geconstrueerd worden. Bottom-up methodes beginnen met een bounding volume voor iedere polygon en voegen volumes samen tot de boom volledig is. Top-down methodes beginnen met een groep van alle polygonen, en splitsen recursief op tot alle bladknopen niet meer gesplitst kunnen worden [52].

Sphere-Trees

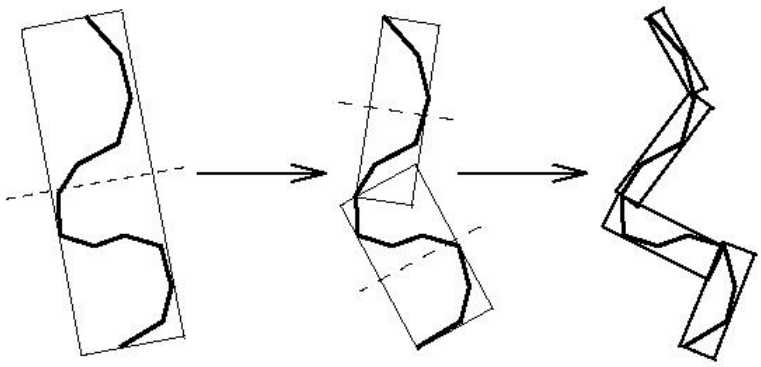
Sphere-trees kunnen de details van objectvormen voorstellen met behulp van een diepe boomstructuur. Palmer en Grimsdale [75] hebben algoritmes ontwikkeld die verzamelingen van spheres gebruiken om een bewegend 3D objectoppervlak te benaderen. Hoewel het in theorie mogelijk is om objecten te benaderen met een verzameling overlappende spheres, is het zeer moeilijk om een object dat gemodelleerd is door een boundary representatie te transleren naar zijn overlappende spheres [86].

OBB trees

De constructie van een OBB tree bestaat uit 2 stappen:

1. Het plaatsen van een tight-fitting OBB rond een verzameling van polygonen.
2. Het groeperen van geneste OBB's in een boomhiërarchie.

Bij de eerste stap moeten de polygonen die uit meer dan 3 zijden bestaan opgesplitst worden in driehoeken. Het algoritme past eerste en tweede orde statistiek toe op vertices van het object om een strak passende OBB te bepalen. De vertices die in het midden van het model liggen kunnen de plaatsing van de bounding box beïnvloeden. Om dit te voorkomen wordt de convexe hull van de vertices van de driehoeken gebruikt. Het sampling probleem is niet verholpen door deze aanpak: een kleine, dichte verzameling van bijna collineaire vertices op de convexe hull kan er voor zorgen dat de bounding box samenvalt met die verzameling. Dit kan opgelost worden door een dichte sampling van het oppervlak van de convexe hull te gebruiken.



Figuur 1.16: Bouwen van een OBB tree: de omhulde polygonen worden recursief gepartitioneerd en de resulterende groepen worden omhuld met een bounding box [52].

Top-down opsplitsen komt neer op het splitsen van de langste as van een bounding box door middel van een vlak orthogonaal op één van zijn assen. De polygonen worden gepartitioneerd afhankelijk van de kant van het vlak waar

hun centrum ligt. Als de langste as niet opgedeeld kan worden, dan wordt de tweede langste as gekozen. Als ook deze niet opgedeeld kan worden dan wordt de kortste as geprobeerd. Indien ook dit niet lukt is de groep van objecten niet op te splitsen (figuur 1.16).

Gegeven een model bestaande uit n driehoeken, is de totale tijd om de boom op te bouwen gelijk aan $O(nlog^2n)$ indien convexe hulls gebruikt worden en $O(nlogn)$ anders. Een box rond een groep van n driehoeken passen en ze splitsen in twee deelgroepen heeft tijdscomplexiteit $O(nlogn)$ met convexe hull en $O(logn)$ zonder. Door het process recursief toe te passen wordt een boom gecreëerd met diepte $O(logn)$ (figuur 1.16). Als de boom geconstrueerd is, kan interferentie detectie efficient uitgevoerd worden met de separating axis methode (sectie 1.3.1) [52, 38].

C-Trees

Een C-Tree is een structuur die een grensvoorstelling voorziet voor objecten met een kinematische hiërarchie, bv. articulated objects. Om collisions op te sporen tussen individuele gearticuleerde delen van een object, dupliceert het algoritme de C-tree. Het object wordt dan behandeld als 2 afzonderlijke objecten. De boom kan een mix van spheres en convexe polyhedra bevatten en moet handmatig geconstrueerd worden, wat het accuraat benaderen van de objecten moeilijk maakt in vele gevallen [38].

1.5 Collision detection algoritmes

1.5.1 Collision detection tussen convexe objecten

Als een groot aantal objecten incrementeel bewegen langs gladde paden, is het belangrijk om de totale tijd nodig voor afstandsberekening te minimaliseren. Als de stappen bij iedere beweging klein zijn, is het mogelijk om voordeel te halen uit geometrische coherentie tussen opeenvolgende afstandsberekeningen. Voor polyhedrale modellen gespecificeerd door een grensvoorstelling vallen de algoritmes in twee categorieën: feature-based en simplex-based [71]. Feature-based algoritmes delen convexe objecten op in vertex, edge en face features en bepalen de dichtst bij elkaar liggende features voor 2 objecten. Simplex-based algoritmes zoeken een simplex die gedefinieerd wordt door de vier vertices van de polyhedron die bepaald wordt door het Minkowski verschil van de betrokkken objecten. Lin-Canny en Gilbert-Johnson-Keerthi zijn respectievelijk voorbeelden van feature-based en simplex-based algoritmes.

Lin-Canny closest features algoritme

Een methode voor snelle collision detection in dynamische simulaties is Ming Lin's en John Canny's closest features algoritme. Het hoofdidee is het opsporen van de features[6] die het dichtst bij elkaar liggen voor twee convexe polyhedra die

[6] vertices, edges, faces

in een ruimte bewegen. Als de objecten concaaf zijn moeten ze eerst opgesplitst worden in convexe delen. Er wordt een criterium gebruikt om te weten of de closest features veranderd zijn en ten opzichte van welke andere features ze veranderd zijn. Er wordt ook een gelinkte lijst van aangrenzende features bijgehouden [64].

Als niet aan een criterium voldaan is, wordt een dichtstbijzijnde aangrenzende feature gekozen. Dit betekent dat het algoritme een gemiddelde uitvoeringstijd $O(n)$ heeft om het eerste closest feature paar te bepalen. Na een kleine, convexe beweging wordt het nieuwe paar closest features in één stap bekomen uit het vorige paar. Indien het aantal aangrenzende features beperkt is kan constante uitvoeringstijd gehaald worden. De afstand tussen twee polyhedra wordt gemakkelijk gevonden als het closest feature paar gekend is. Een botsing treedt op als deze afstand kleiner is dan een bepaalde epsilon waarde. Lin Canny is dus simpel, efficiënt en vooral compleet. De uitvoeringstijd is constant of kan variëren van $O(n)$ tot $O(n^2)$ in het slechtste geval [70].

Gilbert-Johnson-Keerthi distance algoritme gebaseerd op lineaire complexiteit

Het Gilbert-Johnson-Keerthi (GJK) algoritme [50] berekent iteratief de euclidische afstand tussen een paar convexe polytopen in een driedimensionale ruimte. Hiervoor worden het Minkowski verschil en convexe optimalisatie technieken gebruikt [33].

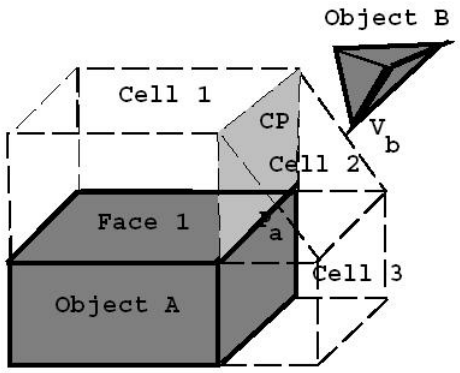
Iedere polytoop wordt gedefinieerd door een lijst van vertices. Voor ieder paar objecten worden de punten die het dichtst bij elkaar liggen berekend, door ze uit te drukken als een convexe combinatie van verschillende vertices. Deze actieve vertices worden gebruikt als input van de volgende uitvoering van GJK. Er wordt dan gecontroleerd of de bewogen vertices actief zijn voor de dichtste punten van de bewogen objecten. Indien dit het geval is, stopt het algoritme onmiddellijk. Voor kleine bewegingen slaagt de test bijna altijd.

Het testen van de bewogen vertices komt neer op evaluaties van inproducten tussen een vaste vector en al de vertices van beide objecten. Aangezien de bewegingen en de inproducten de meeste berekeningstijd vergen, neemt de tijd voor de volledige sequentie van punten lineair toe met het aantal vertices in de twee objecten. Het aantal inproduct evaluaties kan sterk verlaagd worden door voor iedere vertex een lijst van naburige vertices bij te houden. Op die manier wordt de totale tijd onafhankelijk van het aantal vertices. Bovendien blijven de berekeningen correct als de incrementele beweging groter wordt en is numerieke robuustheid gegarandeerd. Datastructuren voor objecten zijn simpel en hebben geen complexe preprocessing nodig. Het gebruik van adjacency informatie in afstandsberekening werd voor het eerst gebruikt door Lin en Canny (sectie 1.5.1) [74].

Voronoi Clip feature based collision detection algoritme

Het Voronoi Clip (V-Clip) algoritme berekent voor ieder object op voorhand de convexe hull. Voor twee objecten die kort bij elkaar liggen, wordt gecheckt

of hun convexe hulls overlappen door incrementeel hun closest feature paren te testen op overlapping.



Figuur 1.17: Voronoi cellen en constraint vlakken.

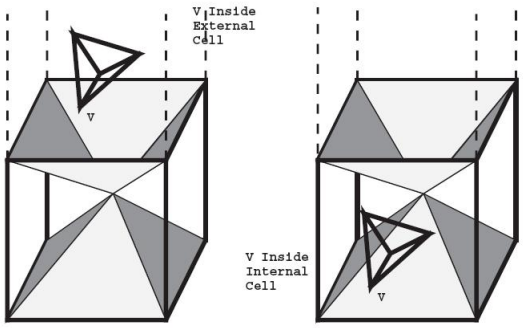
Iedere polytoop heeft een datastructuur met velden voor zijn features en corresponderende Voronoi regio's. Een Voronoi regio van een feature is een verzameling punten die dichter bij die feature ligt dan eender welke andere. Een Voronoi regio van één feature wordt een cel genoemd. Iedere cel bevat een verzameling constraint vlakken die de Voronoi regio afbakenen met pointers naar de aangrenzende cellen. Een punt op een constraint vlak betekent dat het even ver ligt van de twee features die dit vlak gemeenschappelijk hebben in hun Voronoi regio's [62].

Gebruik makend van deze Voronoi regio's, houdt het Lin-Canny algoritme de closest features tussen een paar convexe polyhedra bij. Zoals vermeld in sectie 1.5.1, start Lin-Canny met het zoeken van het closest feature paar en buit het coherentie uit tussen opeenvolgende discrete bewegingen om lineaire uitvoeringstijd te realiseren. Eens de closest features gekend zijn, kunnen de dichtste punten ertussen bepaald worden en dus ook tussen de polyhedra [77].

In tegenstelling tot Lin-Canny kan het V-Clip algoritme penetrerende convexe polytopen opsporen. Behalve de Voronoi regio, die de ruimte buiten de polytoop verdeelt, kunnen we ook Voronoi regio's gebruiken voor interne partitionering. Aangezien de berekening van deze interne Voronoi regio's niet vanzelfsprekend is, werken we met een benadering; de pseudo-interne Voronoi regio.

Zo'n regio wordt gevormd door het centrum van iedere convexe polytoop te berekenen en vervolgens vanuit al zijn zijdes een hypervlak te construeren dat reikt tot aan dat centrum. Alle vlakken van de gegeven polytoop worden dan gebruikt als constraintvlak. Als een kandidaat feature niet voldoet aan de beperking die het vlak oplegt, zal een volgende stap van het algoritme het inwendige van de polygon betreden. Als we op een zeker ogenblik één punt op een feature van één polytoop vinden dat zich in de pseudo interne Voronoi regio bevindt van de andere polytoop is er sprake van een penetratie.

Als alle paren van overlappende features bepaald zijn, kunnen we ze classificeren in 3 categorieën:



Figuur 1.18: Beweging van externe naar interne Voronoi regio's.

- Twee features van originele modellen overlappen. Dit komt overeen met een collision tussen de originele modellen.
- Een feature van een origineel model overlapt met een feature geïntroduceerd door de convexe hull berekening. Verder onderzoek is nodig.
- Twee features geïntroduceerd door de convexe hull berekening. Verder onderzoek is nodig.

In de laatste twee gevallen zal een algoritme gebaseerd op een hiërarchische voorstelling met OBB's gebruikt worden om exacte contact determinatie te realiseren (sectie 1.4.2) [62].

1.5.2 Collision detection tussen polygon soups

Octree en bucket strategie

Het octree en bucket algoritme is toepasbaar op polygon soups die vrij in een ruimte bewegen. De enige vereiste is dat in de x, y en z richting een totale volgorde kan gedefinieerd worden tussen de primitieven. We gebruiken een octree (sectie 1.3.4) waarvan de wortel de AABB van het object voorstelt. Ieder blad van de octree bevat de verzameling van primitieven die in de corresponderende bounding box liggen. Een bucket is een box die overeenkomt met een blad. Bij iedere tijdsstap worden de coördinaten van de bounding box aangepast en iedere primitieve in de juiste bucket geplaatst. Collision detection tussen twee objecten gebeurt door paren van knopen te testen. Indien twee interne knopen[7] overlappen, worden de kinderen van de knoop met het kleinste volume getest ten opzichte van de knoop met het grootste volume. Als maar één van de twee beschouwde knopen een blad is, wordt deze getest ten opzichte van de kinderen van de andere knoop. Als twee bladknopen overlappen, bevinden de twee verzamelingen van primitieven zich in de buckets van die bladeren. In dit geval hangt

[7] knopen die geen bladknopen zijn

de rest van het collision detection proces af van hoe de objecten gemodelleerd zijn.

Om bij iedere tijdsstap de primitieven in de juiste bucket te plaatsen maken we de volgende beschouwing. Een octree met l levels bevat 8^l bladeren die allen overeen komen met een bucket. Als de dimensies van de bounding box S_x, S_y en S_z voorstellen, dan zijn de dimensies van een bucket $\frac{S_x}{2^l}$, $\frac{S_y}{2^l}$ en $\frac{S_z}{2^l}$. De buckets worden geïndexeerd met een triple

$$(n_x, n_y, n_z) : 0 \leq n_x, n_y, n_z < 2^l. \tag{1.5}$$

De index van de correcte bucket voor een primitieve kan berekend worden met de volgende formule:

$$n_x(p) = (p_x - B_x)/S_x. \tag{1.6}$$

In deze formule is p_x de x coördinaat van een primitieve in de ruimte en B_x de minimum x positie van de bounding box. $n_y(p)$ en $n_z(p)$ worden op analoge wijze berekend.

Er zijn twee mogelijke algoritmes om een primitieve aan de correcte bucket toe te kennen:

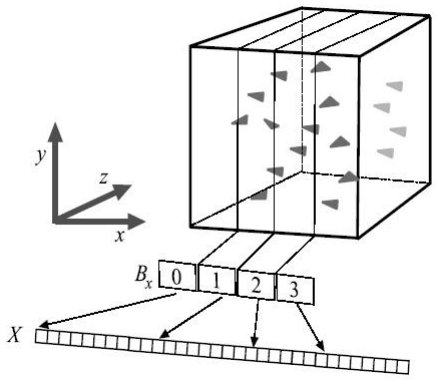
1. Brute force: Bij iedere stap van de simulatie wordt iedere primitieve behandeld en toegekend aan de juiste bucket.

2. Gebruik makend van coherentie tussen opeenvolgende frames: De meeste primitieven blijven in dezelfde bucket tussen twee opeenvolgende stappen.

In het eerste algoritme wordt vergelijking 1.6 uitgevoerd voor iedere primitieve. De kost voor iedere toekenning is 3m aftrekkingen, 3m delingen en 3m floor operaties, indien m het aantal primitieven is. Door frame to frame coherentie uit te buiten kan het aantal toekenningen en dus ook de totale kost aanzienlijk verlaagd worden. We gebruiken drie arrays X, Y en Z, die elk alle primitieven opslaan. Het insertion sort algoritme zorgt ervoor dat deze arrays geordend blijven op de respectievelijke coördinaten. In het geval van inter-frame coherentie heeft insertion sort gemiddelde tijdscomplexiteit $O(n)$.

Bucket indices kunnen verdeeld worden in een aantal intervallen, zodat alle primitieven in het zelfde interval dezelfde index hebben. De positie van de intervallen wordt bepaald door de gehele index van hun eerste element. Dit alles impliceert dat we een array van integers B_x van grootte 2^l kunnen gebruiken. $B_x[k]$ bevat de positie van de eerste primitieve p in X waarvoor $n_x(p) = k$. Bij iedere tijdsstap moet enkel de array B_x aangepast worden. Dit is mogelijk in gemiddeld constante tijd.

Aangezien iedere knoop van de octree statisch verbonden is met een stuk van de ruimte dat de bounding box inneemt, kunnen sommige deelbomen leeg zijn. Om te voorkomen dat lege deelbomen doorlopen worden, propageren we informatie over celbezetting. Iedere knoop in de octree krijgt een geassocieerde variabele. Aan het einde van iedere stap (als de buckets correct zijn geupdate) wordt deze variabele voor alle niet lege buckets gelijk gesteld aan het nummer van de stap. Beginnend in deze buckets werken we naar boven in de boom. Voor een knoop die overeenkomt met een lege deelboom zal de variabele niet aangepast worden.



Figuur 1.19: Voorstelling van de data structuur voor een octree van diepte 2 bekeken vanuit de x richting.

bij het onderzoeken van een bepaalde knoop is het gemakkelijk te achterhalen of zijn deelboom elementen bevat. Checken of de waarde van de variabele gelijk is aan het nummer van de huidige simulatiestap volstaat [48].

IMMPACT: Collision detection tussen massieve polygon soup objecten

De tot nu toe besproken algoritmes bieden real-time performantie voor relatief kleine modellen en zijn zeer geheugenintensief. Ze slaan alle objecten met hun bounding boxes op in het hoofdgeheugen en gebruiken gemiddeld honderden bytes per driehoek. Als de modellen uit een groot aantal polygonen bestaan zijn deze algoritmes te traag.

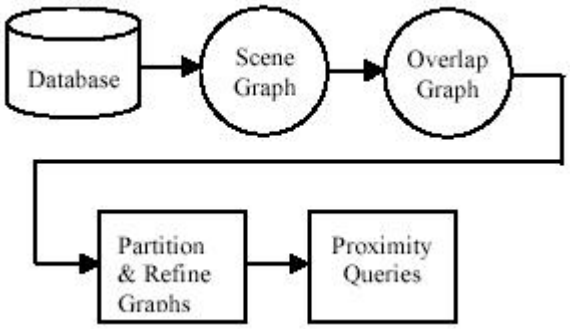
IMMPACT is een collision detection systeem toepasbaar op willekeurige modellen. De gebruiker kan objecten bewegen, invoegen of verwijderen en er wordt een vaste grootte M voorzien voor cachegeheugen. Aangezien proximity queries maar één paar objecten tegelijkertijd beschouwen, moet er op dat moment niet meer in het hoofdgeheugen opgeslagen worden.

Figuur 1.20 geeft een overzicht van de verschillende fases die IMMPACT doorloopt.

Constructie van de overlap graaf

Er wordt een overlap graaf gebouwd om de opeenvolging van proximity queries te realiseren. De knopen komen overeen met objecten in de wereld. Proximity informatie tussen een paar objecten wordt voorgesteld door een boog tussen de overeenkomstige knopen. De bogen worden in een welbepaalde volgorde doorlopen en gelabeld met het resultaat van de gerelateerde proximity query.

Met iedere knoop van de graaf is een gewicht geassocieerd dat overeenkomt met het geheugen ingenomen door een bounding volume hiërarchie voor dat object. Het varieert als een lineaire functie van het aantal polygonen van het object en



Figuur 1.20: Verwerken van geometrische databases voor massieve modellen.

de constante factor wordt beïnvloed door het soort van bounding volume. Het gewicht van een subgraaf wordt berekend door de gewichten van al zijn knopen te sommeren. Het aantal uit te voeren proximity tests wordt geminimaliseerd door enkel de bogen, die knopen met overlappende bounding boxes verbinden, toe te voegen. De bounding boxes kunnen berekend worden met één pass in de database.

Graaf partitionering en verfijning

Een depth-first zoekalgoritme wordt gebruikt om de verbonden componenten en hun gewicht te berekenen. Omdat het gewicht groter kan zijn dan het beschikbare cache geheugen, is het soms nodig om de overlap graaf op te splitsen in een aantal subgrafen. Het gewicht van iedere subgraaf moet minder zijn dan de grootte van de cache en het aantal kruisende bogen tussen de gelokaliseerde subgrafen moet geminimaliseerd worden. Een algoritme dat hier rekening mee houdt bestaat uit de volgende 3 stappen: [90]

1. Objecten worden opgesplitst in twee of meerdere deelobjecten wanneer ze uit een groot aantal polygonen bestaan of bounding volumes hebben die overlappen met een groot aantal andere bounding volumes. Voor elk deelobject wordt een afzonderlijke knoop in de overlap graaf gecreëerd.

2. Er wordt een verzameling knopen met hoge valentie[8] bepaald zodat hun totaal gewicht kleiner is dan de cachegrootte. Door de objecten, voorgesteld door de aangrenzende knopen, een voor een te swappen in de cache, worden alle proximity queries berekend. Deze bogen worden vervolgens verwijderd uit de overlap graaf.

3. De graaf G_0 wordt opgesplitst in een opeenvolging van kleinere grafen G_1,G_2,...,G_m. Deze kleinere grafen worden gebisecteerd. De partitie van de graaf G_m wordt terug geprojecteerd op de oorspronkelijke graaf door tussenliggende partities te doorlopen. De bogen, tesamen met de knopen die ze linken in verschillende partities, vormen een nieuwe graaf. De

[8] Knopen die verbonden zijn met een groot aantal andere knopen.

verbonden componenten van deze graaf worden berekend en de drie deelalgoritmes worden opnieuw uitgevoerd tot de overlap graaf opgesplitst is in gelokaliseerde subgrafen die ieder een gewicht hebben kleiner dan M.

1.5.3 Collision detection tussen vervormbare objecten

Physically-based simulaties gebruiken vaak een groot aantal vervormbare objecten. Algoritmes die in een preprocessing fase voor elk object hulp datastructuren opslaan en updaten, zijn niet toepasbaar op vervormbare objecten. Ook algoritmes die eisen dat objecten convex zijn of bestaan uit een aantal convexe delen kunnen niet gebruikt worden. Vervorming kan immers leiden tot concave objecten.

Algemeen beschouwd wordt de kwaliteit van een methode gebaseerd op bounding volumes beïnvloed door twee factoren (sectie 1.4.2):

- Hoe goed benadert het bounding volume het object?
- Hoe veel berekening is er nodig om overlapping tussen 2 bounding volumes te detecteren?

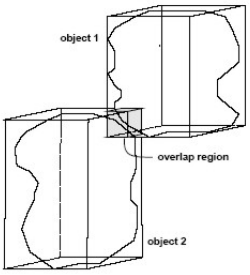
Door de vervormingen kan een bounding volume dat initieel het object het best benadert na een paar iteraties een slechtste benadering opleveren. Dit impliceert dat er geen rekening kan gehouden worden met de eerste factor. Bijgevolg kiezen we steeds voor AABB's met minder dure overlap tests.

NBody processing

In 3D omgevingen bestaande uit convexe of concave objecten die kunnen vervormen door onderlinge interacties, kan collision detection gerealiseerd worden in tijdscomplexiteit kleiner dan $O(N)$. Het algoritme gaat uit van een aantal aannames:

- Alle objecten in de wereld zijn gemodelleerd als polyhedrons.
- De vlakken van een polyhedron zijn triangulaire patches.
- Objecten kunnen concaaf of convex zijn.
- Objecten ondergaan niet vooraf bepaalde bewegingen (rotaties of translaties).
- Objecten kunnen vervormd worden tijdens beweging.
- De snelheid van de objecten is voldoende traag vergeleken met het sampling interval, zodat geen collisions gemist worden.
- De wereld is volledig begrensd door een kubus.

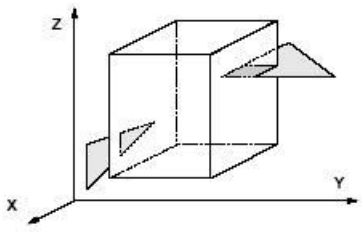
Veronderstel dat er N objecten in een omgeving aanwezig zijn. Ieder object wordt omhuld met een bounding box, die op discrete tijdsinstanties wordt geupdate. Vervolgens worden alle objectparen getest op overlapping van hun bounding boxes (figuur 1.21). Voor ieder paar met overlappende boxes, wordt de overlap regio bepaald en in een lijst bij allebei de objecten toegevoegd. Alle zijden van een object, dat zo een lijst bevat, worden getest op intersectie met



Figuur 1.21: Overlap regio.

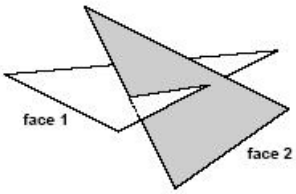
de betreffende overlap regio's (figuur 1.22). Wanneer er effectief een intersectie is, wordt de betrokken zijde in een lijst van zijden voor dat object geplaatst.

Als er een lijst van zijden voor meer dan één object is, wordt er een face octree gebouwd voor de overblijvende zijden. De wortel is de wereldkubus en de face octree wordt gebouwd tot op een door de gebruiker gespecificeerde resolutie.



Figuur 1.22: Testen of zijden intersecteren met overlap regio's.

Voor ieder paar zijden die tot verschillende objecten behoren en dezelfde face octree voxel intersecteren wordt bepaald of ze elkaar intersecteren in de 3D ruimte (figuur 1.23). Zowel de intersectie van zijden met overlap regio's als de



Figuur 1.23: Intersectie test tussen paren van zijdes.

face octree fases testen herhaaldelijk op intersectie van een zijde met een AABB. Dit kan door de AABB te clippen in 2 dimensies, dus ten opzichte van 4 van de vlakken van de box. Voor de overblijvende dimensie moet er enkel gecontroleerd worden of de overblijvende vertices van de geclipte triangulaire patch allemaal

groter zijn dan het maximum of kleiner dan het minimum. Indien één van beide waar is, intersecteert de zijde met de bounding box. Een snelle overlap test tussen de bounding box van de patch en de AABB, en testen of een van de drie vertices van de patch in de AABB ligt, zijn mogelijk optimalisaties die voor de clipping uitgevoerd kunnen worden [83].

1.6 Collision scheduling methodes

Collision scheduling methodes zijn alle technieken die bepalen op welk ogenblik collision tests, in de vorm van proximity queries, uitgevoerd worden. Proximity queries geven informatie over de relatieve plaatsing van twee objecten. Het woord **proximity** duidt aan dat al deze queries van toepassing zijn op objecten die zich in elkaars omgeving bevinden. Voorbeelden van proximity queries zijn:

- Intersectie detectie.
- Tolerantie verificatie: Bepaalt of de afstand tussen 2 objecten kleiner is dan een gegeven tolerantie.
- Exacte afstandsberekening.
- Benaderende afstandsberekening: Bepaalt de minimum afstand tot op een (relatieve en/of absolute) foutmarge tussen een paar objecten.
- Translatie afstand nodig om twee penetrerende objecten van elkaar te scheiden.
- Estimated time of arrival (ETA): Het tijdstip waarop de volgende botsing tussen 2 objecten plaatsvindt wordt berekend aan de hand van hun plaatsing en bewegingen.
- Bepalen van contact normalen, closest features, tussen niet overlappende objecten [59].

Intersectie detectie en exacte afstandsberekening zijn speciale gevallen van tolerantie verificatie en benaderende afstandsberekening respectievelijk [9] [45]. Iedere applicatie heeft nood aan specifieke proximity queries. Intersectie detectie is belangrijk in physically-based modeling en animatie systemen die alle contacten moeten kennen om de collision response te kunnen berekenen. Afstandsinformatie is nuttig om interactiekrachten te berekenen in robot motion planning. De ETA berekening biedt controle over de tijdsstap in een simulatie [63].

1.6.1 Collision detection op vaste tijdsstippen

Een mogelijke collision scheduling methode is op regelmatige tijdstippen op collisions testen. Dit is een simpele techniek met relatief voorspelbare performantie. Het probleem is dat collisions gemist kunnen worden en interpenetratie

[9] Intersectie detectie is tolerantie verificatie met tolerantie 0 en exacte afstandsberekening is benaderende afstandsberekening met foutmarges gelijk aan 0.

van solids mogelijk is. De penetratieafstand of een heuristiek is nodig om een scheidende impuls te berekenen. Verschillende tegengestelde impulsen kunnen leiden tot objecten die blijven interpenetreren [59].

1.6.2 Collision detection met adaptieve tijdsstap

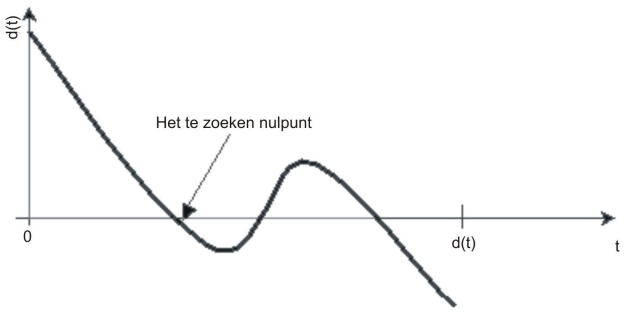
Een andere manier van collision scheduling is het aanpassen van de tijdsstap tussen opeenvolgende collision testen door middel van een van de volgende methodes:

1. Bisectie: Indien een botsing gedetecteerd wordt, halveer het voorgaande tijdsinterval totdat de modellen gescheiden zijn.
2. Voorspelling: Bereken een ondergrens voor de tijd van impact voor een paar modellen en voer de volgende collision test uit op dat tijdstip.

De bisectie methode voorkomt penetraties door de tijd te verlagen tot de tijd van de vroegste botsing alvorens een nieuwe stap te nemen. Er kunnen nog altijd collisions gemist worden. Om dit op te lossen is er de voorspellingsmethode.

De voorspellingsmethode vertraagt collision tests voor modellen die ver van elkaar liggen. Ze voorkomt penetraties en het missen van collisions. Voor modellen die kort bij elkaar liggen, wordt de frequentie van testen verhoogt. Er is bovendien nood aan vaste of begrensde bewegingspaden: gebruikersinvoer of collisions kunnen alle tijden van impact, verbonden met een object, ongeldig maken [59].

1.6.3 Exacte tijd van botsingen

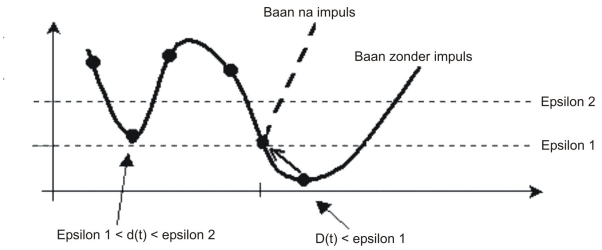


Figuur 1.24: Root finding.

Een andere mogelijkheid om het missen van collisions en de mogelijk volgende penetraties te vermijden is de berekening van de exacte tijd van botsing. Figuur 1.24 beeldt een grafiek af die de minimum afstand tussen twee objecten in functie van de tijd uitdrukt. Er heeft duidelijk een collision plaatsgevonden; d(t) is positief voor t = 0 en negatief voor t = dt. Om het nulpunt van deze functie te

vinden kan bijvoorbeeld de Newton methode gebruikt worden. Een probleem stelt zich wanneer collisions kort achter elkaar gebeuren. Het systeem wordt dan overbelast met berekeningen. Als d(0) $<$ 0; d.w.z. de objecten penetreren; dan stopt de root finding nooit. Dit betekent dat collisions opgelost moeten worden voor de penetratie kan plaatsvinden.

Om de root finding te versnellen kunnen collision windows gebruikt worden. Dit zijn twee waarden, epsilon 1 en epsilon 2, met epsilon 2 groter dan epsilon 1 en beide groter dan 0. Als d(t) van de objecten kleiner is dan epsilon 2 en groter dan epsilon 1 kan collision detection op het tijdstip t opgelost worden, maar als d(t) kleiner is dan epsilon 1 moet een t bepaald worden waarvoor d(t) groter is dan epsilon 1 en kleiner is dan de starttijd t = 0. Maar dit is veel simpeler dan root finding iedere keer dat een collision gebeurt.



Figuur 1.25: Gebruik van windows.

1.7 Collision detection libraries

De bestaande collision detection packages kunnen opgesplitst worden in twee categorieën. Als de objecten in de scene convex zijn of opgesplitst kunnen worden in convexe delen dan zijn de convex based packages aangewezen. Indien de objecten een meer algemene vorm hebben, worden polygon soup based packages gebruikt. In deze sectie wordt een overzicht gegeven van de convex based packages. Een overzicht van de polygon soup based packages is terug te vinden in sectie 1.7.2.

1.7.1 Convex based collision detection packages

SWIFT

SWIFT is een library voor collision detection, afstandsberekening en contact determinatie van drie-dimensionale polygonale objecten die rigide bewegingen ondergaan [46].

SWIFT gebruikt een verbeterd Lin-Canny algoritme (sectie 1.5.1) om de afstand tussen 2 rigide modellen te minimaliseren. De applicatie moet zelf zorgen dat de objecten opgesplitst worden in convexe delen die onderling niet op collisions

getest worden. SWIFT kan bounding volume hiërarchieën voor objecten automatisch genereren en het type van bounding box bepalen. Geometrie culling gebeurt via het sweep and prune algoritme (sectie 1.3.2).

SWIFT kan performantie onafhankelijk maken van bewegingscoherentie en is snel en robuust. Objecten kunnen aan een scene toegevoegd worden met arrays of bestanden en kunnen efficiënt gecopieerd en verwijderd worden. SWIFT voert zelf triangulaties uit, zodat de arrays of bestanden geen specificaties van de driehoeken moeten bevatten.

De gebruiker kan **intersectie detectie** doen op 2 niveau's. Afhankelijk van zijn keuze wordt gerapporteerd dat er een intersectie is in de scene, of een lijst van alle intersecterende objecten gegeven. Het is niet mogelijk om een maat voor de penetratiediepte te op te vragen. Het berekenen van **benaderende afstand** tussen 2 objecten is vooral zinvol als er hiërarchieën gebruikt worden. Zonder hiërarchieën is de benaderende afstand gelijk aan de **exacte afstand**. Benaderende afstand tussen objecten kan berekend worden indien hun bounding boxes overlappen. Tenslotte voorziet SWIFT ook **contact determinatie** [46].

V-CLIP

Het V-Clip collision detection package voorziet een C++ implementatie van het low-level V-Clip collision detection algoritme (sectie 1.5.1) [27]. De input objecten hoeven niet aaneengesloten te zijn[10] en kunnen rigide bewegingen ondergaan. Net zoals SWIFT is V-Clip gebaseerd op het Lin-Canny closest features algoritme (sectie 1.5.1). Het is geschikt voor dynamische omgevingen waarin objecten kunnen bewegen in een sequentie van kleine, discrete stappen.

Behalve afstandsberekening biedt V-Clip nog enkele extra's zoals het bepalen van penetratiepunten en geschatte penetratieafstand voor overlappende modellen[11] en het toevoegen van objecten door middel van arrays of files. Bovendien is het algoritme erg robuust. In tegenstelling tot SWIFT gebruikt V-Clip de sweep and prune techniek niet. De gebruiker moet geen numerieke toleranties bepalen, zodat cycling problemen niet voorkomen. In typische applicaties berekent V-Clip de dichtste punten tussen objecten in bijna constante tijd [71].

I-COLLIDE

I-COLLIDE is een interactieve en exacte collision detection library voor grote omgevingen bestaande uit convexe polyhedra [12]. I-COLLIDE is de originele Lin-Canny implementatie (sectie 1.5.1) en gebruikt het sweep and prune algoritme (sectie 1.3.2) en benut optimaal tijdscoherentie en speciale eigenschappen van de convexe polytopen. De plaats van elk model wordt opgeslagen en een lijst

[10] De convexe opsplitsing moet uit een matig aantal delen bestaan en de hiërarchie mag maar een paar levels diep zijn

[11] minimum afstand nodig om een verzameling te transleren zodoende dat ze disjunct is met een andere verzameling:

$$Pen(A, B) = minimum \parallel v \parallel \tag{1.7}$$

met

$$min_{a \in A} min_{b \in B} \mid a - b + v \mid > 0 \tag{1.8}$$

[65]

van potentiële contactparen wordt aangepast wanneer de modellen van plaats veranderen. I-COLLIDE is niet erg robuust en objecten toevoegen is alleen maar mogelijk met bestanden. Bovendien worden alleen maar rigide bewegingen ondersteund. De gebruiker kan op voorhand opgeven tussen welke objectparen hij de afstand wil bepalen [43].

SOLID

SOLID is een collision detection library voor drie-dimensionale objecten die rigide bewegingen maken en kunnen vervormen. SOLID is ontworpen voor het gebruik in interactieve 3D graphics applicaties, en is bijzonder geschikt voor collision detection van objecten en werelden beschreven in VRML [24].

SOLID gebruikt 2 algoritmes. Het bouwt eerst een bounding volume hiërarchie bestaande uit AABB's. Voor rigide modellen kan SOLID overlap tests tussen AABB'S versnellen, zodat ze de efficiënte van OBB's evenaren. De kleinere hoeveelheid geheugen die AABB's innemen rechtvaardigt de keuze. Voor vervormbare modellen zijn AABB's zelfs sneller te bouwen en aan te passen. Vervolgens maakt de SOLID library gebruik van een uitgebreide versie van het GJK algoritme (sectie 1.5.1) om de afstand te berekenen tussen 2 convexe polytopen. Voor de berekening van de convexe hull gebruikt SOLID de Qhull library [21].

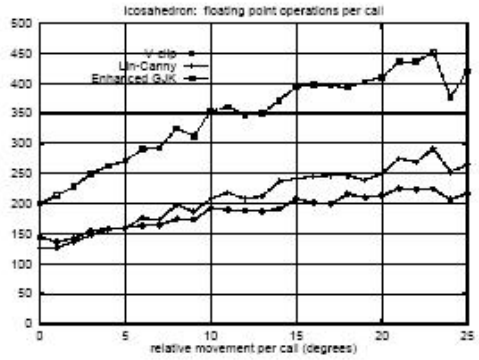
DEEP

DEEP (Dual-space Expansion for Estimating Penetration Depth) is een package om de penetratiediepte tussen convexe polytopen te schatten. Er wordt incrementeel een lokale optimale oplossing gezocht aan de hand van de oppervlakken van de Minkowski sommen. Het algoritme werkt goed als er een hoge bewegingscoherentie in de omgeving is en is meestal in staat om de optimale oplossing te berekenen [56, 8].

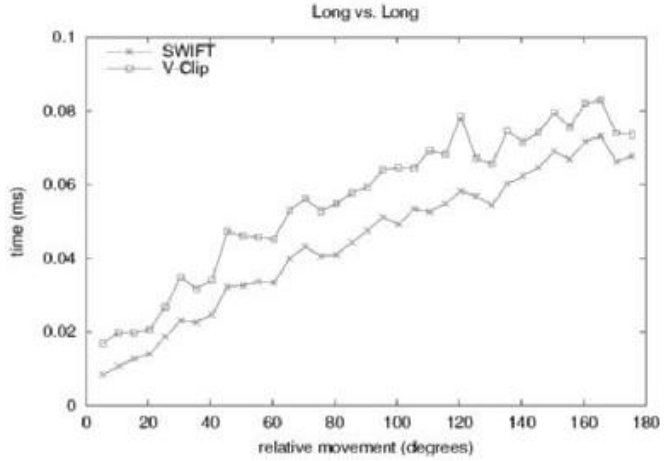
Vergelijking

Uit een experiment blijkt dat het aantal floating point operaties, dat nodig is om de closest features tussen een paar convexe polyhedra bij te houden, vergelijkbaar is voor V-Clip en Lin-Canny op verschillende niveau's van coherentie. Een uitgebreide versie van het GJK algoritme heeft tussen de 50% en 100 % meer operaties nodig. In figuur 1.26 stelt de horizontale as de coherentieparameter voor. Hoe hoger deze waarde, hoe groter de bewegingen zijn van de modellen tussen aanroepen van het algoritme. V-Clip is een stuk robuuster dan Lin-Canny. Tests wezen uit dat Lin-Canny voor kritische configuraties in 4.5% van de gevallen problemen gaf. V-Clip, zonder toleranties, vertoonde geen opspoorbare problemen. Het uitgebreide GJK algoritme schoot in 0.1% van de gevallen tekort; bij een te hoge floating point tolerantie [71].

Alle algoritmes die gebaseerd zijn op het GJK algoritme zijn bijgevolg trager dan degene die op het Lin-Canny algoritme gebaseerd zijn. Verder is er ook aangetoond dat SWIFT sneller is dan V-Clip voor afstandsminimalisaties tussen verschillende vormen van objecten. Figuur 1.27 geeft dit weer door de tijd



Figuur 1.26: Totaal aantal floating-point operaties voor icosahedra [71].



Figuur 1.27: Low level vergelijking tussen de snelheid van SWIFT en V-Clip bij collision detection tussen 2 lange objecten [46].

die nodig is om de afstand tussen twee willekeurige ellipsoïden te berekenen te vergelijken. Er werden geen specifieke optimalisaties gebruikt voor objecten en er werd geen bounding box hiërarchie of caching schema bovenop het low-level algoritme gebouwd. Ook op hoger niveau biedt SWIFT betere performatie dankzij de ondersteuning van het sweep and prune algoritme voor bounding boxes en de mogelijkheid om performantie onafhankelijk te maken van bewegingscoherentie [46].

1.7.2 Polygon soup based collision detection packages

Polygon soup packages werken op lijsten van driehoeken. Typisch voor dit soort packages is dat ze een hiërarchie van bounding volumes creëren met als bladknopen de driehoeken zelf. De hiërarchieën zijn samengesteld uit spheres, AABB's, OBB's, SSV's (Sphere-Swept Volumes), of k-DOP's (Discrete Oriented Polytopes). Deze packages zijn ontworpen om rigide bewegingen te ondersteunen, maar sommigen kunnen aangepast worden om te werken met vervormbare objecten.

RAPID

RAPID is een robuuste en accurate library om interferentie tussen polygonen te detecteren in grote omgevingen bestaande uit ongestructureerde modellen. De library is vrij te gebruiken voor niet-commerciële doeleinden [23]. RAPID bouwt top-down een hiërarchie van OBB's (sectie 1.4.2) op uit een model. Vervolgens worden OBB paren getest op overlappingen. OBB's passen in het algemeen beter rond objecten dan AABB's of spheres. De hiermee verbonden daling in overlap tests resulteert in een minder diepe hiërarchie en betere performantie [52].

V-COLLIDE

V-COLLIDE is een collision detection library voor grote omgevingen. Het is ontworpen om te werken met een groot aantal statische of bewegende polygonale objecten [28]. V-COLLIDE berekent de afstand tussen 2 convexe modellen, gebruik makend van een uitgebreide versie van de distance routine van Gilbert, Johnson en Keerthi (sectie 1.5.1). Objecten kunnen dynamisch toegevoegd of verwijderd worden tussen tijdstappen. Broad phase collision detection gebeurt met het sweep and prune algoritme zoals beschreven in sectie 1.3.2. De narrow phase wordt gerealiseerd met het pair processing algoritme van RAPID [28].

PQP

PQP is een library om drie types van proximity queries uit te voeren op een paar van geometrische modellen samengesteld uit driehoeken: collision detection, afstandsberekening en tolerantie verificatie [19]. PQP gebruikt een top-down strategie om een hiërarchie van Rectangle Swept Spheres (sectie 1.3.1) te creëren uit een polygon soup model. Eens de RSS boom beschikbaar is, worden afstandsberekeningen tussen RSS's uitgevoerd met een gespecialiseerd algoritme om efficiëntie en robuustheid te bevorderen [60].

SWIFT++

SWIFT++ is een collision detection package dat in staat is om intersecties te detecteren, tolerantie verificatie uit te voeren, benaderende en exacte afstand te berekenen, of contacten tussen objectparen te bepalen in een scene samengesteld uit polyhedrale modellen. Het ondersteunt twee bestandsformaten voor het inlezen van informatie over modellen. Het TRI formaat is geschikt voor triangulaire

modellen, terwijl het POLY formaat willekeurige polyhedrale modellen ondersteunt. Beide formaten zijn echter beperkt tot convexe objecten. SWIFT++ kan ook werken met algemene polyhedra, op voorwaarde dat ze eerst met het bijhorende preprocessing programma worden opgesplitst in een hiërarchie van convexe delen. Net zoals de meeste andere packages, ondersteunt SWIFT++ vervorming van objecten niet [45].

1.8 Collision Detection in bekende games

1.8.1 MDK2

In MDK2 wordt er een cylinder gebruikt om character-to-environment collision detection te implementeren. Deze cylinder staat altijd recht omhoog. Door de bilaterale symmetrie van de cylinder is de gebruiker altijd in staat om te draaien. Als de speler een botsing heeft met een andere speler of met een object in de wereld moet hij naar een andere plaats gestuurd worden. Het pad van de speler wordt omhoog gehaald als het botst met een of ander object. Als het omhoog gehaalde pad vrij is, wordt de speler langs dit pad bewogen en daarna wordt het pad terug naar beneden gehaald.

MDK2 is een spel met verschillende characters met een verschillende grootte. Verschillende copieën van de geometrie van de omgeving in het geheugen opslaan is niet mogelijk. Behalve characters, creëert het spel ook vele kleine artefacten om speciale effecten, zoals explosies, te construeren. Deze kleine artefacten hebben ook nood aan snelle collision detection. Een nieuwe BSP tree construeren voor iedere partikelgrootte is niet toelaatbaar. Dit probleem wordt opgelost door het **dynamic plane shifting BSP traversal** algoritme, zoals beschreven in sectie 1.4.1 [69, 13].

1.8.2 Neverwinter Nights

Neverwinter Nights is geen actie/physics geörienteerd spel waarin characters springen en schieten in een 3D omgeving. Het is een Role Playing Game gebaseerd op de regels gebruikt in Dungeons and Dragons. Dit betekent dat de noden voor collision detection in dit spel compleet anders zijn dan in MDK2. Collision detection wordt bijna volledig beperkt tot lijnsegment tests, nodig voor de selecties van gebruikers, het bepalen van de **path-searching** node onder het character en de zichtlijn tussen 2 characters. De game engine moet ook in staat zijn om met polygon soups te werken. Men gebruikt daarom bounding volume hiërarchieën om collision detection te realiseren [69, 16]. Op dit ogenblik is men geneigd om AABB's te gebruiken omwille van de beperktere geheugen benodigdheden. De performantie is bovendien even goed aangezien veel polygonen in dit spel axis-aligned zijn (sectie 1.3.1).

1.9 Besluit

Collision detection vormt vaak de bottleneck in real-time applicaties. Het is van groot belang om bij iedere collision test de irrelevante geometrieën uit te sluiten. Collision detection algoritmes worden daarom meestal opgedeeld in een broad en een narrow phase. In de broad phase worden de objecten die niet in aanmerking komen om met elkaar te botsen genegeerd. De narrow phase test of de overblijvende objecten botsen.

Om de berekeningen nog sneller te doen, worden de tests dikwijls uitgevoerd op de bounding volumes van de betrokken objecten. Afhankelijk van de vorm van de objecten in de wereld en de snelheid van intersectie tests worden er bounding boxes, bounding spheres, k-DOPs of Swepth Sphere Volumes gebruikt. Intersectie tests zijn veel sneller voor axis-aligned bounding boxes dan voor oriented bounding boxes. Hier tegenover staat dat oriented bounding boxes de objecten nauwkeuriger omhullen. In het geval objecten vervormd kunnen worden door onderlinge interacties zijn axis-aligned bounding boxes steeds het meest performant. Intersectie testen tussen 2 spheres komt neer op het berekenen van de afstand tussen hun middelpunten. k-DOPs zijn tight-fitting bounding boxes. AABB's zijn k-DOPs met k = 6. Swepth Spheres kunnen opgesplitst worden in verschillende categorieën die elk geschikt zijn voor specifieke objectvormen en hun eigen kost voor overlap tests hebben. Rectangle Swepth spheres hebben dezelfde kwadratische convergentiesnelheid als OBB's. Line Swepth Spheres convergeren iets sneller, maar nog altijd niet lineair zoals spheres.

Broad phase collision detection kan gebeuren door bounding boxes op een grid te projecteren. Alle bounding boxes die geen zelfde cellen bezetten worden verworpen. Vele algoritmes gebruiken een sweep-and-prune algoritme, dat door een dimensiereductie 3D bounding boxes sorteert, om te bepalen welke bounding boxes overlappen. Vervolgens wordt er top-down of bottom-up een hiërarchie van bounding boxes gebouwd om de nauwkeurigheid te vergroten. Een andere mogelijkheid is om een ruimte recursief op te delen in binaire subruimtes. De opdelingen kunnen in een BSP boomstructuur opgeslagen worden. Deze structuur laat toe om precisiere collision detection uit te voeren. Technieken die nauwkeurige tests doen op de objecten die overblijven na de broad phase behoren tot de narrow phase van het collision detection proces.

Er moet ook rekening gehouden worden met de tijdsstappen tussen opeenvolgende aanroepen van een collision detection algoritme. Indien de tijdsstap groot is kunnen snel bewegende objecten collisions veroorzaken die niet opgespoord worden. De tijdsstap kleiner kiezen houdt in dat het collision detection algoritme veelvuldig aangeroepen wordt, wat niet mogelijk is in real-time applicaties.

Er zijn een aantal collision detection packages gratis beschikbaar voor niet-commercieel gebruik. Indien de input objecten convex zijn of opsplitsbaar in een verzameling convexe, is SWIFT het aangewezen package dankzij zijn robuustheid en snelheid. Er bestaat een uitbreiding SWIFT++ die ook met polygon soups als input modellen werkt. IMMPACT is specifiek ontworpen voor massieve polygon soup modellen. Geen enkel van de packages ondersteunt vervorming van de objecten.

In het volgende hoofdstuk wordt besproken wat een collaboratieve virtuele omgeving is en welke eigenschappen ze heeft. Verder worden er een aantal netwerk

protocollen, netwerk architecturen en methodes tegen netwerk congestie besproken. Tot slot volgt een bespreking over de mogelijke problemen die gepaard gaan met de integratie van collision detection in een CVE.

Hoofdstuk 2

Collaborative Virtual Environments

2.1 Definitie

Een VE-systeem is een computersysteem dat een user-centered driedimensionale, interactieve virtuele omgeving genereert. Er wordt gebruik gemaakt van positie-tracking en real-time updating van visuele, auditieve,... displays als feedback op de acties van de gebruiker. De gebruikers krijgen het gevoel echt in de omgeving aanwezig te zijn [85].

Real-time rendering is noodzakelijk in VR systemen om op ieder ogenblik snelle feedback te geven op de input van de gebruiker. Twee soorten VR systemen worden onderscheiden: VR software systemen en VR hardware systemen. De software systemen bieden een interface naar de onderliggende hardware om hardware-afhankelijke details te verbergen voor de programmeurs. VR hardware systemen kunnen sterk variëren afhankelijk van beschikbare resources en specifieke noden. Desktop VR hardware systemen voorzien simpele small-scale VR mogelijkheden gebruik makend van een monitor die stereoscopische graphics mogelijk maakt en een tracking eenheid met zes vrijheidsgraden. Daar tegenover staan immersieve hardware systemen die verschillende, naast elkaar gepositioneerde projectieschermen combineren en bovendien gebruik maken van een tracking systeem met zes vrijheidsgraden en een computer voor high-speed rendering en high-resolution stereo graphics [53].

Een Shared Virtual Environment (SVE) is een VE-systeem dat door verschillende personen gelijktijdig gebruikt kan worden. Een gebruiker kan interageren met andere gebruikers. Een collaboratief VE-systeem (CVE) is een SVE met als doel samen te werken aan een taak in dezelfde virtuele ruimte [31].

Technisch gezien is het computernetwerk het belangrijkste aspect waarmee men rekening moet houden. Communicatie tussen gebruikers zou duidelijk en intuïtief moeten zijn, wat dikwijls betekent dat deelnemers kunnen praten met elkaar. Coöperatief gebruik van informatie moet ondersteund worden. Visueel moet de gedeelde wereld het zelfde zijn voor alle gebruikers, of hij moet ge-

lijkaardig genoeg zijn zodat geen enkele gebruiker cruciale omgevingselementen mist [53].

Coöperatief gebruik van informatie impliceert de nood aan efficiënte netwerk communicatie. Een definitie van efficiënte netwerk communicatie ligt niet voor de hand aangezien efficiëntie voor iedere applicatie anders is. SVE's trachten door middel van communicatie voor iedere gebruiker maximale consistentie te bieden tussen gedeelde objecten en de netwerk traffiek te minimaliseren. Deze vorm van balanceren is gekend als de Consistency-Throughput Tradeoff:

"Het is niet mogelijk om een dynamische gedeelde toestand regelmatig aan te passen en bovendien te garanderen dat alle hosts gelijktijdig toegang hebben tot identieke versies van die toestand."[53]

De beschikbare bandbreedte van het netwerk moet dus verdeeld worden tussen boodschappen die gedeelde objecten updaten en boodschappen die een consistente view van die objecten garanderen. Beelden moeten bovendien snel genoeg geupdate (hoge refresh rate) worden om bruikbaar te zijn in virtuele realiteit [53].

2.2 Eigenschappen van VE systemen

VE systemen hebben verschillende dimensies waarbinnen ontwerpers keuzes kunnen maken: [61]

- Immersiviteit: De graad waarmee de virtuele omgeving de gebruiker omringt. Desktop systemen, head mounted displays, etc. zijn enkele mogelijke waarden binnen deze dimensie.
- Presence: De graad waarmee de gebruiker zich een deel van de virtuele omgeving voelt. Deze graad is afhankelijk van vele andere aspecten van het VE systeem en is kritiek voor het slagen of falen van de taak van de gebruiker. De graad van presence is een onmiddellijk gevolg van de graad van immersiviteit.
- Architectuur: Is het VE systeem gedistribueerd of niet?
- Persistentie: Bestaat de wereld nog als de gebruiker zich afmeldt?

Het principe van oorzaak en gevolg uit de reële wereld moet in een simulatie ook gegarandeerd worden. Concreet betekent dit dat de events in de gesimuleerde wereld in dezelfde volgorde moeten uitgevoerd worden als de acties in de echte wereld die ze voorstellen [49].

2.3 Toepassingen met CVE's

CVE's kunnen voor uiteenlopende doeleinden gebruikt worden: entertainment, medische applicaties en andere soorten van opleidingen. In de medische sector kunnen studenten geneeskunde en dokters opgeleid worden met CVE's. Het

Figuur 2.1: DISCOVER Tele-Surgery (Trachea) Training[49].

is niet erg om fouten te maken op een virtuele patiënt en uitzonderlijke situaties kunnen gemakkelijk gemodelleerd en bestudeerd worden. Er is een grote vermindering in medische kosten en tekorten (bv. van proefdieren) zijn er niet.

De opleidingen zijn toegankelijk voor iedereen aangezien de afstand niet langer een drempel vormt. Bovendien is het onderhoud van de trainingsfaciliteiten een stuk goedkoper. Virtuele objecten aanmaken kost weinig en kunnen zo dikwijls als nodig gedupliceerd worden. Schade aan een object is niet langer een financieel verlies. De NASA maakt ook al geruime tijd gebruik van virtuele omgevingen om zijn astronauten op te leiden.

Het Human Interface Technology Laboratory van de universiteit van Washington heeft een collaboratieve Web browser ontwikkeld waarin gebruikers simultaan elkaar en virtuele webpagina's kunnen zien. De deelnemers ervaren de anderen hun gezichtsuitdrukkingen, lichaamstaal en bewegingen door een see-through display te gebruiken [49].

2.4 Transport Protocollen

Computers die verbonden zijn met een netwerk kunnen verschillende onafhankelijke processen hosten die gelijktijdig uitgevoerd kunnen worden. Bovendien kan ieder proces over het netwerk communiceren met processen op andere computers. Het netwerk moet duidelijk meer zijn dan een kabel tussen de computers.

Efficiënte communicatie heeft nood aan protocollen. Ze zorgen er voor dat processen elkaar kunnen identificeren en maken zenden en ontvangen van gegevens in 2 richtingen mogelijk. De protocollen maken een tradeoff tussen 2 eigenschappen: reliability en communicatiesnelheid[1]. De gebruiker bepaalt welk van de eigenschappen het meest noodzakelijk is en kiest een passend protocol [51].

Het internet heeft twee hoofdprotocollen in de transportlaag, het connectionless User Datagram Protocol (UDP) en het connection-oriented Transmission

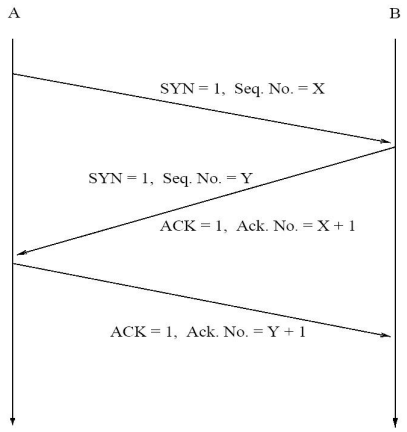
[1]Zowel optimale snelheid als reliability is niet realiseerbaar aangezien reliability overhead veroorzaakt, wat de snelheid aanzienlijk vertraagt.

Control Protocol (TCP) [87].

2.4.1 TCP

TCP is een van de hoofdprotocollen in TCP/IP netwerken. Het is gebouwd boven op het IP protocol, dat zich enkel met paketten bezig houdt. Alvorens twee hosts gegevens kunnen uitwisselen moet er een connectie gemaakt worden met het three-way handschake protocol (figuur 2.2). TCP voorziet een reliable end-to-end gegevensstroom over een onbetrouwbaar netwerk (bv. Internet). Alle gegevens worden verstuurd zonder rekening te houden met errors in de IP laag, gebruik makend van sequentienummers, acknowledgements en retransmissies. Gegevens worden opnieuw gestuurd indien ze hun eindbestemming niet bereiken door fouten of corrupte data. De sequentienummers zorgen ervoor dat de data hun volgorde bewaren over het netwerk. Verder is er ook flow control ingebouwd om te verzekeren dat hosts niet meer data toegestuurd krijgen dan ze kunnen verwerken en is de verbinding full-duplex. Er kan dus tegelijkertijd gezonden en ontvangen worden, zodat de beschikbare bandbreedte optimaal gebruikt wordt.

Een internetwerk verschilt van een eenvoudig netwerk aangezien de delen heel erg verschillende topologieën, bandbreedtes, delays, pakket grootte,... kunnen hebben. TCP is ontworpen voor gebruik op internetwerken en is robuust in het geval van fouten. Een nadeel van TCP en alle andere connection-oriented protocollen is dat er veel duplicaten moeten verzonden worden als er veel data pakketten verloren gaan of niet tijdig aankomen bij de bestemmeling. Dit kan leiden tot congestie van het netwerk en dus grote vertragingen in de communicatie snelheid, zodat de applicatie zelf hier onder zal lijden. [87, 1]



Figuur 2.2: TCP connectie opzetten tussen 2 hosts met het three-way handshake protocol.

Congestion control

Aangezien packet loss te wijten aan transmissie fouten nog maar zelden voorkomt, zijn de meeste time-outs te wijten aan congestie. Alle TCP algoritmes

gaan er van uit dat dit de oorzaak is van de time-outs. TCP tracht congestie te voorkomen in de eerste plaats.

Hoewel de netwerklaag ook probeert om congestion te behandelen, wordt het grote werk gedaan door TCP. De echte oplossing om congestion te voorkomen is het verlagen van de data snelheid. In theorie kan congestie bestreden worden door een principe uit de fysica toe te passen: de wet van behoud van pakketten. Het idee is om geen nieuw pakket in het netwerk te injecteren totdat een oud afgeleverd is. Om dit te realiseren gebruikt TCP twee windows; één voor de capaciteit van de ontvanger (gebaseerd op de buffergrootte van de ontvanger) en één voor de capaciteit van het netwerk. De oplossing die voor Internet gebruikt wordt, bestaat er in om deze twee capaciteiten niet te overschrijden.

Het aantal bytes dat gezonden kan worden is gelijk aan het minimum van de twee windows. Initieel is de grootte van het congestion window gelijk aan de grootte van het maximum segment dat de zender verstuurt over de connectie. Vanaf dan probeert de zender altijd het dubbel aantal segmenten te sturen. Indien hij acknowledgements ontvangt voor al deze segmenten wordt de grootte van het congestion window aangepast [87].

2.4.2 UDP

UDP, zoals gedefinieerd in [RFC 768], is een transportprotocol. Bij UDP wordt het zenden van een IP datagram niet vooraf gegaan door een handshaking tussen zendende en ontvangende entities in de transportlaag. Juist om deze reden wordt UDP als connectionless bestempeld. Behalve muliplexen en demultiplexen en een lichte vorm van foutopsporing voegt UDP niets toe aan IP. UDP voorziet dus geen flow control, error control, of retransmissie bij ontvangst van een slecht segment.

UDP ontvangt boodschappen van applicatie processen. Vervolgens verbindt het de nummervelden van bron- en doelpoorten uit de 8-byte header voor het multiplexen/demultiplexen en levert het een best-effort inspanning om het segment af te leveren bij de ontvangende host. De UDP header bevat verder ook nog een veld dat de lengte van het volledige pakket specificeert en optioneel een veld om de UDP checksum te berekenen. UDP garandeert dus niet dat boodschappen aankomen, maar de waarschijnlijkheid dat een bepaalde boodschap aankomt is vrij groot.

Als een UDP pakket aankomt, wordt zijn payload doorgegeven aan het process dat verbonden is met de doelpoort. Het binding process voor UDP is identiek aan dat van TCP. Het grootste voordeel dat UDP heeft ten opzichte van zuivere IP is de toevoeging van de bron- en doelpoort.

UDP is zeer geschikt in client-server applicaties waar de client een korte request stuurt naar de server en een korte reply terug verwacht. Als het request of de reply verloren gaat veroorzaakt de client een time-out en kan hij opnieuw proberen. Op deze manier zijn er minder boodschappen nodig (één in iedere richting) dan met een connection-oriented protocol waarvoor een initiele setup vereist is [78, 87].

2.4.3 Welk protocol gebruiken?

Als real-time communicatie niet van doorslaggevend belang is, kan de programmeur een communicatie protocol kiezen dat de beste garanties biedt op het gebied van betrouwbaarheid. Veel genetwerkte applicaties, zoals interactieve games, zijn onbruikbaar als ze niet snel genoeg kunnen uitgevoerd worden en gebruiken daarom minder betrouwbare protocollen. Indien een unreliable protocol gebruikt wordt, moet de communicatie zodanig gestructureerd worden dat het verloren gaan van boodschappen geen ernstige invloed heeft op het verloop van de applicatie. Dikwijls moet het netwerk een combinatie van protocollen gebruiken om de verschillende data types te communiceren. De data types kunnen geclassificeerd worden in de volgende categorieën [81]:

- Real-time audio en video data
- Data die de scene of objecten met hun attributen beschrijven: Bijvoorbeeld de beschrijving van een avatar.
- Computer Supported Collaborative Work (CSCW) data: 2D collaboratieve data zoals whiteboards
- Controle data: Voor taken zoals consistentie controle.
- Update boodschappen.

Het is gebruikelijk om real-time audio en video data te verzenden met het Real-Time Transport Protocol (RTP). Login informatie en object beschrijvende gegevens worden meestal via TCP gecommuniceerd. CSCW gegevens en controle boodschappen maken gebruik van Reliable Multicast of een client-server architecture.

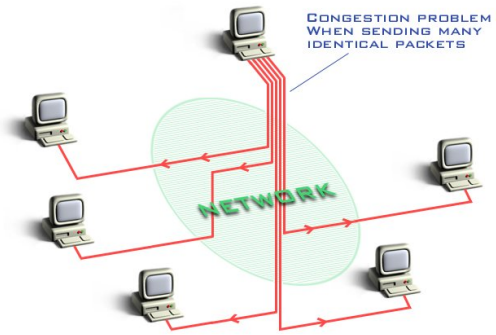
De communicatie van update messages tussen entiteiten is afhankelijk van de implementatie van het systeem: [81]

- Peer-to-peer communicatie: Een entity stuurt update boodschappen naar alle andere hosts. Dit is zeer inefficiënt indien er veel entities en hosts zijn (sectie 2.5.2).
- Client-server: Clients sturen update boodschappen naar een server die ze forward naar de andere clients. De server kan een bottleneck vormen. Bovendien is er een bijkomende delay aangezien de server nu een tussenliggend station is (sectie 2.5.1).
- broadcast: Een entiteit broadcast zijn update boodschap naar iedereen op het subnetwerk. In de praktijk functioneert broadcasting slecht. Iedere host ontvangt en verwerkt het broadcast packet, zelfs als hij niet deelneemt aan de simulatie (sectie 2.5.4).
- UDP multicast: Er is geen garantie dat alle deelnemers de meest recente toestand van een gedeeld object hebben. Door keep-alive boodschappen te sturen om de 5 à 10 seconden hebben de deelnemers altijd een laatste toestand van een object. Maar dit veroorzaakt latency en een toename in bandbreedte gebruik aangezien een object deze boodschappen moet sturen zelfs als het niet verandert.

- Reliable multicast: Reliable Multicast protocollen gebruiken multicast en garanderen bovendien dat alle boodschappen afgeleverd worden. In bepaalde situaties is de toename in bandbreedte en bijkomende delay niet aanvaardbaar (bijvoorbeeld bij teleconfering).

2.5 Gedistribueerde VE's

2.5.1 Client-server

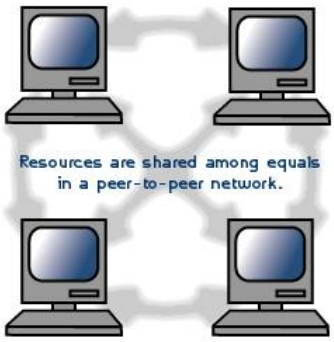


Figuur 2.3: Client-server model [67].

Client-server systemen (zie figuur 2.3) maken het mogelijk om functies en applicaties te centralizeren in één of meerdere dedicated servers. De servers stellen resources beschikbaar aan de clients en voorzien veiligheid. Behalve de centrale opslag en controle van gegevens, garandeert een client-server architectuur flexibiliteit, interoperabiliteit en toegankelijkheid [29].

2.5.2 Peer-to-peer

Een peer-to-peer (P2P) netwerk heeft geen gecentraliseerde server. De peer nodes functioneren zowel als client en server voor de andere nodes in het netwerk. Peer nodes kunnen verschillen in lokale configuratie, processing snelheid, netwerk bandbreedte en opslagruimte [30]. In applicaties die een P2P netwerk gebruiken runt iedere gebruiker zijn eigen simulatie van de applicatie. Elke peer verzamelt al zijn inputs en distribueert ze naar al de andere peers in het netwerk. De ontvangen inputs van de andere peers worden op iedere tijdsstap identiek toegepast op de simulaties die in parallel runnen op iedere peer. Men zegt dat iedere peer in lockstep met de anderen moet runnen om identieke simulatie op iedere peer te garanderen [36]. P2P netwerken zijn in de eerste plaats ontworpen voor kleine tot middelmatige lokale netwerken. AppleShare en Windows for Workgroups zijn voorbeelden van programma's die gebruikt worden als P2P netwerk operating systems.



Figuur 2.4: P2P netwerk [29].

2.5.3 Voor- en nadelen van P2P en client-server topologie

P2P netwerken hebben veel minder initiele kosten dan client-server netwerken. De extra kosten voor een dedicated server moeten bij een P2P netwerk uiteraard niet gemaakt worden. Hier tegenover staat dat er geen centrale opslagplaats voor bestanden en applicaties is. P2P netwerken voorzien bovendien niet de veiligheid die client-server netwerken voorzien [29].

Voor gaming via het internet is P2P de gemakkelijkst bruikbare maar niet de meest ideale topologie. Als boodschappen verloren gaan, blokkeert de applicatie totdat iedere peer alle boodschappen ontvangen heeft (lockstep beperking). Multiplayer games zoals Doom maakten gebruik van een P2P netwerk en konden enkel aan een aanvaardbare snelheid gespeeld worden op lokale LAN's. Het feit dat iedere peer input messages moet ontvangen en zenden naar iedere andere peer zorgt voor een explosie van input messages. Het groot aantal connecties en boodschappen vormen in dit geval de bottleneck.

Indien een client-server model gebruikt wordt, is de server de enige computer die input messages ontvangt van de clients. De server is dus alleen verantwoordelijk voor de simulatie van de wereld. Als input messages verloren gaan op weg van een client naar de server, wordt enkel de gameplay van die client beïnvloed. Positie update messages die verloren gaan in de omgekeerde richting, tasten ook enkel de gameplay van de client in kwestie aan. Bij de client-server topologie stelt zich het lockstep probleem duidelijk niet en bovendien moet enkel de server over voldoende bandbreedte beschikken om te communiceren met iedere client [36].

Het grote nadeel van client-server netwerken is dat ze niet schaleerbaar zijn. De centrale computer vormt een bottleneck in deze configuratie als het aantal client processen groot wordt. Bovendien zijn de clients afhankelijk van de server. Bij een crash van de server stoppen alle client operaties [51, 3].

2.5.4 Broadcasting

In een simulatie met N deelnemers moet iedere deelnemer een wijziging van zijn status aan al de anderen melden. Als iedere deelnemer zijn positie update voor

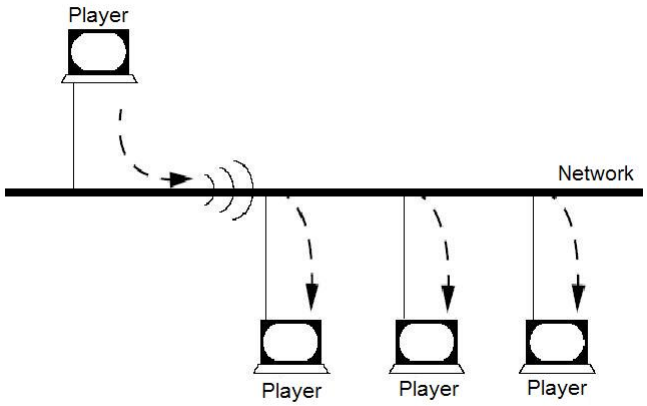
broadcast type	*IP adres*	*ontvangen door*
flooded	*255.255.255.255*	*alle hosts*
directed	*131.108.255.255*	*alle hosts op netwerk 131.108*
directed	*131.108.4.255*	*alle hosts op subnet 4 van netwerk 131.108*

Figuur 2.5: *Voorbeelden van directed en flooded broadcasts*

het renderen, moet hij per rendering frame N(N-1) boodschappen zenden. Een alternatief is de gewijzigigde database te broadcasten als één boodschap [51].

Broadcast netwerken hebben slechts één communicatiekanaal dat gedeeld wordt door alle machines op het netwerk. Broadcast pakketten worden herkend door alle hosts. Routers moeten geconfigureerd worden om onnodige verspreiding van pakketten tegen te gaan. Om dit te realiseren houdt ieder pakket een adres veld bij in zijn header. Twee soorten broadcasts worden onderscheiden aan de hand van dit adres. Een directed broadcast is een pakket gezonden naar een specifiek netwerk of serie van netwerken. Flooded broadcasts naar ieder netwerk gestuurd worden. Bij ontvangst controleert een machine het adresveld van een pakket. Indien het voor hem bestemd is, verwerkt hij het pakket. Een packet bestemd voor een andere machine wordt genegeerd [87].

Broadcasting vereenvoudigt de implementatie; een deelnemer die inlogt in een applicatie moet geen connecties maken met de andere gebruikersprocessen. Het nieuwe proces moet enkel weten op welk broadcast kanaal het moet luisteren. Eens het process het kanaalnummer kent, kan het luisteren voor broadcasts van andere processen en zijn eigen updates broadcasten [51].



Figuur 2.6: Broadcast communicatie [51].

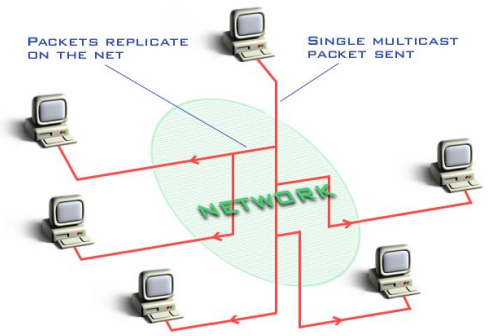
Broadcasts kunnen gebeuren op de datalink laag of op de netwerk laag. Datalink broadcasts worden gezonden naar alle hosts die verbonden zijn met een bepaald fysisch netwerk. Broadcasts op het niveau van de netwerklaag worden gezonden naar alle hosts verbonden met een bepaald logisch netwerk.

Problemen van broadcasting

Hoewel huidige IP implementaties een broadcast adres bestaande uit allemaal enen gebruiken, gebruikten de eerste IP implementaties een broadcast adres bestaande uit allemaal nullen. Het gevolg is dat veel oudere implementaties broadcast adressen die bestaan uit allemaal enen niet herkennen en dus niet correct op de broadcast antwoorden. Andere gedateerde implementaties forwarden broadcasts bestaande uit allemaal enen. Dit heeft een serieuze netwerk overload tot gevolg, beter gekend als broadcast storm [25].

2.5.5 Multicasting

Figuur 2.3 illustreert het probleem van een client-server architectuur. Indien een client informatie naar de server zendt die de toestand van de applicatie wijzigt, moet de server alle andere clients op de hoogte stellen. Het aantal bytes dat de server zendt, is gelijk aan het aantal bytes aan gegevens vermenigvuldigd met het aantal geconnecteerde clients. Als bij iedere beweging van de muis of toetsaanslag gezonden wordt, is de netwerktraffiek hoog. Bijgevolg is deze configuratie enkel bruikbaar voor een beperkt aantal deelnemers [67].



Figuur 2.7: Multicasting [67].

Multicasting is een deelverzameling van broadcasting waarbij groepen van processen gemaakt kunnen worden, zodat enkel deze groepen de boodschap ontvangen. Het vermindert dus het aantal te zenden gegevens door de taak van pakketreplicatie te verschuiven van de server naar de eigenlijke netwerk infrastructuur [87]. Aan multicasting zijn ook nadelen verbonden:

- De openheid van multicast groepen kan er voor zorgen dat pakketten makkelijker te sniffen zijn. Normaal gezien kunnen UDP pakketten enkel onderschept worden tussen hun bron en bestemming, maar bij multicasting is dit mogelijk eender waar in het netwerk. Aanmelden bij de juiste groep is voldoende.
- sommige Internet Service Providers en netwerken ondersteunen multicasting nog niet.

2.6 Netwerk Delays

2.6.1 Latency problemen

Een eenheid om latency tussen twee nodes in een netwerk uit te drukken is de round trip time (RTT). Dit is de tijd die een packet nodig heeft om een node te bereiken en terug te keren tot bij de zender. Uiteraard is dit een benadering en is er geen garantie dat de heen- en terugweg evenveel tijd in beslag nemen. De RTT is ook onderhevig aan frequente, onvoorspelbare veranderingen, ook wel jitter genoemd. Simpele technieken om delays te bepalen zijn bijgevolg meestal niet zo effectief. Voor genetwerkte games zijn latency, jitter en packet loss bepalende factoren. Hoe groter ze zijn hoe meer problemen de gebruikers zullen ervaren. Het doel van een netwerk is om iedere gebruiker een nauwkeurige toestand van de omgeving, die door een server gesimuleerd wordt, te presenteren zonder een natuurlijke navigatie in die omgeving in de weg te staan.

Als de delay in een netwerk zeer klein is, is er geen merkbaar verschil met echte interacties. In sommige gevallen is de delay niet te verwaarlozen en stellen zich de volgende problemen [39]:

1. De wereld die de gebruiker waarneemt is al verouderd met 1/2 RTT. Dit is precies de tijd die nodig was om de informatie te ontvangen van de server.
2. Als de gebruiker input verstrekt, verstrijkt een bijkomende tijd van 1/2 RTT alvorens de server deze input heeft ontvangen en toegepast. De geschiktheid van de input van de gebruiker, die al gebaseerd is op een verouderd systeem, wordt nog verder gereduceerd.
3. De gebruiker moet 1 volledige RTT wachten op feedback op zijn input. Dit maakt een succesvolle interactie in veel gevallen onmogelijk.

Indien een spel bestaat uit verschillende interagerende spelers zijn er twee bijkomende problemen. Ten eerste moet er consistentie zijn in de toestand van het spel voor alle spelers. Maar de connecties met de centrale game server zijn onderhevig aan de zonet besproken beperkingen, waardoor iedere speler verschillende latencies zal ervaren. Bovendien kan iedere individuele client lokaal delays proberen te compenseren, bv. d.m.v. dead reckoning (sectie 2.7). Dit draagt er toe bij dat de spelers een heel verschillende view van de game state hebben en verwarringen ontstaan.

Een tweede aandachtspunt is eerlijke interactie tussen de spelers. De server kan de input van de clients toepassen in de volgorde van ontvangst. Maar dit zou de spelers met connecties met hogere latency een bijkomend nadeel geven. Anderzijds kan de server trachten om de latency van de clients te schatten en te verrekenen bij het toepassen van de inputs. Mogelijk kunnen hierdoor wel nieuwe inconsistenties gecreëerd worden [39].

2.6.2 Oplossen van latency problemen

In het algemeen biedt dead reckoning een effectieve oplossing voor de problemen die in puntje 1 en 2 van voorgaande sectie beschreven staan. Maar bij het derde

probleem is de gebruiker verplicht om één volledige RTT te wachten alvorens hij feedback krijgt over zijn input van de server.

Een andere techniek, short circuiting, die gebaseerd is op extrapolatie kan gebruikt worden. Aan de hand van een ping stream maakt de client een schatting van de delay tussen zichzelf en de server. De input van de gebruiker wordt niet enkel naar de game server gestuurd maar lokaal geëxtrapoleerd rekening houdend met die ping. Op die manier krijgt de gebruiker onmiddelijk feedback op het effect dat zijn input heeft wanneer hij de server bereikt [39].

2.7 Dead reckoning

Algemeen

Dead reckoning is een methode die het aantal te zenden boodschappen reduceert. Deze techniek wordt gebruikt in populaire simulatiemechanismes zoals DIS (the Distributed Interactive Simulation protocol) [73], SIMNET [76] en NPSNET [68]. Het is een vorm van herhaalde berekening.

Dead reckoning is gebaseerd op het feit dat iedere client bewegingen van zijn eigen avatar en die van de andere clients simuleert. Het niveau van betrouwbaarheid is hierbij meestal nogal laag. Op voorhand wordt een verzameling van algoritmes overeengekomen, die elke deelnemer gebruikt om het gedrag van entiteiten te interpoleren. Voorbeelden hiervan zijn voorspelling en convergentie. Er wordt ook bepaald in welke mate deze benaderingen mogen afwijken van de werkelijke beweging alvorens er een correctie wordt doorgevoerd.

Convergentie maakt het mogelijk objecten vloeiend te laten bewegen bij opeenvolgende discrete updates. Een tijdstoename tussen de updates kan wenselijk zijn om de hoeveelheid netwerkverkeer voor een object te verminderen. Het toekomstig gedrag van een object wordt voorspeld aan de hand van zijn recente gedrag door afgeleide polynomialen te gebruiken. Deze polynomialen worden geconstrueerd gebruik makend van eerste, tweede en derde afgeleiden van de huidige positie van een object. Voorspellingen voor ieder object afzonderlijk zijn accurater dan algemene voorspelling. [51]

DIS, dat onstond als een omgeving voor tank simulatoren tesamen in een netwerk te gebruiken, is een van de eerste computer systemen die gebruik maakt van dead reckoning. DIS is een strikte P2P netwerk architectuur die alle data stuurt naar alle simulatoren. Deze kunnen dan uitmaken of ze deze data nodig hebben of niet. Hoe dead reckoning in DIS precies in zijn werk gaat is beschreven in de volgende sectie.

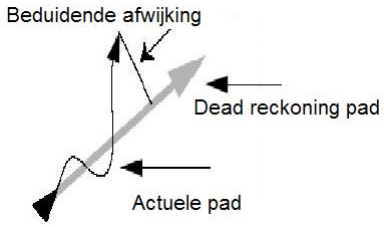
Hoe gebeurt dead reckoning in DIS

Bij DIS stuurt de computer die een entiteit of voertuig creëert een Protocol Data Unit (PDU) naar alle andere computers in het netwerk. Die beschrijft de toestand van een entiteit aan de hand van zijn unieke informatie. Concreet is dat de huidige kinematische toestand: positie, snelheid, versnelling, oriëntatie en andere informatie zoals het damage level. Verder bevat de PDU ook een

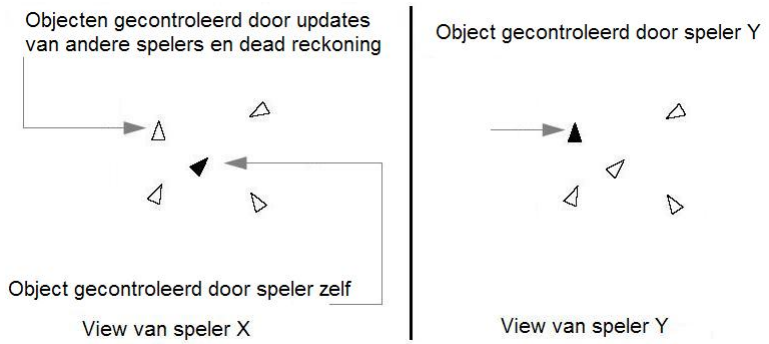
identifier die alle andere computers in het netwerk vertelt welk dead reckoning algoritme gebruikt moet worden voor de entiteit in kwestie.

De PDU die gezonden wordt bij de creatie van een entiteit zorgt ervoor dat ze waarneembaar is op de andere computers in het netwerk. Die maken op dat ogenblik een lokale kopie en bewegen de entiteit door een overeengekomen dead reckoning algoritme toe te passen. Vanaf dan worden PDU's gezonden om de vijf seconden als de entiteit voorspelbaar blijft bewegen of op het ogenblik dat ze afwijkt van het pad. Zonder extrapolatie algoritme, zou de entiteit enkel bewegen als er bijkomende PDU's met geupdate parameters gezonden worden. Dit resulteert in schokkerige bewegingen van de entiteit. Door de tijd tussen het zenden van PDU's te verlagen worden de bewegingen vloeiender, maar bij een groot aantal entiteiten is de bandbreedte van het netwerk snel uitgeput.

Om mogelijk afwijkingen van het voorspelde pad op te sporen, houdt de eigenaar van een entiteit bij wanneer hij de laatste keer een PDU heeft verzonden. Hij voert zelf het dead reckoning algoritme, gebaseerd op die PDU, uit. Hij heeft dus behalve de werkelijke waarde ook een kopie van wat alle andere clients in het netwerk zien. Door de dead reckoning waarden te vergelijken met de werkelijke toestand, stelt hij vast of een overeengekomen afwijking overschreden worden. Indien dit het geval is, stuurt hij een nieuwe PDU naar alle clients, die hun kopie van de entiteit updaten. Vanaf dan kan dead reckoning opnieuw beginnen met het nieuwe data punt. [32]



Figuur 2.8: Vergelijking tussen het werkelijke pad van de entiteit en het dead reckoning pad [51].



Figuur 2.9: Iedere speler ziet zijn eigen entiteit en de entiteiten gecontroleerd door updates van andere spelers en dead reckoning [51].

2.8 Collision Detection in CVE's

Bij het integreren van collision detection in een netwerk moeten een aantal keuzes gemaakt worden. Allereerst moet afgewogen worden welke netwerk architectuur het meest geschikt is. Om in een P2P netwerk te voorkomen dat er collisions gemist worden is een goede synchronisatie nodig. Dit impliceert dat er tussen alle peers in het netwerk boodschappen gecommuniceerd moeten worden. Dit is net de bottleneck van dergelijke netwerken (sectie 2.5.2). Bij een client-server architectuur kan men de centraliteit van de server uitbuiten. Wetend dat de server steeds de meest recente informatie over iedere client bevat, kan er met zekerheid voorkomen worden dat collisions gemist worden (zie sectie 2.8.1). Bij een client-server netwerk kan een groot aantal gebruikers er voor zorgen dat de applicatie niet behoorlijk functioneert (sectie 2.5.1).

Behalve de keuze van een geschikte netwerk topologie is het ook van belang te overwegen welk protocol gebruikt wordt om update informatie te versturen over het netwerk. Reliable protocollen, zoals TCP, bieden het voordeel dat packets altijd worden afgeleverd bij de ontvanger. In het geval van een slechte communicatielijn betekent dit dat er dikwijls packets opnieuw gestuurd zullen moeten. Connection-less protocollen zoals UDP zullen daarentegen nooit gegevens opnieuw sturen, maar kunnen niet voorkomen dat packets verloren gaan. Met het oog op collision detection zijn verouderde posities ontstaan door retransmits nutteloos (sectie 2.4.3). UDP lijkt dus de beste oplossing, maar er moeten maatregelen genomen worden om te voorkomen dat het verlies van update messages foutieve collision detection veroorzaakt.

In alle realtime object simulatoren zijn bewegingen discreet. De illusie van gladde bewegingen komt voort uit het feit dat de afstanden waarmee de objecten vooruitspringen zeer klein zijn. Maar hoe sneller een object beweegt, hoe verder het moet springen tijdens een vaste tijdstap. Als een object bij een update ver genoeg beweegt, is het mogelijk dat het door een ander object heenvliegt. Het missen van collisions kan opgelost worden door de stapgrootte van het object in kwestie te verkleinen of door het gebied, dat het object vanuit een positie met één stap kan bereiken, te begrensen met een speedbox. In dit geval zijn de collision tests wel minder accuraat [34].

2.8.1 Collision detection in een client-server netwerk

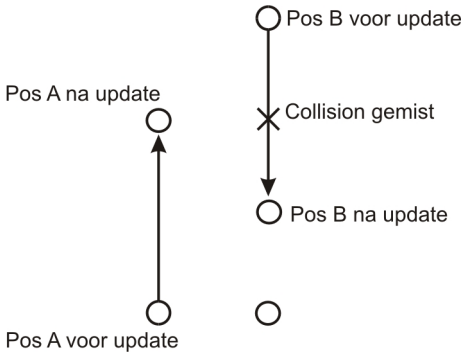
Een oplossing waarbij geen collisions gemist zullen worden is om de clients iedere update aan te laten vragen aan de centrale server. De server voert het collision detection algoritme uit en stuurt de nieuwe positie naar al de clients. Indien het move request packet niet of foutief bij de server aankomt, zal de positie van de client sowieso niet geupdate worden en zal dus naar geen enkele client een nieuwe positie gestuurd worden. Op die manier blijft de synchronisatie behouden.

Indien een update request wel correct bij de server aankomt maar het bevestigingspacket voor alle of enkele van de clients verloren gaat, dan zullen deze clients (mogelijk inclusief de client die gerequest heeft) de avatar van de requestende client nog altijd op de zelfde plaats waarnemen. Maar aangezien de server altijd de meest recente positie van alle avatars heeft en enkel hij de collision detection regelt zullen er nooit collisions gemist worden. Dit impliceert

dat zowel de eigen avatar van de client als die van de andere clients sprongen kunnen maken.

2.8.2 Collision detection in een P2P netwerk

De kracht van een client-server architectuur is net het feit dat de server altijd over de meest recente gegevens van iedere client beschikt. Zoals reeds gezegd bestaat er in een P2P netwerk geen centrale repository. Onderstel de situatie zoals geschetst in figuur 2.10.



Figuur 2.10: Collisions worden gemist indien update messages verloren gaan.

Peer A update zijn positie zoals te zien is in het linkerdeel van figuur 2.10. Vervolgens stuurt hij een update van zijn positie door naar peer B. Maar de update message gaat verloren en B update zijn positie aangezien hij niets weet van de nieuwe positie van A. Zoals te zien in het rechterdeel van de figuur mist peer B dus de collision. Op dat moment stuurt hij zijn nieuwe positie naar A en de avatar van peer B verschijnt plots achter die van peer A.

Zoals beschreven wordt in sectie 2.7 is dead reckoning een techniek om het aantal gecommuniceerde data packets in een netwerk te reduceren. Iedere client voorspelt zijn eigen positie. Zolang zijn positie evolueert langs een overeengekomen pad en de voorspelde positie dus niet te veel afwijkt van de echte positie stuurt hij geen update naar de andere clients. Bovendien voorspelt hij ook de posities van de andere clients. Hij past de positie van een client aan indien hij een update krijgt van die client.

Indien in bovenstaande situatie dead reckoning gebruikt wordt, is de kans op packet loss kleiner (maar niet onbestaande) doordat er minder packets gecommuniceerd worden. Dead reckoning functioneert bovendien niet goed als de clients veel bruuske bewegingen maken, aangezien er dan veel updates verzonden moeten worden.

2.9 Besluit

Collaborative Virtual Environments hebben verschillende eigenschappen waar ontwerpers rekening mee moeten houden: presence, immersiviteit, architectuur, persistentie. Een hoge graad van presence wordt bekomen door iedere speler een natuurlijk viewpoint op de omgeving en een unieke, of tenminste herkenbare, voorstelling te geven. De immersiviteit neemt toe als de input van de gebruikers real-time geupdate wordt. Net zoals persistentie is de keuze van een netwerk architectuur afhankelijk van de aard van de applicatie. Een beste keuze voor iedere situatie bestaat niet. Het gecentraliseerde model kent problemen bij een groot aantal deelnemers. Het gedistribueerde model heeft eveneens een schaleerprobleem als er veel connecties en boodschappen zijn. De taak van de ontwerper is de resources, nodig om het netwerk op te bouwen, te minimaliseren zonder dat de bovenstaande eigenschappen in het gedrang komen.

Een bekende techniek om het aantal boodschappen en/of connecties in het netwerk te reduceren is dead reckoning. De voordelen van dead reckoning in games en simulaties zijn tweevoudig. Het verbergt de latencies die inherent zijn in een netwerk en vermindert het data verkeer over het netwerk aangezien simulaties enkel informatie doorsturen als het nodig is.

In genetwerkte simulaties is collision detection vaak onmisbaar. Om correcte collision detection te kunnen garanderen is een goede synchronisatie van groot belang. In een client-server netwerk heeft men het voordeel dat de server steeds over de meest recente informatie van iedere client beschikt. Indien iedere client via UDP een update van zijn positie aanvraagt aan de server die na het uitvoeren van collision detection al dan niet deze moves bevestigt, zijn we dus verzekerd van een redelijke goede synchronisatie.

Multicasting als een deel van broadcasting vermindert het aantal te communiceren boodschappen door de netwerkinfrastructuur pakketreplicatie te laten afhandelen. De performantie van broadcasting laat in de praktijk dikwijls te wensen over en multicasting is nog niet voldoende ondersteund.

Het is mogelijk om verschillende netwerk protocollen te combineren binnen de netwerk architectuur. In de praktijk blijkt dit dikwijls noodzakelijk aangezien niet alle gecommuniceerde data gegarandeerd bij de bestemmingshost moeten aankomen. Updates van posities of move requests reliable versturen, kan tot grote vertragingen leiden en heeft dus vaak geen enkel nut. Indien deze data omwille van een of andere reden herzonden moeten worden zijn ze niet meer accuraat en onbruikbaar. Hier biedt UDP de oplossing. Reliable protocollen worden bijvoorbeeld gebruikt bij het inloggen of disconnecten van een client. Deze informatie moet zeker bij alle betrokken clients aankomen; het verloren gaan van deze packets kan een negatieve invloed op de simulatie hebben.

Bij de implementatie van een CVE moeten dus een aantal afwegingen gemaakt worden. Als synchronisatie belangrijker is dan de latencies die de gebruikers aanvaarden en als het geen large-scale NVE betreft, is een client-server netwerk, met de server als synchroniserende entiteit, aangewezen. Voor large-scale NVE's is een P2P netwerk beter geschikt, hoewel synchronisatie dan moeilijker is en meer netwerk verkeer vereist. Indien de CVE collision detection moet voorzien is synchronisatie een belangrijke factor. Voor real-time netwerk verkeer is het

TCP protocol te traag als het aantal gebruikers groot is. De packet loss die onvermijdelijk gepaard gaat met het gebruik van UDP moet zo goed mogelijk voor de gebruikers verborgen blijven.

Hoofdstuk 3

Implementatie

3.1 Inleiding

De focus van de implementatie ligt op het modelleren van een genetwerkte virtuele omgeving. Binnen deze omgeving kunnen verschillende gebruikers door middel van unieke avatars met elkaar interageren en kan er collaboratief gewerkt worden. Om het realisme van de applicatie te waarborgen, wordt collision detection voorzien. Om de samenwerking tussen de gebruikers vlot te laten verlopen, is het belangrijk dat het netwerk zo gestructureerd is dat iedere gebruiker op ieder ogenblik een geupdate toestand van de omgeving heeft. De concrete afwegingen die op dit vlak gemaakt moesten worden zijn het type van netwerk topologie en de te gebruiken protocollen voor ieder type van data packet.

Aangezien een goede synchronisatie levensnoodzakelijk is voor een correcte collision detection en een vlotte samenwerking tussen de gebruikers, is een client-server architectuur te prefereren boven een P2P architectuur (cfr. 2.9). Bovendien is de aandacht van de thesis in mindere mate gevestigd op het ontwikkelen van een large-scale NVE, en zal het aantal clients bijgevolg geen bottleneck vormen.

De eigenschap dat alle client informatie gecentraliseerd is op de server vereenvoudigt bovendien het collision detection proces. Een client voert zelf geen collision detection uit, maar laat alle beslissingen over aan de server. Dit garandeert dat de er geen collisions gemist worden aangezien een client zijn eigen informatie of die over de andere geconnecteerde clients slechts update indien hij de toestemming (in de vorm van een update boodschap) van de server ontvangt. Op die manier kan hij de virtuele omgevingen correct renderen.

Omwille van de redenen aangehaald in secties 1.7.1 en 1.7.2 wordt het SWIFT++ package gebruikt om collision detection uit te voeren.

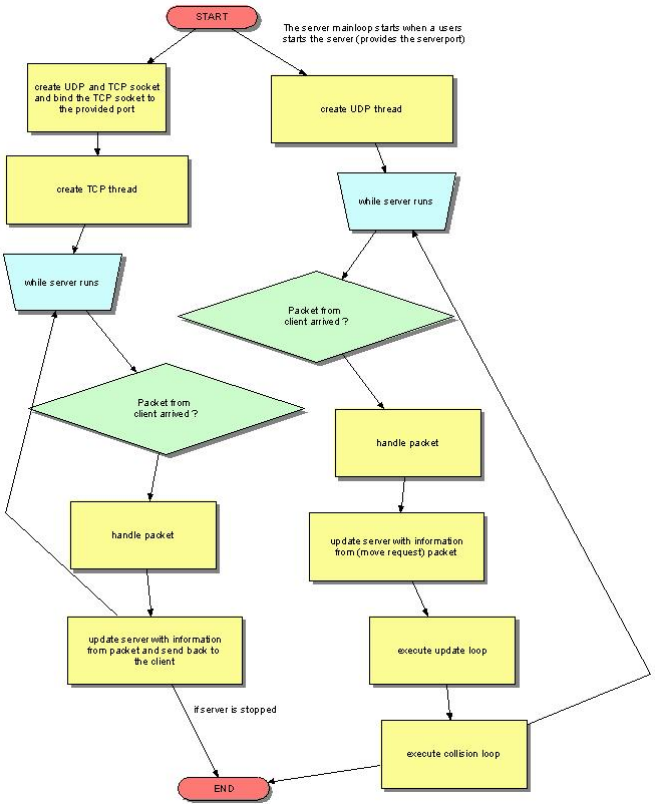
Zoals beschreven in sectie 2.4.3 heeft ieder soort data dat verzonden moet worden over een netwerk andere vereisten. Gegevens zoals login informatie en het disconnecten van clients, moeten zeker altijd aankomen en worden dus reliable verstuurd met TCP. De delays die optreden in het geval van retransmissies vormen in dit geval geen probleem.

Positie update messages moeten accuraat zijn. Het is in dit geval minder erg af en toe een packet te missen dan met sterk verouderde gegevens te moeten werken. Hoe het verloren gaan van packets gecompenseerd kan worden staat beschreven in sectie 2.8.

Sectie 3.2 en 3.3 beschrijven respectievelijk de uitvoeringslussen van de server en de client. Voor de server zijn dit de main loop (3.2) en de collision detection loop (3.2.2). In de client worden de main loop (3.3.1) en de rendering loop (3.3.2) onderscheiden.

In sectie 3.4 wordt dieper ingegaan op de manier waarop de gegevens tussen client en server gecommuniceerd worden. De experimentele resultaten betreffende het netwerk en de collision detection zijn terug te vinden in sectie 3.5.

3.2 Server



Figuur 3.1: De main loop van de server.

3.2.1 Main loop

De server kan opgestart worden door in de serverapplicatie de optie **start server** te kiezen in het **file** menu. Er verschijnt een dialoogbox op het scherm waarin de

poort waarop de server luistert kan ingevuld worden. Bovendien kan een wereld gekozen worden die de server zal runnen. Wanneer de server via deze dialoogbox wordt opgestart, is hij klaar om TCP en UDP packets te communiceren met de clients. Indien de server een packet van een client ontvangt, handelt hij het af en zendt hij een packet terug naar die client en eventueel ook naar de andere clients. Een uitzondering vormt het afhandelen van move requests van clients. De server zal eerst bepalen of de positie die de client wil innemen geen collision veroorzaakt. Enkel als er geen collisions zijn stuurt hij de geupdate positie naar alle clients. De main loop van de server (figuur 3.1) eindigt wanneer de server wordt afgesloten door in het menu file op stop server te klikken of door het venster af te sluiten. Een precies overzicht van welke packets met welk protocol gecommuniceerd worden volgt in sectie 3.4.

3.2.2 Collision Detection

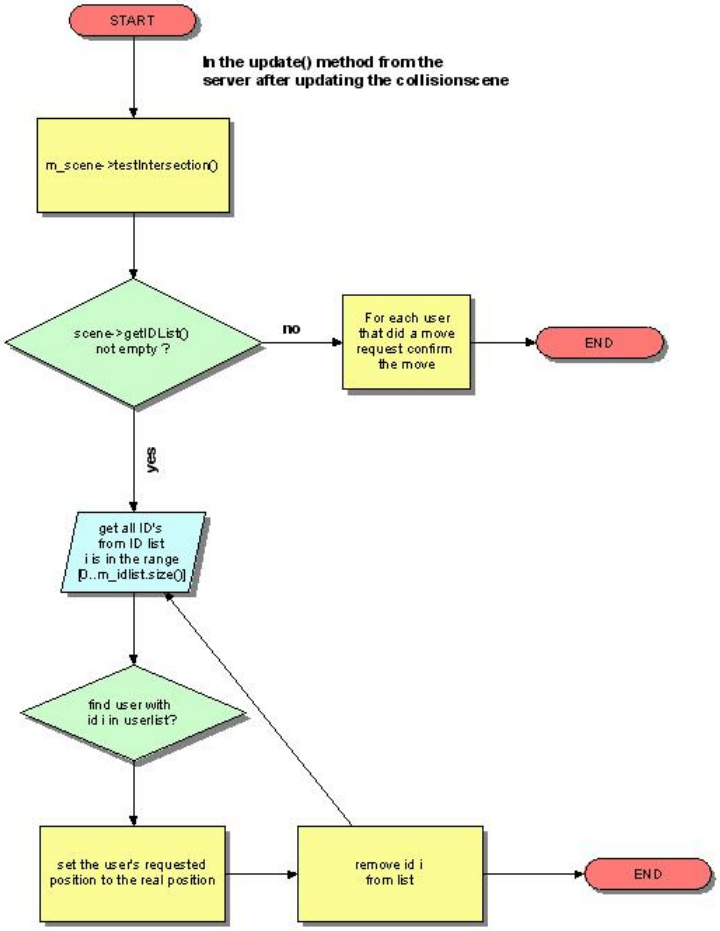
SWIFT++

```
TRI

8   // aantal vertices

12  // aantal faces

-0.5 0.5 -0.5   // Vertex met index 0
0.5 0.5 -0.5
-0.5 0.5 0.5
0.5 0.5 0.5          // Voor iedere vertex zijn 3 coördinaten
-0.5 -3.5 -0.5
0.5 -3.5 -0.5
-0.5 -3.5 0.5
0.5 -3.5 0.5

0 2 3
3 1 0
4 5 7
7 6 4
0 1 5
5 4 0       // Voor iedere face zijn vertex indices in tegenwijzerzin
1 3 7
7 5 1
3 2 6
6 7 3
2 0 4
4 6 2
```

Figuur 3.2: TRI file na converteren van een balk.

Het opsporen van collisions tussen de milkshape (ms3d) objecten in de virtuele omgeving gebeurt met het SWIFT++ collision detection package (sectie 1.7.2). De informatie die nodig is om een TRI bestand van een object aan te maken kan rechtstreeks uit het ms3d bestandsformaat van dat betreffende object gehaald worden.

Een voorbeeld van zo een TRI bestand is te zien in figuur 3.2. Het eerste getal dat na het woord **TRI** staat duidt op het aantal vertices waaruit het model bestaat. Vervolgens wordt het aantal faces gespecificeerd en volgt er een lijst van coördinaten voor iedere vertex. Tenslotte bevat het bestand voor iedere face zijn drie vertex indices gegeven in tegenwijzerzin. Uit sectie 1.7.2 weten we dat TRI bestanden enkel convexe objecten kunnen representeren. Indien de objecten een meer algemene vorm hebben, moeten de TRI bestanden met de

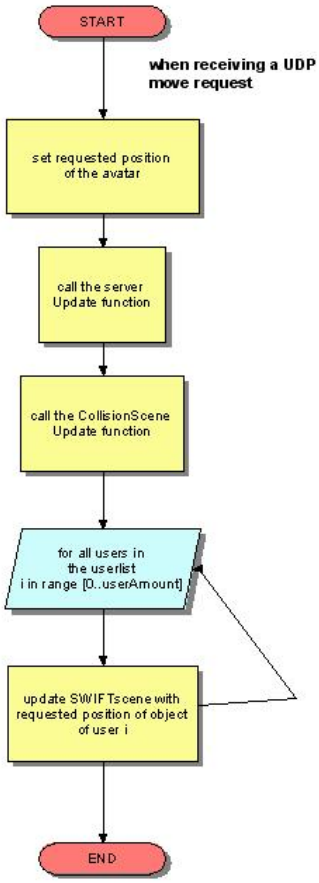


Figuur 3.3: De collision detection loop.

bijgeleverde SWIFT++ decomposer opgesplitst worden in een unie van convexe delen en opgeslagen in een ander bestandsformaat. Aangezien de decomposer voor een aantal modellen niet functioneerde en de geconverteerde bestanden bijgevolg onbruikbaar waren om in te laden, werden voor deze objecten de TRI bestanden van hun (convexe) bounding box gebruikt als input.

Collision detection loop

Iedere keer als de server van een client een aanvraag krijgt om te bewegen, gaat hij onderzoeken of de client een botsing veroorzaakt indien hij die beweging daadwerkelijk zou maken. Indien de beweging 'veilig' is stuurt de server een packet terug naar de client (en ook naar de andere geconnecteerde clients) om die beweging te bevestigen. Aangezien collision detection enkel server-side wordt uitgevoerd en de server steeds weet waar alle objecten uit de virtuele omgeving zich bevinden (objecten kunnen slechts geupdate worden als server bevestigt),



Figuur 3.4: De update loop van de server.

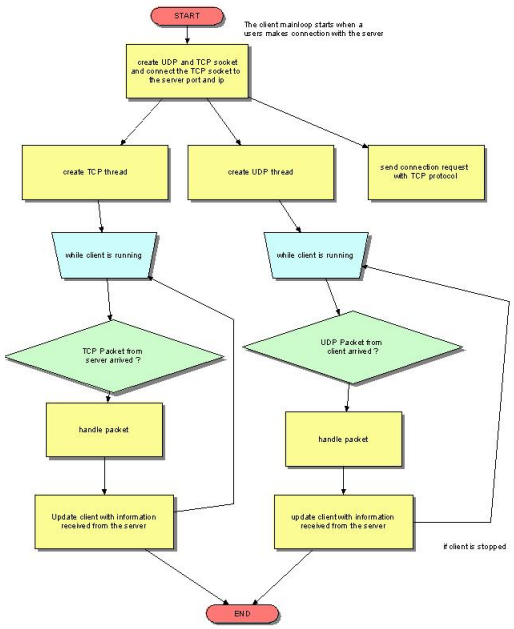
kunnen er onmogelijk collisions gemist worden. Een meer gedetailleerd overzicht van de updating en de collision detection wordt respectievelijk in figuur 3.4 en 3.3 gegeven.

3.3 Client

De client applicatie kan opgestart worden door in het menu **file** op **connect** te klikken en de gewenste gegevens in te vullen in de dialoogbox die vervolgens op het scherm verschijnt. Eens de gebruiker ingelogd is verschijnt hij in de virtuele omgeving waarin hij zich kan voortbewegen door middel van de pijltjestoetsen. Bovendien kan hij al de andere client avatars en hun bewegingen waarnemen. De camera staat gericht achter en boven de avatar van de gebruiker, zodat deze een 3d person view ervaart. Door dit viewpoint is het ook duidelijk zichtbaar dat collisions correct gedecteerd worden. De wereld bestaat uit een aantal vaste objecten en wordt afgebakend door een skybox. In afbeelding 3.6(a) wordt de view op de virtuele omgeving geïllustreerd. Figuur 3.6(b) is het beeld dat een

gebruiker (in dit geval de gebruiker aangeduid met de zwarte pijl) heeft van de virtuele omgeving en van de andere gebruikers aanwezig in de virtuele omgeving.

3.3.1 Main loop



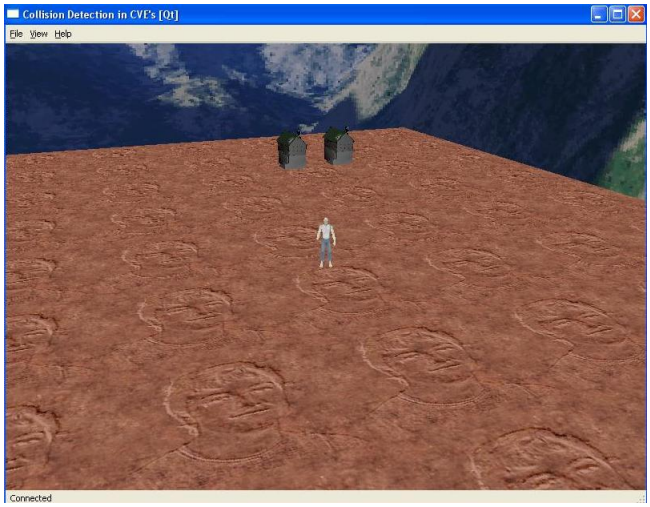
Figuur 3.5: De main loop van de client.

De client main loop start wanneer een gebruiker in de applicatie de gegevens (nickname, poort en avatar) in de connectie dialoogbox invult en bevestigt door op ok te drukken. Op dat ogenblik wordt een TCP en een UDP verbinding met de server opgezet. Packets van de server worden door de client bij ontvangst onmiddellijk afgehandeld. Hij update zijn status aan de hand van de informatie die het packet bevat. Het eerste packet dat door een client ontvangen wordt (in de TCP thread) is een connection confirm packet. Vanaf dat ogenblik wordt de renderloop van de client om de 50 milliseconden uitgevoerd. De main loop eindigt wanneer de client de verbinding met de server verbreekt door in het file menu van de applicatie op **disconnect** te klikken of door het venster te sluiten.

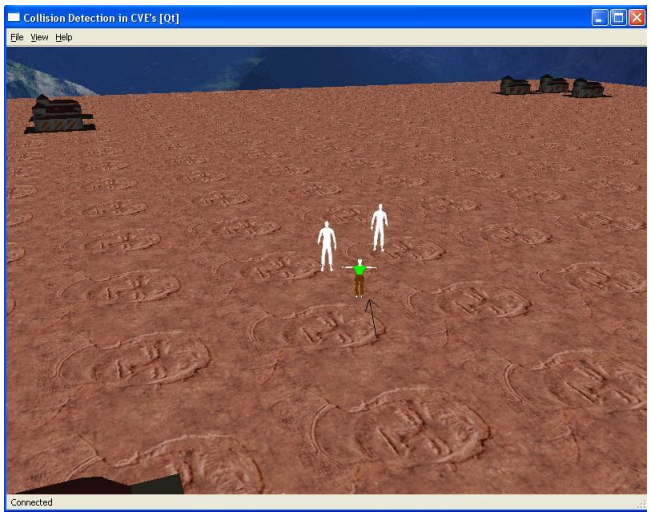
3.3.2 Render Loop

Rendering van een client gebeurt om de 50 milliseconden eens de client connectie heeft gemaakt met de server en stopt indien de client de verbinding met de server verbreekt.

In de render functie van de ClientView klasse worden de coordinaten van de camera gezet, zodanig dat hij zich achter en boven de avatar bevindt. Vervolgens wordt de render functie van de Client klasse aangeroepen. Het positioneren van



(a) 3rd person view.



(b) Verschillende gebruikers zijn gelijktijdig aanwezig in de virtuele wereld.

Figuur 3.6: Virtuele omgeving.

```
void Scene::render()
{
    if (m_world)
        m_world->render();
    if (m_objectlist)
    {
        SceneRenderVisitor visitor;
        m_objectlist->accept(&visitor);
    }
}

template <class USER_T>
void Userlist<USER_T>::accept(ObjectVisitor* visitor) {
    for (int i=0; i<getUserAmount(); i++)
        visitor->visitObject(m_userlist[i]->getAvatar());
}

void Scene::SceneRenderVisitor::visitObject(Object* object)
{
    object->render();
}
```

Figuur 3.7: Code fragment over Visitor Design Pattern voor de rendering van de avatars van alle gebruikers.

de camera achter het object zorgt er voor dat we een 3rd person view krijgen. De render functie van de client voert vervolgens de render functie van de Scene klasse uit. Deze laatste zorgt ervoor dat de wereldobjecten de avatars van alle clients worden gerenderd.

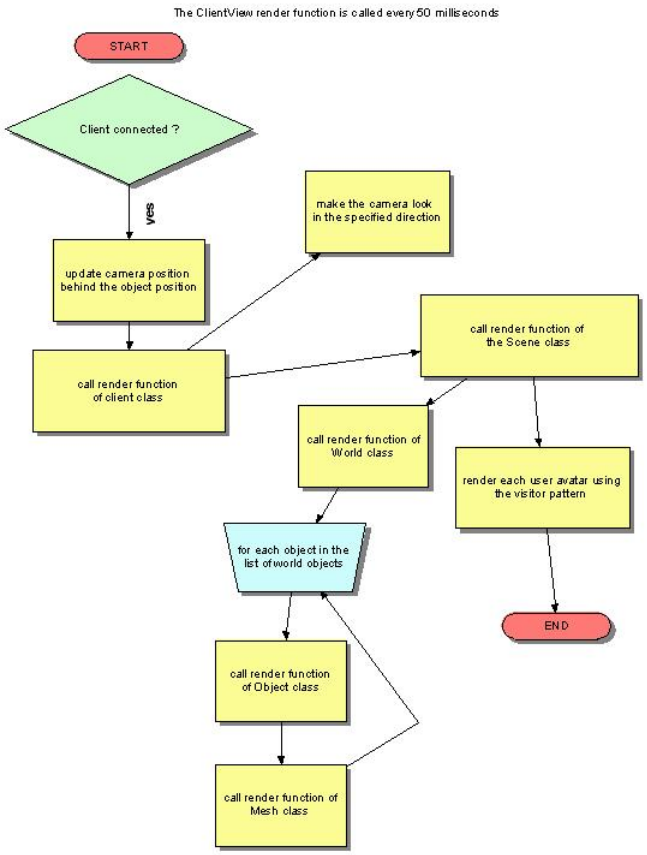
Om de avatar van iedere gebruiker uit de userlist te renderen wordt het Visitor design pattern gebruikt. De Scene klasse heeft een member Visitor klasse die de rendering voor zijn rekening neemt. Er wordt een call-back gedaan op de lijst van gebruikers. De aangeroepen **accept()** functie heeft als parameter de visitor en roept voor alle gebruikers de **visitObject()** functie uit de visitor klasse aan met als parameter hun avatar object. De render functie van de object klasse transleert en roteert het object, in die volgorde, alvorens het te renderen met een gespecificeerde schaleringsfactor.

3.4 Communicatie

Zoals in de inleiding van het hoofdstuk vermeld, wordt de communicatie in de virtuele wereld gerealiseerd met een client-server architectuur. Dit impliceert dat alle gecommuniceerde gegevens langs de server passeren. De gegevens worden gecommuniceerd door middel van packets. Deze packets hebben allemaal hetzelfde formaat: een header van 1 byte gevolgd door een payload (figuur 3.9).

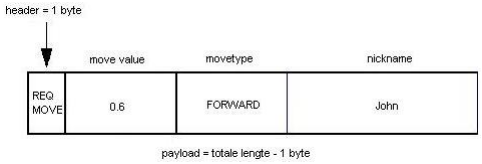
De headers zorgen ervoor dat de ontvanger van het packet kan afleiden om welk soort informatie het gaat. Indien het packet bijvoorbeeld de header REQ_MOVE heeft, weet de ontvanger (in dit geval de server) dat het om een verzoek van een client om te bewegen gaat en de payload uit de stapgrootte, het type van beweging en de nickname van de client bestaat.

Voor de meeste gegevens is het zonder meer duidelijk met welk protocol ze moeten verstuurd worden. Aanvragen van clients om een verbinding op te zetten met de server en het verbreken van deze verbinding zijn boodschappen die moeten aankomen bij de server en worden daarom reliable via TCP verstuurd.



Figuur 3.8: De rendering loop van de client.

Indien een nieuwe gebruiker zich aanmeldt bij de server, verwittigt deze de andere clients hiervan via TCP. De belangrijkste beslissing die op het gebied van



Figuur 3.9: Een packet bestaat altijd uit een header van 1 byte en een payload.

netwerk communicatie moest genomen worden is de keuze van een protocol om objectbewegingen tussen de clients en de server te communiceren. Zoals in de inleiding van dit hoofdstuk beschreven staat, stuurt iedere client bij het maken van een beweging een packet naar de server. De server controleert of de gewenste beweging van de client mogelijk is zonder dat er een collision optreedt. Indien de beweging 'veilig' blijkt te zijn, stuurt hij een bevestiging van de beweging naar alle clients. Aangezien de meeste gecommuniceerde packets aanvragen en beves-

tigingen van bewegingen zijn, kan een reliable protocol de simulatie aanzienlijk vertragen. Daarom worden zowel de de packets die de bewegingen aanvragen als de bevestigingen die de server terugstuurt via het unreliable UDP protocol gezonden. Omwille van de redenen geformuleerd in sectie 2.8 weten we dat het verlies van packets geen drastische invloed heeft op de performantie van de collision detection in de applicatie indien iedere client slechts beweegt als de server toestemming geeft.

Ook het doorsturen van een verandering in oriëntatie gebeurt best via UDP. Iedere client stuurt bij een draaibeweging de rotatiehoek naar de server. De server propageert deze informatie via UDP naar de andere clients, die de geroteerde client nu correct kunnen renderen.

3.5 Experimentele resultaten

Zoals reeds vermeld werd in sectie 3.1 werd er een client-server netwerk geïmplementeerd dat een combinatie van het UDP en het TCP protocol gebruikt om boodschappen tussen de clients en de server te communiceren. De implementatie werd aan een test onderworpen met als doel te bepalen of er merkbare vertragingen optreden indien een aantal gebruikers inloggen op de centrale server en te controleren of de collision detection goed functioneert. Er werd door vier gebruikers gelijktijdig deelgenomen aan deze test, waarvan er geen enkele vertragingen in de navigatie ervaarde. De collision detection tussen de gebruikers en de vaste objecten in de wereld en tussen de gebruikers onderling werd eveneens uitvoerig getest (in het bijzonder voor snel naar elkaar toebewegende objecten) en bleek correct te werken.

3.6 Besluit en mogelijke uitbreidingen

De implementatie is een verwerking van de theoretische kennis over CVE's en collision detection en voorziet oplossingen voor de problemen die zich stellen bij de integratie van beiden. Gegeven het feit dat er binnen de virtuele omgeving collision detection nodig is, is een client-server netwerk de meest voor de hand liggende keuze. Door enkel de collision detection server-side uit te voeren kunnen er immers nooit collisions gemist worden (zie sectie 2.8.1). Indien een large-scale CVE vereist is, zal een client-server topologie niet afdoende snel zijn. In dat geval kan men beter een P2P netwerk gebruiken, maar dan zijn de synchronisaties uiteraard veel moeilijker.

De clients en de server communiceren zowel via TCP als UDP met elkaar. Een gedetailleerder overzicht van welke packets via TCP en welke via UDP worden gezonden is te vinden in sectie 3.4. De experimentele resultaten, zoals beschreven in sectie 3.5, tonen aan dat de keuze voor een client-server topologie, gebruik makend van een combinatie van TCP en UDP, gerechtvaardigd is.

Mogelijke uitbreiding van deze applicatie zijn een interessante gameplay of eventueel teleconferencing al naargelang de aard van de gewenste toepassing. Een andere uitdaging is om de collision detection ook mogelijk te maken voor vervormbare objecten.

Hoofdstuk 4

Conclusie

De meeste computers zijn tegenwoordig uitgerust met snelle grafische hardware. Dit biedt software ontwerpers de mogelijkheid om hun applicaties zo realistisch mogelijk te maken voor de modale computer gebruiker. Een factor die bijdraagt tot realisme is te voorkomen dat objecten kunnen interpenetreren door tijdig collisions te detecteren en een gepaste respons hierop te verschaffen.

Het real-time uitvoeren van collision detection is tegenwoordig mogelijk dankzij gespecialiseerde algoritmes, snelle processoren en een grote geheugencapaciteit. Real-time collision detection is niet mogelijk door alle objectparen onderling te testen op collisions (tijdscomplexiteit $O(n^2)$) als er sprake is van een groot aantal objecten. Daarom zijn alle real-time collision detection algoritmes opgesplitst in twee fases. De zogenaamde broad phase zorgt ervoor dat objecten die niet van belang zijn tijdens een collision test ook daadwerkelijk niet getest worden. De volgende fase, die narrow phase collision detection wordt genoemd, bepaalt de precieze delen van de objecten die interfereren.

Uit ervaring weten we dat er zich bijkomende problemen stellen wanneer collision detection moet voorzien worden in een collaboratieve virtuele omgeving. Gemiste collisions door trage of foutieve netwerkcommunicatie kunnen desastreuze gevolgen hebben voor de gebruikers en doen de werking van een collision detection algoritme teniet.

Een eerste afweging die gemaakt moet worden is welke netwerk architectuur te gebruiken. Aangezien er in geen geval collisions tussen de gebruikers mogen gemist worden, moet iedere gebruiker de posities van alle andere gebruikers kennen. In het geval van een P2P netwerk moet iedere peer input messages ontvangen en zenden naar iedere andere peer. Dit vormt een serieuze bottleneck.

Het voordeel van een client-server netwerk is dat de server (als tussenliggend station tussen de clients onderling) steeds over de meest recente informatie van iedere client beschikt. Indien iedere client elke beweging aanvraagt aan de server kunnen er nooit collisions gemist worden; de client zal enkel bewegen indien de server zijn aanvraag bevestigt (d.w.z. als de aangevraagde positie geen collision veroorzaakt). Het nadeel van client-server is dat de centrale computer een bottleneck wordt als het aantal processen groot wordt. Het gebrek aan een centrale entiteit in een P2P netwerk impliceert veel moeilijkere synchronisatie

en dus meer kans op het missen van collisions. Mede daardoor en door het feit dat een large-scale NVE niet het opzet van deze thesis was, is een client-server netwerk een verantwoorde keuze.

Een bijkomend probleem vormt de keuze van een geschikt netwerk protocol. Voor de meeste te communiceren data is het onmiddellijk duidelijk welk protocol er gebruikt moet worden. Enkel de keuze van het protocol dat gebruikt wordt om de bewegingen van de clients te communiceren verdient extra aandacht. Wat bewegingen van gebruikers betreft is verouderde informatie niet meer zinvol. Dit betekent dat duplicaten sturen bij foute transmissies enkel overhead is. Beter dan TCP te gebruiken, is de packet loss inherent aan UDP, zo op te vangen dat de gebruiker er zo weinig mogelijk hinder van ondervindt. Packet loss kan niet helemaal uitgeschakeld worden, maar dead reckoning kan er voor zorgen dat er minder connecties en boodschappen nodig zijn en dat er bijgevolg ook minder packet loss is.

Na al deze afwegingen te hebben gemaakt, werd besloten om een client-server netwerk te implementeren en een combinatie van UDP en TCP te gebruiken. Uit de testen die uitgevoerd zijn bleek dat de collision detection goed functioneert en dat er tijdens de navigatie door de gebruikers geen merkbare vertragingen worden waargenomen indien meerdere clients geconnecteerd zijn met de server.

Bibliografie

[1] http://www.it.iitb.ac.in/~jaju/tutorials/net/tcpip/.

[2] Bsp tree. http://encyclopedia.thefreedictionary.com/BSP%20tree.

[3] Client-server computing. http://www.unm.edu/~network/presentations/course/appendix/appendix_k/sld033.htm.

[4] Collision detection. http://www.cs.brown.edu/courses/gs007/lect/sim/web/murat.html.

[5] Collision detection en response. http://didactiek.edm.luc.ac.be/aac/coldet_slides.ppt.

[6] Constructive solid geometry. http://www.olympus.net/personal/mortenson/preview/definitionsc/constructivesolidgeometry.html.

[7] Dataglove. http://www.ics.uci.edu/~kobsa/courses/ICS104/course-notes/dataglove.jpg.

[8] Deep: Dual-space expansion for estimating penetration depth. http://www.cs.unc.edu/~geom/DEEP/.

[9] Discrete orientation polytopes (k-dops). http://www.ams.sunysb.edu/~jklosow/quickcd/QCD_kdops.html.

[10] The dive homepage. http://www.sics.se/dive.

[11] Doom. http://games.activision.com/games/doom.

[12] I-collide. http://www.cs.unc.edu/~geom/I_COLLIDE/.

[13] Mdk2 armageddon. http://www.bioware.com/games/mdk2_armageddon/.

[14] Minkowski sum. http://mathworld.wolfram.com/MinkowskiSum.html.

[15] Motivation for bsp trees: The visibility problem. http://graphics.csail.mit.edu/classes/6.838/F01/lectures/BSP/BSP2D.pdf.

[16] Neverwinter nights. http:://nwn.bioware.com.

[17] Optimal oriented bounding boxes. http://www.mlahanas.de/CompGeom/opt_bbox.htm.

[18] Picture quadtree. http://www.cs.iusb.edu/~danav/teach/c481/quadtree.gif.

[19] Pqp - a proximity query package. http://www.cs.unc.edu/~geom/SSV/.

[20] Prince of persia.

[21] Qhull. http://www.qhull.org.

[22] Quake 2. http://www.idsoftware.com/games/quake/quake2/.

[23] Rapid version 2.01. http://www.cs.unc.edu/~geom/OBB/OBBT.html.

[24] Software library for interference detection. http://www.win.tue.nl/~gino/solid/.

[25] Udp broadcast flooding. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ics/cs006.htm.

[26] Unreal. http://www.unreal.com/.

[27] V-clip collision detection library. http://www.merl.com/projects/vclip/.

[28] V-collide: Collision detection for arbitrary polygonal objects. www.cs.unc.edu/~geom/V_COLLIDE.

[29] What is a network operating system? http://fcit.usf.edu/network/chap6/chap6.htm.

[30] Wikipedia, the free encyclopedia. http://en.wikipedia.org.

[31] Project Number Acts. D2.6 guidelines for building cve applications.

[32] Jesse Aronson. Dead reckoning: Latency hiding for networked games. September 1997.

[33] Gino Van Den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *Journal of Graphics Tools: JGT*, 4(2):7–25, 1999.

[34] Jonathan Blow. Practical collision detection.

[35] Nick Bobic. Advanced collision detection techniques.

[36] Shawn Bonham, Daniel Grossman, William Portnoy, and Kenneth Tam. Quake: An example multi-user network application : problems and solutions in distributed interactive simulations. May 31, 2000.

[37] Paul Bourke. Implicit surfaces. http://astronomy.swin.edu.au/~pbourke/modelling/implicitsurf/.

[38] Gareth Bradshaw. *Bounding Volume Hierarchies for Level-of-Detail Collision Handling*. PhD thesis, Trinity College Dublin, May 2002.

[39] Robert F. Buchheit. Delay compensation in networked computer games. January 2004.

[40] Jonathan C. Carr, Richard K. Beatson, Jon B. Cherrie, Tim J. Mitchell, W. Richard Fright, Bruce C. McCallum, and Tim R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 67–76. ACM Press / ACM SIGGRAPH, 2001.

[41] Charlton. *Eric Charlton's Ph.D. Dissertation.* PhD thesis. `http://hpcc.engin.umich.edu/CFD/users/charlton/Thesis/html`.

[42] Yam San Chee. Networked virtual environments for collaborative learning. In *Proceedings of ICCE/SchoolNet 2001-Ninth International Conference on Computers in Education, Seoul, S. Korea*, pages 3–11, 2001.

[43] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995.

[44] Paul Dawkins. Paul's online math tutorials and notes. `http://tutorial.math.lamar.edu/AllBrowsers/2415/ParametricSurfaces.asp`.

[45] Stephen Ehmann. Speedy walking via improved feature testing for non-convex objects. `http://www.cs.unc.edu/~geom/SWIFT++`.

[46] Stephen Ehmann. Swift - speedy walking via improved feature testing.

[47] David EppStein. Quadtrees and hierarchical space decomposition.

[48] Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. Buckettree: Improving collision detection between deformable objects, 2000.

[49] Nicolas D. Georganas. Collaborative virtual environments. Technical report, School of Information Technology and Engineering, University of Ottawa, 2004.

[50] E. G. Gilbert, D. W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journ. Of Robotics and Automation*, 4:193–203, 1988.

[51] R. Gossweiler, R. J. Laferriere, M. L. Keller, and R. Pausch. An introductory tutorial for developing multiuser virtual environments. *Presence*, 3(4):255–264, 1994.

[52] S. GottSchalk, M.C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[53] Patrick Lyle Hartling. *Octopus: A study in collaborative virtual environment implementation.* PhD thesis, Iowa State University Ames, 2001.

[54] Chris Joslin, Igor S. Pandzic, and Nadia Magnenat Thalmann. Trends in networked collaborative virtual environments.

[55] JongWon Kim. Networking application design: Origin of networked virtual environment.

[56] Y. Kim, M. Lin, and D. Manocha. Deep: Dual-space expansion for estimating penetration depth between convex polytopes. In *IEEE Conference on Robotics and Automation*, 2002.

[57] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[58] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *Knowledge Discovery and Data Mining*, pages 16–22, 1999.

[59] Eric Larsen. Collision detection for real-time simulation.

[60] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast proximity queries with swepth sphere volumes. Technical Report 018, Department of Computer Science, University of N. Carolina, Chapel Hill., 1999.

[61] Jason Leigh and Andrew E. Johnson. Supporting transcontinental collaborative work in persistant virtual environments. *IEEE Computer Graphics and Applications*, 16(4):47–51, 1996 1996.

[62] M. Lin, D. Manocha, and J. Cohen. Collision detection: Algorithms and applications. In *Proceedings of the 2nd Workshop on Algorithmic Foundations of Robotics*, 1996.

[63] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. pages 37–56.

[64] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation.

[65] Ming C. Lin and Dinesh Manocha. Collision and proximity queries.

[66] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31(Annual Conference Series):199–208, 1997.

[67] Denis Lukianov. Advanced winsock multiplayer game programming: Multicasting. `http://www.gamedev.net/reference/articles/article1587.asp`.

[68] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A network software architecture for large-scale virtual environment. *Presence*, 3(4):265–287, 1994.

[69] Stan Melax. Bsp collision detection as used in mdk2 and neverwinter nights. `http://www.gamasutra.com/features/20010324/melax01.htm`.

[70] Brian Mirtich. Lin-canny closest features algorithm.

[71] Brian Mirtich. Rigid body contact: Collision detection to force computation. Technical Report 01, 201 Broadway, Cambridge, Massachusetts 02139, 1998.

[72] Robert Dunlop Microsoft DirectX MVP. Collision detection, part 1: Using bounding spheres. http://www.mvps.org/directx/articles/using_bounding_spheres.htm.

[73] Institute of Electrical and Electronics Engineers. Protocols for distributed interactive simulation. March 1993.

[74] Chong Jin Ong. The gilbert-johnson-keerthi distance algorithm: A fast version for incremental motions.

[75] I.J. Palmer and R.L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[76] Arthur Pope. Bbn report no. 7102. *The SIMNET Network and Protocols*, July 1989.

[77] M. Reggiani, M. Mazzoli, and S. Caselli. An experimental evaluation of collision detection packages for robot motion planning, 2002.

[78] Keith W. Ross and James F. Kurose. 3.3 connectionless transport: Udp. http://www-net.cs.umass.edu/kurose/transport/UDP.html.

[79] J. Rossignac and A. Requicha. Solid modeling, 1999.

[80] Philip J. Schneider. Nurb curves: A guide for the uninitiated. http://www.mactech.com/articles/develop/issue_25/schneider.html.

[81] Shervin Shirmohammadi and Nicolas D. Georganas. Collaborating in 3d virtual environments: A synchronous architecture.

[82] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments - Design and Implementation*. Acm Press, 1999.

[83] Andrew Smith, Yoshifumi Kitamura, Haruo Takemura, and Fumio Kishino. A simple and efficient method for accurate collision detection among deformable objects in arbitrary motion. In *the IEEE Virtual Reality Annual International Symposium*.

[84] Nilo Stolte and Arie KAUFMAN. Discrete implicit surface models using interval arithmetics.

[85] Rory Stuart. *The Design of Virtual Environments*. Mcgraw-Hill, New York, 1996.

[86] C.J. Su, F.H. Lin, and B.P. Yen. An adaptive bounding object based algorithm for efficient and precise collision detection of csg-represented virtual objects.

[87] Andrew S. Tanenbaum. *Computer Networks*. fourth edition, 2003.

[88] Greg Turk and James F. O Brien. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics*, 21(4), October 2002.

[89] Joe van den Heuvel and Miles Jackson. Pool hall lessons: Fast, accurate collision detection between circles or spheres.

[90] A. Wilson, E. Larsen, and D. Manocha. Immpact: a system for interactive proximity queries on massive models, 1998.