Faculteit Toegepaste Wetenschappen
Vakgroep Informatietechnologie
voorzitter: prof. dr. ir. P. Lagasse

3D Polygonale Bundeltrekker voor Akoestische Simulaties

Bram de Greve

promotor: prof. dr. ir. D. Botteldooren
begeleider: ir. T. De Muer

afstudeerwerk ingediend tot het behalen van de graad van burgerlijk elektrotechnisch ingenieur, optie toegepaste elektronica

academiejaar 2002–2003

Woord vooraf

Een afstudeerwerk maken doet niemand alleen. Steeds is er de steun van anderen die je helpen deze taak tot een goed einde te brengen. Graag zou ik die mensen willen bedanken die mijn steun en hulp waren voor dit eindwerk, en in het bijzonder:

- Freddy Allemeersch, voor het enthousiasme waarmee hij ons de kunst van de ruimtemeetkunde aanleerde.
- Alle bezoekers van flipCode, een thuishaven voor ieder die houdt van 3D en `C++`, in het bijzonder Nicolas Capens, Dirk Gerrits, Jarkko Lempiainen, Max McGuire, Kurt Miller, Jaap Suter en Per Vognsen.
- Mijn ouders en broer, voor de constante steun en motivatie die ze me gaven tijdens het schrijven van dit eindwerk.
- mijn begeleider ir. Tom De Muer, voor zijn goede begeleiding en enorme hulp bij het oplossen van technische problemen.
- mijn promotor prof. dr. ir. D. Botteldooren, voor het scheppen van de mogelijkheid om dit eindwerk te verrichten, zijn begeleiding en goede raad

Toelating tot bruikleen

27 mei 2003
Bram de Greve

3D Polygonale Bundeltrekker voor Akoestische Simulaties

Bram de Greve

afstudeerwerk ingediend tot het behalen van de graad van burgerlijk elektrotechnisch ingenieur, optie toegepaste elektronica

academiejaar 2002–2003

Universiteit Gent
Faculteit Toegepaste Wetenschappen
Vakgroep Informatietechnologie
voorzitter: prof. dr. ir. P. Lagasse

promotor: prof. dr. ir. D. Botteldooren
begeleider: ir. T. De Muer

Samenvatting

`bass3` is een 3D polygonale bundeltrekker voor akoestische simulaties, geschreven in `C++`. Het is in staat om in volledige 3D omgevingen geluidspaden te genereren vanaf een opgegeven bron naar ontvangers in de omgeving. Het brengt hierbij zowel speculaire reflectie als diffractie in rekening. Diffractie werd geïmplementeerd aan de hand van de uniforme diffractie theorie. Tegelijkertijd is voor een flexibele architectuur gezorgd, zodat `bass3` eenvoudig op verschillende domeinen kan worden ingezet. Zo is er een directe koppeling met `ABT` gerealiseerd, een 2.5D bundeltrekker in de onderzoeksgroep *Acustica*.

Trefwoorden: bundeltrek, diffractie, geometrische akoestiek.

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Geometrische akoestiek

Numeriek methodes voor het oplossen van akoestische veldproblemen kan men opsplitsen in twee grote categorieën: *golf akoestiek* en *geometrische akoestiek* [5]. Golf akoestiek bevat methodes als eindige elementen, eindige differentie en randdiscretisatie. Deze trachten de golfvergelijking op te lossen door de te simuleren omgeving in de ruimte en in de frequentie of in de tijd te discretiseren. Het voordeel van deze methodes is dat ze de golfvergelijking in alle aspecten volledig oplossen. Maar om goede resultaten te verkrijgen moet men dit zeer fijn ten opzichte van de golflengte doen, wat voor grote omgevingen tot een onaanvaardbaar groot geheugengebruik kan leiden.

Geometrische akoestiek omvat methodes als het *spiegelbronmodel (Eng.: image sources)*, *stralentrek (Eng.: ray tracing)*, en *bundeltrek (Eng.: beam tracing)* [8, 13]. Deze methodes verwaarlozen in eerste instantie het golfkarakter van geluid. Men kan dan geluid beschouwen als deeltjes die door de geluidsbron worden uitgestraald, zich een weg banen door de ruimte, eventueel op een wand speculair weerkaatsen, om tenslotte de waarnemer te bereiken. De baan die het deeltje beschrijft wordt dan als een *geluidsstraal* beschouwd. Deze stralentheorie is volledig analoog aan de geometrische optica, waardoor we vele parallellen kunnen trekken.

Het grootste probleem bij geometrische akoestiek is de verwaarlozing van diffractie, interferentie en diffuse reflectie. De stralentheorie is namelijk een benadering van de golfvergelijking waarbij we deze in de limiet $f \to \infty$ nemen. Dit betekent dat we de golflengte zeer klein veronderstellen, of in de limiet $\lambda \to 0$. Voor de meeste problemen is de golflengte echter niet zeer klein in vergelijking met de karakteristieke afmeting van de omgeving, waardoor diffractie en interferentie wel een rol spelen. Bijkomende maatregelen moeten worden genomen om ook deze fenomenen te simuleren.

Voor interferentie kan een eenvoudige oplossing worden gevonden. Alle methodes in de geometrische akoestiek zullen geluidsstralen genereren. Hiervan kan men snel de *afgelegde weglengte* bereken, en deze tezamen met het aantal reflecties vertalen naar een fase waarmee

de verschillende vermogens worden opgeteld. Het ontbreken van diffractie en diffuse reflectie is minder eenvoudig op te lossen. Diffractie werd in dit eindwerk aangepakt met behulp van de Uniforme Diffractie Theorie (UDT) [29] zoals in 2.4 wordt besproken. Diffuse reflectie werd niet behandeld.

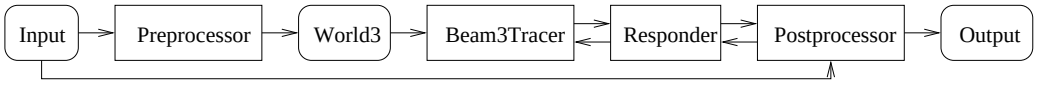
1.2 Doel van deze scriptie

In deze scriptie wordt het ontwerp en de implementatie behandeld van een *3D polygonale bundeltrekker* [9], een algoritme dat in de categorie van geometrische akoestiek valt. De belangrijkste eisen voor het ontwerp zijn de volledige drie-dimensionaliteit en de inbouw van diffractie. Tegelijkertijd is ook zeer veel aandacht besteed aan het zo generiek mogelijk houden van de bundeltrekker, zodat ze zonder veel moeite in wisselende problemen kan worden ingezet. Zo werd deze reeds gebruikt in een optische simulatie van een stadsomgeving. Het resultaat kreeg de naam `bass3`, wat staat voor *"3D Bundeltrekker voor Akoestische SimulatieS"*, of ook *"3D Beamtracer for Acoustical SimulationS"*.

Eerst zullen we bekijken welke technieken er bestaan om geometrische akoestiek uit te voeren, en hoe het 3D polygonale bundeltrekken daar in past. Daarna zullen we overgaan tot een bespreking van `bass3` zelf.

1.3 Architectuur van bass3

We geven hier kort een overzicht van de architectuur van `bass3`. Hiermee kunnen we de verschillende hoofdstukken in een beter kader plaatsen.



Figuur 1.1: architectuur van `bass3`

Input: de invoer data die alle gegevens bevat over de te simuleren omgeving. Dit omvat de akoestische eigenschappen van materialen, de plaatsing van ontvangers, de plaatsing van bronnen.

Preprocessor: maakt vanaf de invoer data een `World3` object aan. Dit proces wordt in het bijzonder voor 2.5D informatie besproken in hoofdstuk 4.

World3: het object dat de te simuleren omgeving voorstelt, klaar om door de `Beam3Tracer` component te worden verwerkt. Dit bevat de geometrie van de omgeving en de plaatsing van de ontvangers. Naar akoestische eigenschappen worden enkel handles (handvatten) bijgehouden. De opbouw van dit object wordt besproken in hoofdstuk 3.

Beam3Tracer: de eigenlijke bundeltrekker, bepaalt vanuit een opgegeven bron potentieel zichtbare ontvangers en geeft die met een `Beam3Stack` (hier niet afgebeeld) door aan een `Responder`. De `Beam3Tracer` wordt tezamen met `Beam3Stack` in hoofdstuk 5 beschreven.

Responder: de schakel tussen `Beam3Tracer` en de postprocessor. Het ontvangt potentieel zichtbare ontvangers van de `Beam3Tracer` met bijbehorende `Beam3Stack`. Daarvan bepaalt het een geluidspad tussen bron en ontvanger en controleert of dit geometrisch gezien ook geldig is. Daarna geeft het dit geluidspad door aan de postprocessor. Tegelijkertijd informeert het de `Beam3Tracer` ook van de geldigheid van het pad, en zal ook een aantal niet-geometrische vragen voor de `Beam3Tracer` oplossen. Het algemeen principe van de `Responder` wordt besproken in 5.3, en een specifieke `Responder` voor ABT wordt in hoofdstuk 6 uiteengezet.

Postprocessor: ontvangt van de `Responder` een geluidspad tussen bron en een ontvanger, en zal met behulp van de akoestische gegevens uit de invoer data de nodige akoestische informatie over dit geluidspad berekenen, zoals de afgelegde weg, de fasedraaing, de attenuatie, ... Deze informatie wordt tenslotte bij de uitvoer data weggeschreven. Tegelijkertijd zal het ook de `Responder` informeren over deze akoestische eigenschappen, zodat deze opnieuw de `Beam3Tracer` kan bijsturen. De postprocessor is iets dat niet in `bass3` is ingebouwd, maar er wordt wel een specifieke postprocessor voor ABT in hoofdstuk 6 besproken.

Output: de uitvoer data die door de postprocessor wordt gegenereerd.

Hoofdstuk 2

Geometrische akoestiek

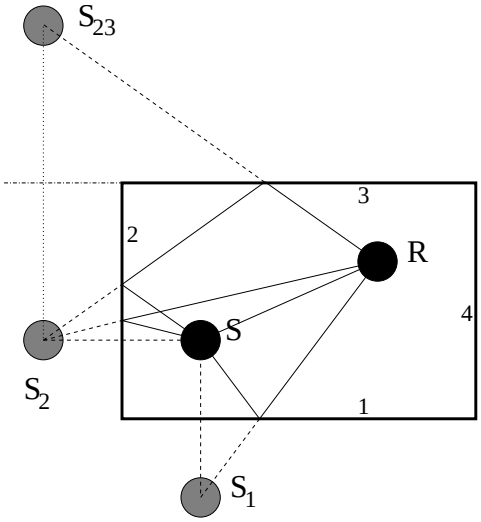
Vooraleer wordt overgegaan tot een gedetailleerde bespreking van `bass3`, zullen eerst een aantal methodes worden geschetst uit de geometrische akoestiek. In dit kader wordt het bundeltrekken geplaatst en uitvoeriger besproken.

2.1 Spiegelbronmodel

Bij het spiegelbronmodel zal men enkel speculaire reflectie beschouwen, omdat in dat geval de geluidspaden – die we hier *reflectiepaden* zullen noemen – triviaal te berekenen zijn. Men kan ze immers vinden door virtuele bronnen S_i te plaatsen die verkregen worden door de geluidsbron S te spiegelen ten opzichte van elke polygonale wand Ω_i in de wereld. Voor iedere virtuele bron S_i kan dan het speculaire reflectiepad worden gevonden door het snijpunt van de wand Ω_i met het lijnstuk tussen S_i en de ontvanger R verb indt. Men kan dit proces recursief herhalen waardoor reflectiepaden van gelijk welke orde kunnen worden berekend. De orde van een reflectiepad is het aantal wanden waarop het reflecteert alvorens de ontvanger te bereiken.

Het grote voordeel van deze methode is dat ze zeer robuust is, en ze alle reflectiepaden tot een gegeven orde of padlengte zal berekenen. Er zijn echter twee grote nadelen aan verbonden

- Het spiegelbronmodel houdt enkel rekening met speculaire reflecties. Diffractie en diffuse reflectie worden verwaarloosd.
- Het aantal virtuele bronnen stijgt exponentieel met de maximum reflectie orde r: er worden $O\left(n^r\right)$ virtuele bronnen gegenereerd bij n wanden in de wereld en reflectie orde r. Bovendien zijn veel van de gegenereerde paden ongeldig. Een pad kan bijvoorbeeld geblokkeerd worden door een extra wand. Om enkel geldige paden in de berekening op te nemen, moeten er dus complexe zichtbaarheidstesten per pad worden uitgevoerd. Bovendien, moet dit alles gebeuren per bron–ontvanger paar.

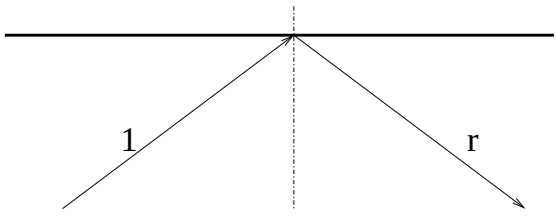


Figuur 2.1: spiegelbronmodel

Zelfs als we in de wereld een rooster van *"slechts"* 256×256 bronnen plaatsen, dan moet deze hele procedure reeds 65536 keren worden herhaald. En dit voor iedere bron!

2.2 Stralentrek (ray tracing)

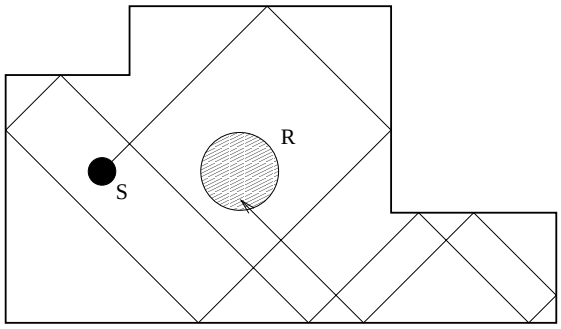
Deze methode zal geluidspaden genereren door stralen te laten vertrekken bij de bron, en deze volgens bepaalde regels te laten propageren door de ruimte. Het eenvoudigste is om speculaire reflectie te veronderstellen, waar bij reflectie er precies één weerkaatste straal is volgens de gekende wet: invalshoek = reflectiehoek en vermogen in de straal daalt met een factor r. Refractie is op gelijkaardige wijze te implementeren.



Figuur 2.2: speculaire reflectie met attenuatie factor r

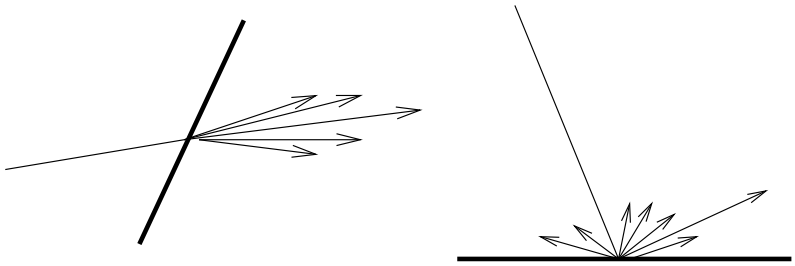
Men kan de stralen aan de bron laten vertrekken volgens een regelmatig patroon, of deze met willekeurige richting volgens een uniforme distributie genereren. In dit laatste geval spreekt men van een *Monte Carlo* simulatie. Als een straal een ontvanger passeert, dan wordt voor de ontvanger de bijdrage van de bron berekend. Men stopt met het volgen van de straal als de straal een maximum aantal reflecties heeft bereikt, of als het vermogen in de straal beneden een bepaald tolerantieniveau is gezakt. Een ontvanger kan hier niet als

een punt worden voorgesteld, omdat de kans dat een straal een ontvanger snijdt gelijk is aan 0. Als oplossing neemt men een bol of kubus als ontvanger.



Figuur 2.3: stralentrek

Het grootste voordeel van deze methode is wel de eenvoud. Stralentrek is volledig gebaseerd op straal-oppervlak intersectie. Dit is eenvoudig te realiseren, zelfs met kromme oppervlakken, en de complexiteit groeit slechts sublineair met het aantal wanden in de wereld. Stralentrek gaat uit van bemonstering van de continue wereld rondom de bron (er worden slechts stralen getrokken door een discreet aantal punten rond de bron), en met die gedachte kunnen fenomenen als diffractie en diffuse reflectie geïmplementeerd worden. Men selecteert één of meerdere stralen uit het oneindig aantal dat door diffractie of diffuse reflectie wordt veroorzaakt. Voor diffractie worden deze gegeven door de Uniforme Diffractie Theorie (sectie 2.4) en voor diffuse reflectie door de cosinus wet van Lambert.



Figuur 2.4: discretisatie bij stralentrek: diffractie en diffuse reflectie

Deze bemonstering is tevens ook het grootste nadeel van stralentrek. Belangrijke geluidspaden kunnen volledig worden gemist omdat geen enkele gegenereerde straal dit pad volgt. Ze slippen als het ware door de mazen van het net. Als compensatie zal men een zeer groot aantal stralen genereren waardoor men hoopt de fout zo klein mogelijk te maken, maar deze is nooit volledig weg te werken. Men kan de mazen van een net zeer fijn maken, maar nooit volledig wegwerken. Uiteraard stijgt hiermee de rekentijd, wat meestal onwenselijk is.

2.3 Bundeltrek (beam tracing)

Tenslotte zijn we aangekomen bij de methode die is gebruikt voor `bass3`: bundeltrek, of *beam tracing*. Men onderscheidt een aantal varianten: *conische*, *piramidale* en *polygonale* bundeltrek, waarbij de kwaliteit van de methode verbetert in deze volgorde. We bespreken ze in deze zelfde volgorde.

2.3.1 Conische en piramidale bundeltrek

Deze twee varianten kan men beschouwen als stralentrek met *"dikke stralen"*.

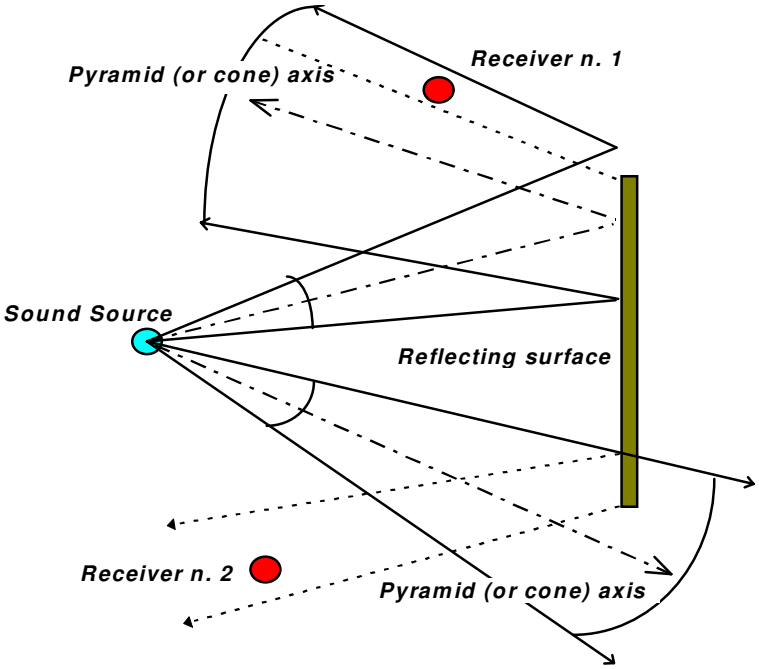
Conische bundeltrek

Het probleem met stralentrek zijn de oneindig dunne stralen. Daarom zal men bij conische bundeltrek deze stralen vervangen door kegels met cirkelvormige doorsnede. Daardoor krijgen de stralen – die we nu bundels noemen – een zekere doorsnede waarmee de kans vergroot dat ze een ontvanger zullen raken. Vergelijk dit met de methode bij stralentrek waar men de ontvangers een zekere doorsnede gaf. En doordat deze kegels verwijden naarmate dat men verder van de bron komt, blijft er ook een dichte bedekking van de ruimte ver van de bron.

Hoewel dit een veel belovende aanpak lijkt, lijdt deze methode onder een aantal moeilijke problemen .

- Het is met kegelvormige bundels onmogelijk om een perfecte bedekking van de ruimte rondom de bron te krijgen, zonder dat men de bundels laat overlappen. Als een ontvanger in de doorsnede van twee bundels ligt, dan zullen beide bundels ook voor een geluidspad naar deze ontvanger zorgen. En dat is niet wenselijk. Men moet maatregelen nemen om te zorgen dat de ontvanger slechts één keer gedetecteerd wordt, of men moet gewichtsfactoren gebruiken om de dubbele detectie te compenseren.

- Doordat men zich op de straal in het midden van de bundel concentreert om de propagatie van de bundel te berekenen, brengt dit een aantal problemen zoals in figuur 2.5 wordt geïllustreerd. De bovenste bundel zal terecht reflecteren op de wand, maar zal hierbij de gereflecteerde bundel overschatten en ontvanger 1 ten onrechte detecteren. Tegelijkertijd had er een deel van de bundel ook voorbij de wand moeten trekken i.p.v. te reflecteren. Dit is hier niet gebeurd en zal dus een aantal ontvangers nooit detecteren terwijl ze wel voor de bron zichtbaar waren. De onderste bundel is het duale voorbeeld. Hier zou de bundel voor een deel moeten reflecteren maar zal dit nooit doen omdat de as van de bundel voorbij de wand schiet. Hierdoor zullen we ontvanger 2 missen en dus het aantal geluidspaden onderschatten. Tegelijkertijd



Figuur 2.5: detectiefouten bij conische bundeltrek

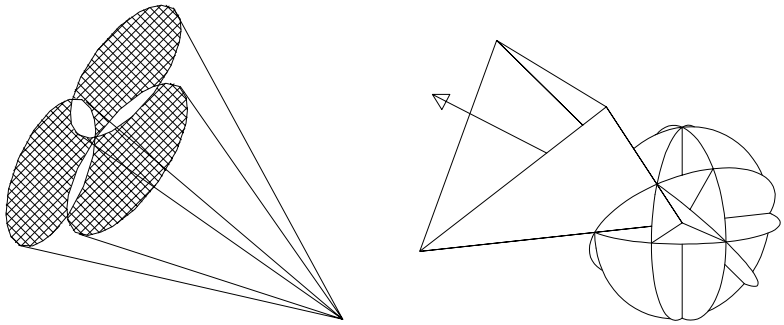
zullen een aantal ontvangers worden gedetecteerd die eigenlijk in de schaduw van de wand liggen. Dit probleem is helaas niet zo eenvoudig op te lossen. Het beste wat men kan doen is steeds trachten de bundels te overschatten en bij de detectie van ontvangers nog een extra zichtbaarheidstest doen. Zo zou de bovenste bundel ook nog rechtdoor kunnen laten propageren.

Piramidale bundeltrek

Om het probleem van de overlappende bundels aan de bron tegen te gaan, kan men de kegels vervangen door piramides met een driehoekige doorsnede. Dit noemt men piramidale bundeltrek. Verder zal men opnieuw zoals bij conische bundeltrek de propagatie van deze bundels berekenen aan de hand van de straal die door het midden van de bundel loopt, met dezelfde nadelen als gevolg. Om ook dit probleem op te lossen stappen we over op polygonale bundeltrek.

2.3.2 Polygonale bundeltrek

Dit is de eigenlijk methode die gebruikt is in `bass3`, en wordt ook wel eens *adaptieve bundeltrek* genoemd. Het is gebaseerd op piramidale bundeltrek, maar nu zal men piramides met willekeurige convexe polygonale doorsneden toelaten. Dit in tegenstelling tot bovenstaande waar enkel driehoeken werden toegelaten. Dit houdt in dat men bij reflectie en



Figuur 2.6: in tegenstelling tot conische bundels omsluiten piramidale bundels de bron zonder overlapping

occlusie een nieuwe bundel kan maken die het resultaat perfect omsluit. Met andere woorden, men past de bundel aan de geometrie aan, vandaar het *"adaptieve"*. Het genereren van geluidspaden met polygonale bundeltrek bestaat uit twee fasen. De eerste fase is de eigenlijk bundeltrek en zal vanuit de bron alle zichtbare ontvangers bepalen. De tweede fase bepaalt de eigenlijke geluidspaden met behulp van het spiegelbronmodel.

Het grote voordeel ten opzichte van de oorspronkelijke spiegelbronmethode is dat we nu op voorhand weten welke ontvangers zichtbaar zijn, en dat het spiegelbronmodel voor deze ontvangers steeds een geldig pad zal opleveren. Hierdoor vermijden we het doorrekenen van enorme hoeveelheden virtuele bronnen die toch geen geldig pad kunnen opleveren. Voor een balkvormige ruimte blijkt de spiegelbronmethode toch efficiënter te zijn, omdat de kost van de bundeltrek niet opweegt tegen het testen van virtuele bronnen dat in deze omgeving zeer snel kan gebeuren.

Tegenover stralentrek heeft bundeltrek als grootste voordeel het uitbuiten van ruimtelijke coherentie. Iedere bundel kan worden gezien als een unie van een oneindig aantal stralen. Waar we bij stralentrek vele stralen moeten doorrekenen, volstaat het bij bundeltrek om een aanzienlijk minder aantal bundels door te reken. En tegelijkertijd wordt de ruimtelijke discretisatie vermeden, eigen aan stralentrek.

Met deze voordelen komt wel een verlies aan eenvoud van het algoritme, wat als een nadeel kan worden gezien. De geometrische operaties om een bundel door een 3D omgeving te trekken zijn een stuk minder eenvoudig dan bijvoorbeeld bij stralentrek. Een goede modellering van de omgeving is hiervoor vereist. Zoals zal worden besproken in hoofdstuk 3, zullen convexe cellen het eenvoudigste bundeltrek algoritme toelaten. Bovendien is bundeltrek niet zonder meer geschikt voor omgevingen met niet-polygonale wanden, maar dat wordt echter niet in dit eindwerk behandeld.

We bespreken nu de 3D polygonale bundeltrek in detail. Voor de illustraties zullen we in twee dimensies werken, omdat dit eenvoudigere tekeningen toelaat.

Voorstelling van een bundel

Zoals hierboven vermeld, bestaat een bundel bij polygonale bundeltrek uit een piramide waarvan de doorsnede niet noodzakelijk driehoekig hoeft te zijn. De top van deze piramide stelt de (virtuele) bron voor. We eisen wel nog steeds dat de piramide convex is. Daardoor kunnen we zulke piramides het best voorstellen aan de hand van een aantal *snijvlakken* Σ_i met hun cartesiaanse vergelijking.

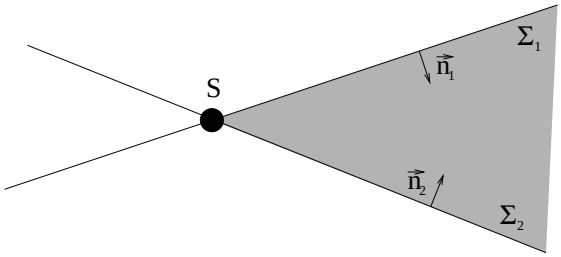
$$\Sigma_i \leftrightarrow \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i = 0 \tag{2.1}$$

Hierbij is $\overrightarrow{n_i}$ een normaal vector van Σ_i, O de oorsprong van de ruimte, P een willekeurig punt en is d_i representatief voor de afstand van het vlak tot de oorsprong. Let hierbij wel op dat als $\overrightarrow{n_i}$ niet genormaliseerd is, deze afstand niet $|d_i|$ is maar $\frac{|d_i|}{\|\overrightarrow{n_i}\|}$.

Met deze vergelijking kunnen we namelijk een voor- en achterkant aan een vlak toekennen. Elk punt P van de ruimte kunnen we in één van de volgende drie categorieën plaatsen:

$$\begin{cases} \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i > 0 & \textit{voor het vlak} \\ \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i = 0 & \textit{op het vlak} \\ \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i < 0 & \textit{achter het vlak} \end{cases} \tag{2.2}$$

Deze snijvlakken zullen de piramide begrenzen. We zorgen ervoor dat elk punt in de piramide zich voor elk snijvlak bevindt. Dit doen we door de normaalvectoren naar binnen te richten.



Figuur 2.7: een bundel begrensd door snijvlakken

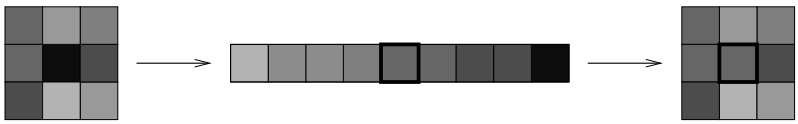
Dit leidt tot een zeer eenvoudige *punt-in-bundel* test. Deze test controleert of een punt zich binnen de bundel bevindt door te testen of het voor ieder snijvlak ligt. Inderdaad, voor elk ander punt zal er tenminste één snijvlak zijn die een negatief resultaat oplevert.

Bemerk dat we de punten op de mantel ook tot de piramide rekenen. Als er naast deze bundel ook nog een andere bundel ligt met dezelfde top en die een deel van de mantel deelt, dan betekent dit dat er punten zijn die tot beide bundels behoren. Dit kan leiden tot dubbele detectie van ontvangers, en blijkt niet zo eenvoudig te vermijden te zijn. De fouten die hierdoor ontstaan hebben echter wel de karakteristiek van impulsruis, waardoor

Algorithm 1 punt-in-bundel test

```
bool Beam3::contains(P)
{
    for (∀Σ_i)
    {
        if (n_i · OP + d_i < 0) return false;
    }
    return true;
}
```

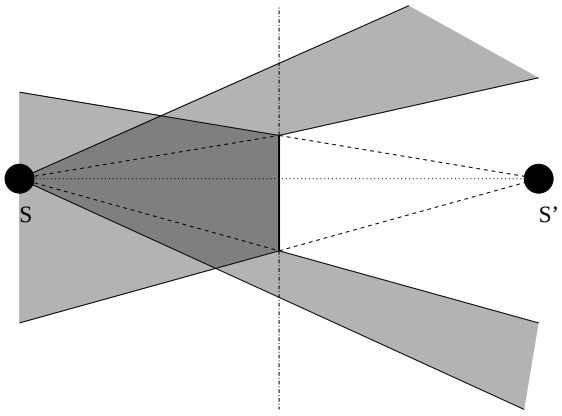
ze wel vrij goed te reduceren zijn door een mediaan filter toe te passen op het resultaat [25].



Figuur 2.8: een mediaan filter reduceert impulsruis veroorzaakt door dubbele detectie

Speculaire reflecties

We veronderstellen eerst dat een bundel op een kleine convexe polygonale wand valt, en dat deze wand volledig zichtbaar is. Voor deze wand moeten we nu een gereflecteerde bundel aanmaken. Dit kan eenvoudig door de bron S te spiegelen t.o.v. de wand, en vervolgens snijvlakken te trekken door de randen van de wand. Elke rand is een lijnstuk met twee eindpunten P_i en Q_i, en deze vormen met de virtuele bron S' 3 niet colineaire punten zodat hierdoor precies één snijvlak Σ'_i gaat.

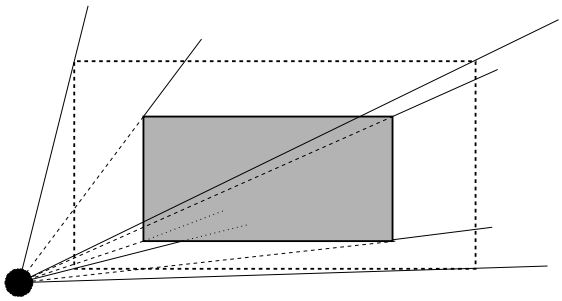


Figuur 2.9: reflectie van een bundel op een wand

Als de wand slechts gedeeltelijk in de bundel ligt, dan moet eerst het *zichtbare* gedeelte worden bepaald. Dit kan vrij eenvoudig met een standaard 3D Sutherland-Hodgeman

clipping algoritme [1, 4].

Schaduwzone



Figuur 2.10: concave zone voorbij wand

Er is een gedeelte van de bundel die voorbij de wand trekt zonder gereflecteerd te worden. Het probleem hier is dat de schaduwzone wel convex is (ze is namelijk het gespiegelde van de gereflecteerde bundel), maar wat overblijft van de bundel is meestal concaaf en kan zelfs niet samenhangend zijn. In 2D lijkt dit vrij eenvoudig op te lossen, maar in 3D is dit wel wat complexer. Een mogelijke aanpak hiervoor is het gebruik van *portals*. Wat portals zijn en hoe deze het probleem oplossen zal in hoofdstuk 3 zal worden besproken.

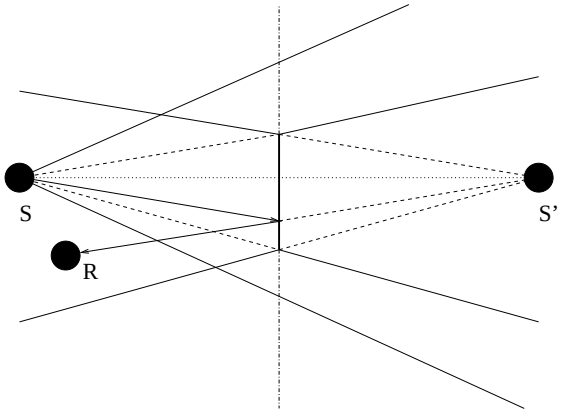
Padgeneratie

Eenmaal men een bundel heeft gevonden die vanaf de bron tot aan een ontvanger loopt, moet hieruit nog één enkele geluidstraal worden geconstrueerd. Dit proces noemt men de padgeneratie. Wanneer enkel speculaire reflectie werd gemodelleerd, blijkt dit zeer eenvoudig te zijn en is zeer verwant aan het spiegelbronmodel. Men moet namelijk enkel de bundel *ontplooien* over de verschillende reflecterende wanden om een geluidspad tussen bron en ontvanger te vinden. Dit werd in figuur 2.11 geïllustreerd voor een reflectie van 1ste orde, maar dit is triviaal uit te breiden tot een reflectie van hogere orde. Bemerk dat de geluidstraal zich steeds binnen de bundel zal bevinden, waardoor het zeker geldig is.

2.4 Uniforme diffractie theorie

Zoals eerder vermeld is het implementeren van diffractie één van de moeilijkheden bij geometrische akoestiek. Er bestaan een aantal benaderingen voor dit probleem in polygonale omgevingen zoals de Huyghens-Fresnel diffractie theorie. We gebruiken hier een andere benadering, de uniforme diffractie theorie.

Het principe van Huyghens [16] zegt dat elk punt van een golffront kan worden gezien als een nieuwe puntbron, en dat het voortgeplante golffront door de superpositie van al



Figuur 2.11: padgeneratie bij speculaire reflectie

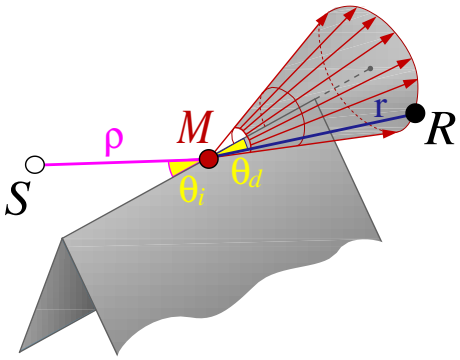
deze puntbronnen wordt gevormd. De uitbreiding van Fresnel hierop was het toevoegen van interferentie tussen deze puntbronnen, zodat diffractie kon worden behandeld. Hij verdeelde ook de ruimte tussen bron en ontvanger in concentrische ellipsoïden met frequentie afhankelijke stralen: de Fresnel ellipsoïden.

Gebaseerd op een discrete Huyghens interpretatie, kan men analytische uitdrukkingen vinden die filters geven in tijdsdomein voor diffractie over een aantal eindige ribben. De ribben worden verdeeld in een eindig aantal puntbronnen en de bijdragen van al deze bronnen worden gesommeerd. Deze modellen blijken zeer nauwkeurig te zijn voor diffractie van lage orde. Toch wensen wij dit principe niet toe te passen in deze bundeltrekker, omdat we er net alles aan willen doen om de ruimte niet te discretiseren. Een betere benadering voor onze doeleinden is de Uniforme Diffractie Theorie.

De *Uniforme Diffractie Theorie (UDT) (Eng.: Uniform Theory of Diffraction, UTD)* koppelt diffractie met de stralentheorie. Het behandelt een ribbe waaraan diffractie optreedt als secundaire bron van stralen die opnieuw kunnen interageren met de omgeving alvorens de ontvanger te bereiken. Het is een hoogfrequentie benadering, en is geldig voor oneindig lange diffractie ribben waarbij bron en ontvanger relatief ver van de ribbe zijn verwijderd. De UDT is reeds met succes toegepast in verschillende domeinen, en is voor akoestische golven en een klein aantal diffractie ribben gevalideerd tot een minimum van $150Hz$.

Een straal die op een ribbe invalt en daar diffractie veroorzaakt, zal volgens de UDT een kegel aan uitgaande stralen veroorzaken. De top van deze kegel is het snijpunt van de straal en de ribbe, de as van de kegel valt samen met de ribbe en de halve openingshoek θ_d is gelijk aan de hoek θ_i van de inkomende straal met de ribbe (figuur 2.12).

Bij een gegeven bron en ontvanger modelleert de UDT diffractie over een ribbe door een *enkelvoudige* straal die geattenueerd wordt met een complexe diffractie coëfficiënt. Deze straal snijdt de ribbe in één enkel punt en maakt daar een hoek. Hierbij wordt het principe van Fermat gehandhaafd, dat zegt dat bij een homogeen propagatie medium de geluidstraal het kortste pad van de bron tot de ontvanger volgt. Dit is ook de manier waarop het punt

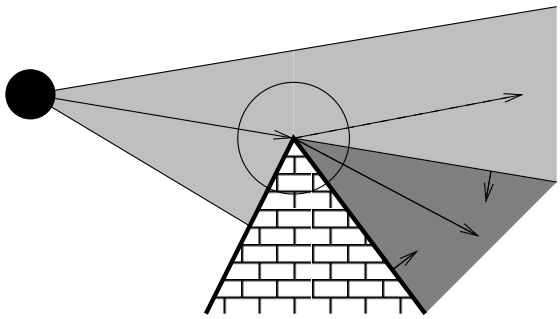


Figuur 2.12: uniforme diffractie theorie (uit [29])

M bepaald wordt.

2.4.1 Integratie in 3D polygonale bundeltrek

Het grootste probleem bij het integreren van UDT in het bundeltrek algoritme, is dat het enkel iets zegt over één enkele straal. In de bundeltrek kennen we deze straal echter nog niet. We kennen het punt M nog niet, het kan elk punt van de ribbe worden. Daarom bestaat de verzameling van potentiële stralen (de "bundel") niet uit één enkele kegel, maar uit een continue verzameling van kegels, elk volgens figuur 2.12 en met hun top op de ribbe. Deze verzameling wordt uiteindelijk begrensd door de twee kegels aan de uiteinden van de ribbe, en vormt nu de *diffractie-bundel* die voortspruit uit diffractie aan een ribbe. Helaas is deze bundel niet polygonaal, en is onze bundeltrekker niet zonder meer in staat deze te verwerken.



Figuur 2.13: schatting van diffractie-bundel

Bovendien bestrijkt deze diffractie-bundel de volle 360° rond de ribbe. We stellen echter voorop dat we enkel diffractie willen doorrekenen in de schaduwzone van het obstakel, zoals geïllustreerd in figuur 2.13. De diffractie ribbe staat loodrecht op het blad. De bundel (lichtgrijs) die op het obstakel invalt wordt voor een stuk tegengehouden door het obstakel, maar kan ook er voor een stuk voorbij trekken. Aldus ontstaat een schaduwzone

(donkergrijs) voorbij het obstakel. De diffractie-bundel die ontstaat aan de ribbe is symbolisch voorgesteld met een cirkel. Ze bestrijkt de volle 360°, maar ze is enkel van belang in de schaduwzone omdat elders de diffractie te verwaarlozen is ten opzichte van de rechtstreekse geluidspaden. Bijgevolg beperken we de diffractie-bundel tot deze schaduwzone door het invoegen van twee extra snijvlakken (aangeduid door hun normaalvector).

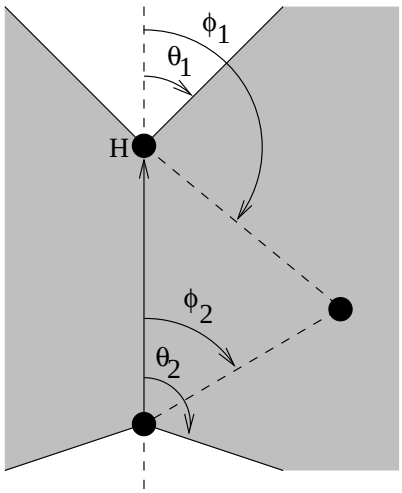
De grootste uitdaging bij het implementeren van diffractie met 3D polygonale bundeltrek ligt in het zichtbaarheidsprobleem, omdat de bundel niet meer polygonaal is. De punt-in-bundel test is nog vrij eenvoudig, men kan gemakkelijk testen of een punt al dan niet in een kegel ligt. De echte problemen ontstaan als we deze bundel verder laten propageren. Wat gebeurt er bij de intersectie van deze bundel met een wand? Hoe moet het clipping algoritme er uit zien dat bepaalt welk deel van de wand zichtbaar is? Omdat de bundel afgebakend is door kegels krijgen we kegelsneden als doorsneden. Deze zijn verre van polygonaal en we wensen deze dan ook te vermijden.

In plaats daarvan zullen we dit zichtbare gebied overschatten en later tijdens de padgeneratie eventuele ongeldige paden verwerpen. We zeggen dat we een conservatieve schatting van het zichtbare gebied maken [29]. Dit zal ons meer bundels opleveren en zal ontvangers ten onrechte als zichtbaar bestempelen, maar de padgeneratie zal nog steeds in staat zijn om ongeldige ontvangers te verwerpen als het geen geldig geluidspad kan creëren. We maken de conservatieve schatting door de kegels niet mee te rekenen in de bundeltrek. Enkel de vlakken die de bundel beperken tot de schaduwzone blijven over. Zo loopt de bundel oneindig door verticaal op het blad.

Om de padgeneratie toch niet al te zeer te overbelasten met ongeldige ontvangers (padgeneratie met diffractie is immers vrij kostelijk), zullen we de kegels wel nog meerekenen in de punt-in-bundel test (algoritme 2). Hiervoor slaan we twee hoeken θ_1 en θ_2 op die ons de halve openingshoeken van beide begrenzende kegels geven (figuur 2.14).

Algorithm 2 punt-in-bundel test voor diffractie bundels

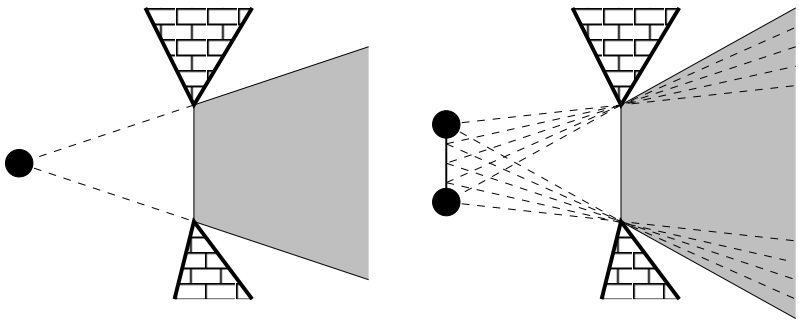
```
bool Beam3::contains(P)
{
    for (∀Σ_i)
    {
        if (n_i · OP + d_i < 0) return false;
    }

    if (diffracted beam)
    {
        if (φ_1 < θ_1 || φ_2 > θ_2) return false;
    }

    return true;
}
```



Figuur 2.14: punt-in-bundel test voor diffractie-bundels

2.4.2 Bundels met lijnbronnen

De gediffracteerde bundel heeft nu weer een polygonale begrenzing, wat de verwerking eenvoudig maakt. Voor één ding moeten we echter uitkijken: deze bundel heeft geen punt meer als top, maar een volledig *lijnstuk*: de diffractie ribbe. We zeggen dat de bundel een lijnbron als bron heeft. Dit heeft ernstige gevolgen op het zichtbaarheidsprobleem. We zullen dit illustreren in 2D, maar de redenering is evenzeer in 3D geldig. Stel dat vanuit een lijnbron het zichtbare gebied bepaald moet worden voorbij de opening tussen twee obstakels. Bij een puntbron is dit eenvoudig, men trekt lijnen door de bron en elk hoekpunt (figuur 2.15, links). Voor een lijnbron moet met dit eigenlijk voor elk punt van de lijnbron doen, en het uiteindelijk zichtbare gebied is de unie van alle deze resultaten. Dit wordt geïllustreerd in figuur 2.15, rechts.

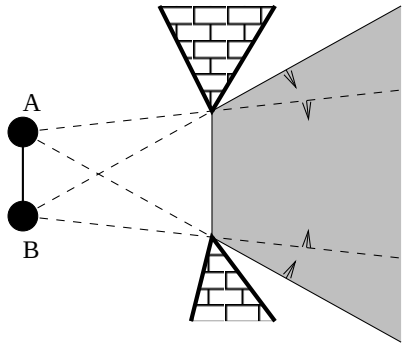


Figuur 2.15: zichtbaar gebied van een puntbron (links) en een lijnbron (rechts)

Dit doen voor een oneindig aantal puntbronnen op één lijnbron is uiteraard niet gewenst. We kunnen het resultaat echt ook direct bepalen, door op te merken dat het voldoende is om de unie te maken van het zichtbare gebied voor elk van de eindpunten, en dat zijn er slechts twee. Bovendien, kunnen we deze unie op een zeer eenvoudige manier bepalen.

Dit is geïllustreerd in figuur 2.16. Bemerk dat we normaalvectoren aan de lijnen hebben toegevoegd, die de voor- en achterkant ervan bepalen, en zodoende ook de binnenkant van het zichtbaarheidsgebied bepalen.

We bepalen eerst de begrenzing voor het bovenste obstakel, daarna voor het onderste. Eerst bepalen we de lijn door A en ribbe van het obstakel. Dit levert ons de streepjeslijn op. Daarna controleren we of B voor of achter deze lijn ligt. Ligt deze ervoor (zoals in dit geval), dan wordt de lijn vervangen door deze door B en de ribbe. Deze laatste lijn wordt dan de begrenzing voor het zichtbare gebied. Het andere verhaal vindt men bij het onderste obstakel. Opnieuw proberen we eerste de lijn door A en de ribbe, en testen we of B achter deze lijn ligt. Dit maal ligt ze er ook achter, en dat laat ons toe om te besluiten dat de lijn door A de correcte begrenzing is.



Figuur 2.16: bepalen van zichtbaarheidsgebied van een lijnbron

Hetzelfde verhaal is ook geldig bij reflectie. De opening is dan een reflecterende wand, en de lijnbron is dan een virtuele bron, deze is het gespiegelde van de originele lijnbron.

Hoofdstuk 3

Voorstelling van de omgeving: World3

De afgesloten ruimte waarin de simulatie zal plaatsvinden, noemen we de wereld. Specifiek is dit hier een stadsomgeving. Om simulaties in deze wereld uit te voeren, moet een manier gevonden worden om de wereld in het computergeheugen op te slaan. Er is nood aan een datastructuur. Er zijn hiervoor tal van mogelijkheden met elk een eindeloos aantal varianten.

Met het oog op onze toepassing worden er in dit hoofdstuk een aantal besproken met hun voor- en nadelen. Eerst komen eenvoudige datastructuren aan bod waarin enkel de obstakels worden opgeslagen, en daarna volgen er meer geavanceerde datastructuren die de omgeving opsplitsen in convexe cellen. Het concept van convexe cellen wordt nader behandeld, en tenslotte wordt de *winged-pair* datastructuur `World3` besproken die werd gekozen voor `bass3`. Hierbij gaat het steeds om de opdeling van een 3D omgeving, tenzij anders vermeld.

3.1 Overzicht van verschillende mogelijkheden

3.1.1 Polygonsoup

Een polygonsoup (letterlijk vertaald *polygonensoep*) bestaat uit een ongeordende lijst van losse polygonen. Deze polygonen stellen de wanden van de obstakels in de omgeving voor. Ze delen geen gemeenschappelijke data zoals ribben en hoekpunten. Het grootste voordeel van deze datastructuur is de erg eenvoudige opbouw, maar dit is ook tevens het grootste nadeel. Een polygonsoup is zeer snel aan te maken: we hoeven enkel van elke vaste wand in de wereld een polygoon te maken, en die toe te voegen aan de lijst. Bestaat de wereld uit twee kubussen als obstakels, dan bestaat deze lijst uit twee maal zes vierkanten. De beperkingen worden echter al snel zichtbaar, zelfs als zeer eenvoudige vraagstukken worden opgelost:

- Om te weten of een lijn al dan niet een polygoon snijdt, en welke polygoon dit is, dan moeten alle polygonen worden overlopen en gecontroleerd of die door de lijn wordt doorsneden. Bovendien kan een lijn meerdere polygonen snijden, zodat ook nog moet worden bepaald welke van deze van belang is.
- Om polygonen te vinden die een bepaalde ribbe delen met een gegeven polygoon, moeten alle polygonen worden verlopen worden tot het juiste antwoord is gevonden.

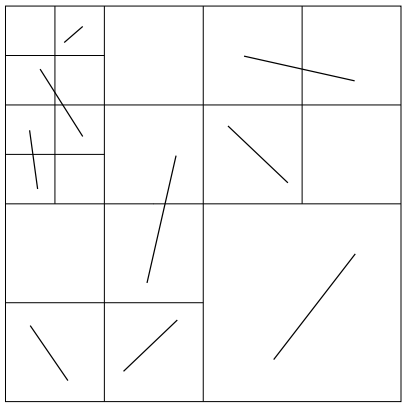
Het is duidelijk dat een polygonsoup een zeer rekenintensieve datastructuur is, die bovendien complexe algoritmen vereist om op betrekkelijk eenvoudige vraagstukken een antwoord te bieden. Maar bovenal ontstaat er een probleem qua convexiteit bij het eigenlijke bundeltrekken. Als een bundel op een wand invalt, dan zal – zoals reeds in 2.3.2 besproken – een deel van deze bundel op de wand reflecteren, maar zal er ook een stuk voorbij de wand trekken. Dit tweede stuk is gelijk aan de originele bundel min de schaduwzone, en is typisch concaaf van vorm. Concave bundels moeten absoluut worden vermeden, dus moet deze bundel worden opgesplitst in convexe delen. Algoritmes hiervoor kunnen worden geïmplementeerd, maar dit zal slechts één van de zovele zware operaties zijn in het geval van een polygonsoup. We zoeken dus een beter alternatief.

Men zou kunnen de convexe celstructuur – zoals zal worden uitgelegd in sectie 3.1.4 – implementeren met een polygonsoup. Dit zou het convexiteitsprobleem oplossen, maar het zou de zoekoperaties enkel nog *meer* belasten. De relatie tussen de verschillende polygonen kan immers niet worden gevat in worden gevat in een polygonsoup. Het kan immers enkel afzonderlijke polygonen bewaren. Zo gaat de informatie over de afzonderlijke cellen verloren en blijven enkel de polygonen over. Omdat er voor een convexe celstructuur meer polygonen nodig zijn dan de voor oorspronkelijke obstakels, zullen dus bij elke zoekopdracht meer polygonen moeten worden doorzocht. Gelukkig bestaan er voor convexe celstructuren betere methodes om deze op te slaan, zoals zal worden uitgelegd in sectie 3.3.

3.1.2 Quadtree/Octree

Bij een quadtree wordt het horizontale vlak in vier regio's opgedeeld. En elke regio kan recursief nogmaals worden opgedeeld in vier nieuwe regio's. Op die manier kan men trachten om elk detail (in dit geval één enkele polygoon die een wand voorstelt) in een afzonderlijke regio te plaatsen. Meestal wordt niet één enkel detail in een subregio geplaatst, maar een maximum aantal. Zo zal elke subregio z'n eigen polygonsoup bevatten, m.a.w. de polygonsoup van hierboven wordt opgesplitst in afzonderlijke kleinere polygonsoups.

Deze methode biedt een aantal voordelen: als gezocht wordt welke polygonen een lijn snijden, dan moeten niet meer alle polygonen worden overlopen, maar enkel die welke behoren tot de subregio's waar de lijn door gaat.



Figuur 3.1: quadtree: iedere kwadrant wordt opgedeeld totdat er niet meer dan n obstakels per kwadrant zijn. Hier $n = 1$.

Een vraag die moet worden opgelost bij het opbouwen van een quadtree is hoe diep het horizontaal vlak moet worden opgesplitst. Een eerste opmerking hierbij is dat niet alle kwadranten even diep moeten worden opsplitst. Men kan bijvoorbeeld het eerste kwadrant helemaal niet verder opsplitsen terwijl men het tweede kwadrant 10 niveaus diep opsplitst.

Een typische opbouw gaat als volgt: het horizontale vlak wordt in vier kwadranten opgesplitst en de invoerdata wordt over deze vier regio's verdeeld. Elke regio die meer dan het maximum toegelaten aantal details bevat, wordt opnieuw opsplitst in vier subregio's en de details worden verdeeld over deze subregio's. Daarna zal dit proces voor elke subregio dit proces recursief worden herhaald, tot elke subregio niet meer dan het maximum toegelaten aantal details bevat, of een maximaal niveau van opsplitsing is bereikt (men kan bijvoorbeeld besluiten om niet meer dan 16 niveaus diep op te splitsen).

Een typisch probleem dat ontstaat bij quadtrees is dat men veelal een detail niet kan toekennen aan één enkele subregio, maar dat het zich uitstrekt over twee of meer subregio's. Hiervoor bestaan twee oplossingen:

- Het detail opsplitsen in delen die wel in één enkele subregio passen. Dit is echter niet altijd mogelijk of gewenst.
- Het detail simpelweg toekennen aan alle subregio's waartoe het behoort. Daarbij heeft men het voordeel dat het detail niet hoeft te worden opgesplitst, maar er ontstaat wel het gevaar dat men bij het doorlopen van de quadtree hetzelfde detail meermaals tegenkomt. Voor sommige algoritmes kan dit fataal zijn. Voor die algoritmes moeten extra voorzorgen worden genomen.

Een octree is gelijkaardig aan een quadtree, maar hierbij wordt de ruimte ook verticaal opgesplitst. Zo zal elke regio worden opgesplitst in acht octanten. Bij een quadtree wordt er geen rekening gehouden met de z-informatie.

Quad- en octrees hebben hun nut bewezen voor open omgevingen zoals open landschappen. In de grafische wereld treft men deze dan ook dikwijls aan in zogenaamde *outdoor engines*. Voor een stadsomgeving zou dit ook een goede kandidaat zijn, ware het niet dat een quadtree aan hetzelfde probleem lijdt als een polygonsoup i.v.m. de convexiteit bij het bundeltrekken. Immers, quadtrees zijn niet meer dan een verzameling op slimme manier geordende polygonsoups. Tenzij we hier toch convexe cellen zouden gebruiken, maar daarbij geldt hetzelfde nadelen als werd vermeld bij de polygonsoup: een quad- of octree slaat afzonderlijke polygonen op en is niet in staat om de cellen zelf te vatten.

3.1.3 BSP-tree

BSP-tree staat voor *"Binary Space Partitioning tree" (letterlijk vertaald: binaire ruimte verdeling boom)*. Bij een BSP-tree wordt de ruimte opnieuw recursief verdeeld in regio's, en zoals de naam al laat vermoeden wordt hierbij elke regio verdeeld in precies twee subregio's. Op het eerste zicht lijkt dit een beperking ten opzichte van een quad- en octree. Toch is dit niet zo omdat me hier voor de opsplitsing een arbitrair vlak kan gebruiken (of een rechte in het 2D geval), terwijl bij quad- en octrees de opsplitsing steeds volgens de assen van de ruimte gebeurde.

Als de details bestaan uit afzonderlijke polygonen (zoals het geval is voor de bundeltrekker), dan wordt bij een BSP-tree veelal het draagvlak van deze polygoon als splitsingsvlak gebruikt. Elk knooppunt in de BSP-tree is dan een vlak dat een regio opsplitst in twee subregio's. Het knooppunt heeft dan twee zonen (een linker- en een rechterzoon) die elke subregio opnieuw opsplitsen.

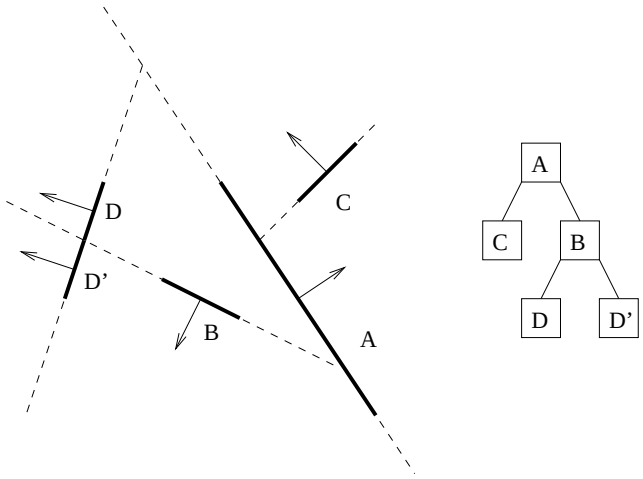
Een BSP-tree heeft hetzelfde nadeel als quad- en octrees. Het is namelijk wederom mogelijk dat een polygon niet precies in één regio past. Men kan opnieuw kiezen om op te splitsen, of om die ene polygon toe te kennen aan meerdere regio's. Een bijkomend nadeel is dat het zeer lastig is om een dynamische omgeving bij te houden in een BSP-tree. Dit is een omgeving waarvan de polygonen wijzigen in de tijd. De reden is dat één wijziging veelal invloed heeft op een groot stuk van de BSP-tree. Als bijvoorbeeld de polygoon die in de wortel zit verplaatst wordt, dan zal zijn draagvlak de ruimte iets anders opsplitsen, en moet de ganse boom herzien worden. Gelukkig is er in `bass3` geen sprake van dynamische omgevingen.

Men onderscheidt twee manieren op een BSP-tree te gebruiken: *planar* en *leafy*. De eerste wordt veel gebruikt voor de visulatiesatie van obstakels, de tweede zal gebruik maken van convexe cellen.

Planar BSP-tree

De BSP-tree bestaat uit de draagvlakken van de polygonen en deze polygonen worden met de draagvlakken opgeslagen in de knooppunten. De informatie zit met andere woorden

opgeslagen in de splitsingsvlakken, vandaar de naam planar BSP-tree. Aan een vlak wordt typische een voor- en achterkant toegekend volgens de normaalvector waarmee het vlak wordt voorgesteld. De kant waarnaar de normaalvector wijst wordt doorgaans bestempeld als voorkant. Als men dan steeds de regio aan de voorkant toekent aan de linkerzoon, en de regio aan de achterkant aan de rechterzoon, dan kan men de ganse ruimte systematisch opsplitsen. In figuur 3.2 is geïllustreerd hoe zo'n BSP-tree kan worden opgebouwd. De wortel A splitst de ganse ruimte op in twee helften. Daarna worden B, C en D toegevoegd in de boom. Het splitsingsvlak van B doorsnijdt D. We kunnen dan zoals geïllustreerd D opsplitsen in D en D', maar we hadden ook D twee keer kunnen invoegen, analoog als bij een quad- of octree.



Figuur 3.2: de opbouw van een BSP-tree: obstakels A tot D werden achtereenvolgens toegevoegd, A werd hierbij de wortel van de boom en D werd door B opgesplitst in D en D'

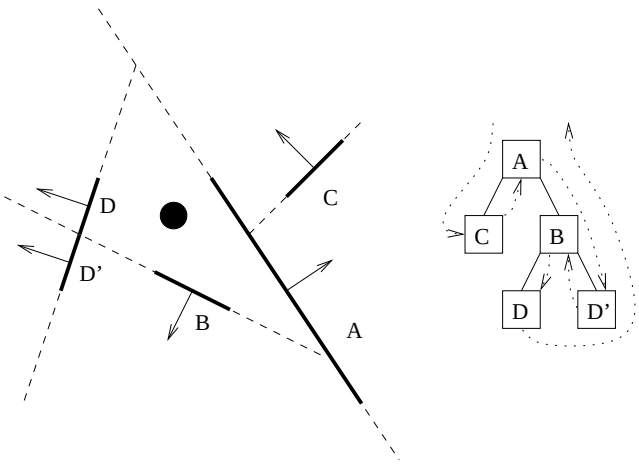
Eens de structuur is opgebouwd, kunnen we de boom ook systematisch doorlopen, bijvoorbeeld *front-to-back walk (letterlijk: doorloop van achter naar voor).* Wat wordt hiermee bedoeld? Als men in de grafische wereld polygonen op het scherm wil afbeelden, dan wil men bijvoorbeeld het zogenaamde *schildersalgoritme (Eng.: painters algorithm)* gebruiken [1]. Hierbij beeld men eerst de polygonen af die het verst verwijderd zijn, en daarna de polygonen die dichter bij de waarnemer liggen. Op die manier worden eventuele niet zichtbare delen van de verste polygonen overschreven zoals bij een schilderij, en krijgt men de gewenste indruk van de omgeving. Dit overschrijven van polygonen (of eigenlijk de pixels waarmee ze op het scherm worden afgebeeld) noemt met *overdraw.*

Bij een polygonensoep betekent dit voor iedere afbeelding een intensieve sortering van de polygonen, maar bij een BSP-tree is dit vraagstuk snel opgelost. Men begint aan de wortel en bepaalt of men zich voor of achter het eerste vlak bevindt. Bevindt men zich voor dit vlak, dan weet men dat de rechterzoon verder van de waarnemer verwijderd is dan de wortel. Men tekent dan eerst de rechterzoon, dan de wortel, en dan de linkerzoon (want die bevindt zich dichter bij de waarnemer dan de wortel). Bevindt men zich achter het

wortelvlak, dan tekent men eerste de linkerzoon, dan de wortel, en dan de rechterzoon. En voert men dit proces vervolgens recursief uit voor beide zonen, dan zal men alle polygonen in de BSP-tree perfect *van achter naar voor* doorlopen.

Algorithm 3 doorlopen van een planar BSP-tree

```
void renderBackToFront(BSPnode* a_node)
{
    if (a_node != 0)
    {
        if (a_node->plane()->inFront(position))
        {
            // position is in front of plane.
            renderBackToFront(a_node->right());
            draw(a_node->polygon());
            renderBackToFront(a_node->left());
        }
        else
        {
            // position is in back of plane.
            renderBackToFront(a_node->left());
            draw(a_node->polygon());
            renderBackToFront(a_node->right());
        }
    }
}
```



Figuur 3.3: de BSP-tree uit figuur 3.2 wordt van-achter-naar-voor doorlopen voor een waarnemer in het midden van de ruimte

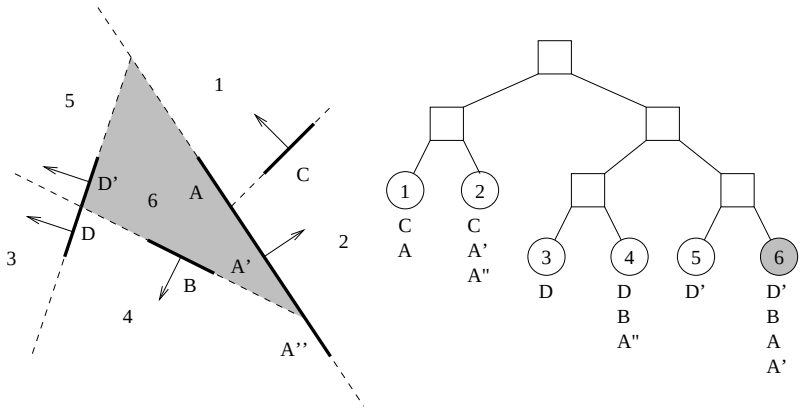
Deze techniek heeft z'n nut reeds vele malen bewezen [1]. Dit is vooral te danken aan het feit dat het schildersalgoritme met een BSP-tree in lineaire tijd kan gebeuren: elk knooppunt wordt precies één maal bezocht. Er is tegenwoordig wel een verschuiving naar een *front-to-back* walk om de overdraw te beperken. Dit kan om dat de huidige grafische

kaarten zijn uitgerust met een *z-buffer* dat bijhoudt hoe *diep* (hoe ver) een pixel is gelegen. Vooraleer een pixel wordt beschreven wordt dan gecontroleerd of die al dan niet voor de huidige pixel ligt. Als de nieuwe pixel achter de huidige ligt, dan wordt deze niet naar het scherm geschreven. Op deze manier kan men het zichtbaarheidsprobleem perfect oplossen, onafhankelijk van de volgorde waarmee te polygonen worden getekend. Als je echter eerst de meest nabije polygonen tekent, dan zullen de pixels minimaal worden overschreven wat heel wat bandbreedte bespaart. Op moderne systemen is de bandbreedte dikwijls een beperkende factor.

Alhoewel dit een interessante methode lijkt, is deze toch niet geschikt voor ons doel. Het schildersalgoritme is uitstekend geschikt om een omgeving af te beelden met pixels op een raster (*rasterizing*), maar om bundels te creëren is deze veel minder geschikt. We moeten namelijk in staat zijn om stukken uit bestaande bundels te snijden (overschilderen), en dat leidt meestal tot concave bundels. Deze moeten dan terug tot convexe bundels worden herleid, wat niet zo eenvoudig is. Er is echter nog een andere manier om BSP-trees te gebruiken waarbij convexe cellen worden gebruikt. En deze keer zal de BSP-tree wel in staat zijn om de cellen zelf te coderen. We noemen deze manier een *leafy BSP-tree*.

Leafy BSP-tree

Een BSP-tree deelt de ruimte op in verschillende regio's. We kunnen deze regio's beschouwen als de blaadjes van de BSP-tree. Immers, elke eindtak (een tak op het uiterste van de boom die zelf geen zonen meer heeft), deelt een regio op in twee subregio's die niet meer verder zullen worden opgesplitst. Deze twee subregio's hangen dan als twee blaadjes aan deze tak, en alle blaadjes van alle eindtakken vullen de volledige ruimte. Bij een leafy BSP-tree wordt de informatie (onze polygonen) gestockeerd in die blaadjes. Dit in tegenstelling tot een planar BSP-tree waarbij de informatie wordt gestockeerd in splitsingsvlakken, die tevens ook de draagvlakken van die polygonen zijn.

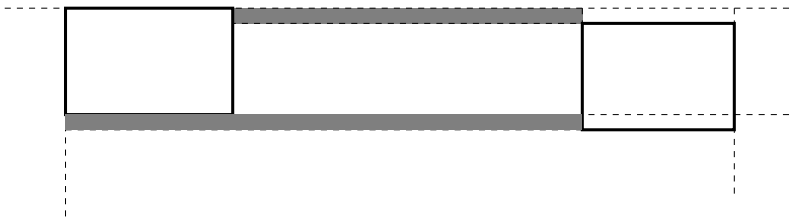


Figuur 3.4: bij een leafy BSP-tree worden de polygonen opgeslagen per subregio, bemerk dat deze (meestal) behoren tot twee subregio's

Hier moeten we wel alle polygonen opsplitsen zodat deze tot slechts twee subregio's behoort: één aan beide zijden. Bij een planar BSP-tree bespraken we reeds dat een polygon kan worden opgesplitst door een splitsingsvlak dat hoger in de hiërarchie is gelegen. Zo werd D opgesplitst door het draagvlak van B, en elk deel werd aan één van B gestockeerd. Bij een leafy BSP-tree kunnen polygonen ook opgesplitst worden door splitsingsvlakken die lager in de hiërarchie zitten. Zo zal A moeten worden opgesplitst in drie stukken volgens de draagvlakken van B en C. Inderdaad, elk van deze stukken ligt tussen precies twee subregio's, terwijl A op zijn geheel aan vier subregio's grenst.

Op deze manier komen we tot het concept van convexe cellen. Inderdaad, de subregio's zijn steeds convex, en het zijn bovendien veelvlakken. Het zijn echter nog geen gesloten veelvlakken. Inderdaad, in figuur 3.4 kan men zien dat in cel 6 naast de oorspronkelijke polygonen (dikke lijnen) er nog een aantal openingen overblijven. Deze zal men dan sluiten door *doorzichtige* polygonen: portals (portals worden later uitgelegd en behandeld).

Typisch zal men een leafy BSP-tree precies gebruiken om een convexe celstructuur aan te maken . Dit heeft echter een groot nadeel. Er kunnen namelijk zeer smalle en lange cellen ontstaan, zoals in figuur 3.5 wordt geïllustreerd. We hebben voor de duidelijkheid sterk overdreven, maar het idee is dat de rechthoeken eigenlijk even hoog hadden moeten liggen, maar wegens numeriek fouten de een net iets hoger ligt dan de andere. Bijgevolg zullen de boven en onderzijden niet perfect in elkaars verlengde liggen, en zullen hun splitsingsvlakken zeer fijne en lange cellen veroorzaken. Deze cellen kunnen dan leiden tot zeer fijne bundels die sterk onder numeriek precisiefouten kunnen leiden. Het is een situatie die we liever vermijden. Een oplossing hiervoor is het trianguleren van de ruimte, zoals zal worden voorgesteld in hoofdstuk 4.



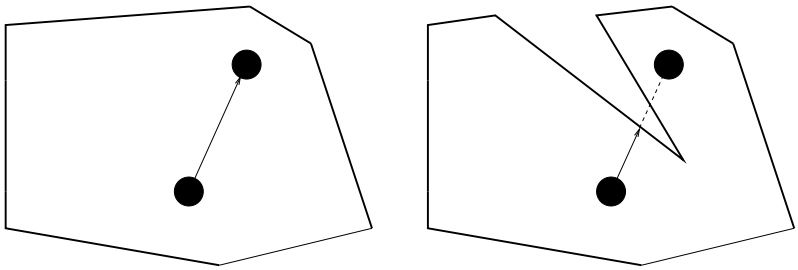
Figuur 3.5: 2D BSP-tree kan leiden tot zeer fijne en lange cellen

Het grote voordeel van deze convexe cellen is dat ze het probleem met concave bundels zullen oplossen. We zullen daarom de eigenschappen van een convexe celstructuur wat nader bij bekijken.

3.1.4 Convexe celstructuur

Bij een convexe celstructuur wordt de wereld opgebouwd uit een aaneenschakeling van convexe veelvlakken. Eén convex veelvlak is één cel. Men kan celstructuren ook opbouwen uit concave cellen, maar dan verliest men veel van de eenvoud. Zo kan men bij een convexe

cel een lijn kan trekken tussen twee punten in de cel, zonder dat die lijn de wand van de cel snijdt en gedeeltelijk buiten de cel ligt. Dit zal zeer belangrijk blijken voor het zichtbaarheidsprobleem. Stel een convexe cel voor, en plaats je als waarnemer in die cel. Nu kan je elk punt binnen deze cel *zien*, alsook elke wand (polygoon) van de cel. Dit is niet langer het geval bij concave cellen, waar bepaalde wanden stukken van de cel aan het zicht kunnen onttrekken. In wat nu volgt, zullen we dus een cel steeds als een convex veelvlak beschouwen, tenzij expliciet anders vermeld.



Figuur 3.6: convexe vs. concave cellen

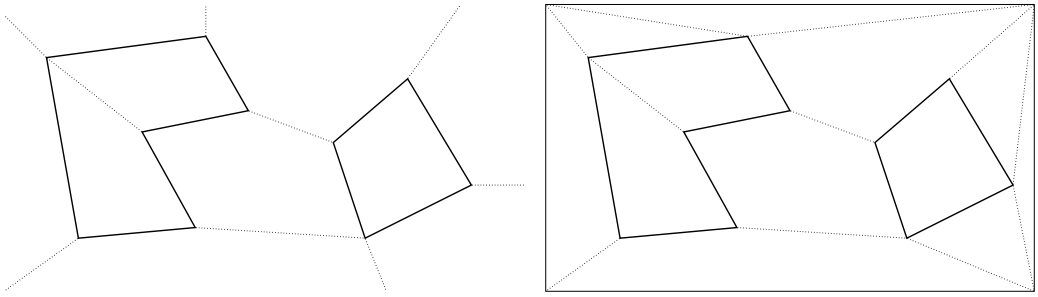
De volledige ruimte kan nu worden gezien als een unie van cellen. Deze cellen liggen tegen elkaar aan, en twee buurcellen delen precies één wand. Deze wanden zijn als het ware membranen die de verschillende cellen scheiden, en we noemen ze *portal*s. We kunnen namelijk door een portal een buurcel *binnenkijken* of *binnengaan*. De meeste wanden zijn portals, en zeker alle wanden van cellen die volledig omringd zijn door andere cellen. Een geval dat we apart moeten bekijken is wat er gebeurd met cellen die aan de rand van de wereld liggen.

Volle en open portals

Vooraleer we verder gaan moeten we een belangrijke opmerking maken over het begrip *portal*. Doorgaans worden portals als doorzichtig beschouwd. Zo zal men bijvoorbeeld bij een leafy BSP-tree enkel die polygonen die men invoegt om de cel te sluiten een portal noemen. Dit doen wij in `bass3` echter **niet**! Een portal is voor ons elke wand die gedeeld wordt door twee cellen, of deze nu doorzichtig is of niet. Om toch een onderscheid te maken zullen we wel spreken van volle en open portals. *Volle portals* zullen wanden zijn met akoestische eigenschappen, waarop bijvoorbeeld geluid kan reflecteren en dus ondoorzichtig zijn. Alle volle portals samen vormen onze omgeving zoals deze door de invoerdata is beschreven. In figuur 3.7 zijn dit de vette volle lijnen. Met *open portals* bedoelen we doorzichtige portals zonder akoestische eigenschappen. Ze zijn door een preprocessor ingevoegd om van de omgeving een convexe celstructuur te maken, en zijn dus volledig transparant voor het geluid. Open portals komen dus niet in de invoerdata voor. In figuur 3.7 zijn alle stippellijnen open portals.

Rand van de wereld

Tot nu toe beschouwden we steeds een wereld die oneindig groot is. Dit zou tot gevolg hebben dat een aantal cellen open zijn. We willen immers de wereld opbouwen uit een eindig aantal cellen. Het is echter niet altijd wenselijk om zulke cellen te hebben. Wat men dan meestal doet is de wereld begrenzen door middel van een AABB (een *axis-aligned bounding box*, een balk waarvan de assen parallel liggen aan de coördinaatassen, [23], sectie 4.5). De cellen aan de rand van de wereld, kunnen dan meteen ook worden afgesloten met een deel van de mantel van de AABB. Op deze manier komen we tot de conclusie dat er bij een convexe celstructuur er twee soorten wanden bestaan: portals die naar een buurcel leiden, en enkelzijdige wanden die slechts behoren tot één cel en aan de rand van de wereld liggen. We zullen later hiermee expliciet rekening moeten houden.



Figuur 3.7: celstructuur in oneindig en eindig grote wereld *(dikke volle lijnen = volle portals, fijne stippellijnen = open portals, volle fijne lijnen = enkelzijdige wanden)*

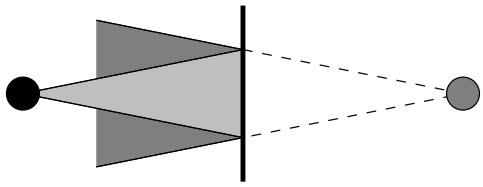
3.2 Bundeltrek met convexe cellen

3.2.1 Eigenschappen

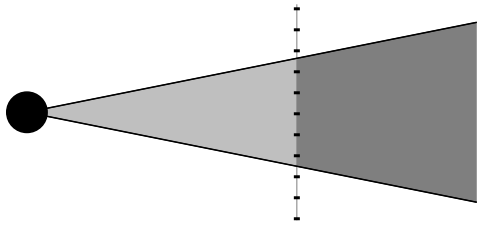
Het grootste voordeel van een convexe celstructuur is dat deze een zeer eenvoudig algoritme toelaat voor de bundeltrek. Zowel het zichtbaarheidsprobleem als het convexiteitsprobleem worden eenvoudig opgelost. Het eerste dankzij de convexe cellen, het tweede dankzij de open portals. We zullen dit nu trachten aan te tonen aan de hand van eenvoudige eigenschappen. Het is geen *"compacte set"* van eigenschappen maar is eerdere illustratief bedoeld.

Eigenschap 1: een bundel gaat nooit door een portal: Het belangrijkste om te onthouden is dat een bundel nooit *door* een wand gaat. Bij volle portals is dit triviaal. De bundel reflecteert namelijk op een volle portal, en deze reflectie bundel is een nieuwe bundel die de weg van het geluid verder zet (figuur 3.8). Men een open portal is dit minder evident, maar wel cruciaal. De originele bundel gaat steeds tot aan de open portal, en

de weg wordt langs de andere zijde verder gezet door een nieuwe bundel, een transmissie-bundel (figuur 3.9).



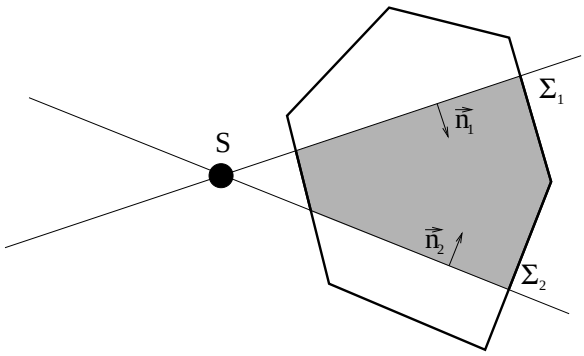
Figuur 3.8: inval van een bundel (lichtgrijs) op een volle portal (dikke lijn) veroorzaakt reflectie: de weg wordt dan verder gezet door een reflectie-bundel (donkergrijs) met de gespiegelde bron als top (eigenschap 1)



Figuur 3.9: Inval van een bundel (lichtgrijs) op een open portal (dikke stippellijn) veroorzaakt transmissie. De originele bundel stopt aan de portal, en de weg wordt verder gezet door een transmissie-bundel (donkergrijs). (eigenschap 1).

Eigenschap 2: een bundel is steeds binnen één cel gelegen: Inderdaad, als een bundel over meerdere cellen zou zijn verspreid, dan zou het ten minste één portal moeten hebben doorsneden. En dat zou in strijd zijn met eigenschap 1. Dit genoodzaakt ons om de definitie van de begrenzing van een bundel te herzien. Een bundel wordt nu begrensd door een aantal snijvlakken én een cel. Bemerk dat de top van de piramide niet binnen de cel hoeft te liggen. We krijgen nu een nieuwe zichtbaarheidstest voor onze bundel. De voorwaarde opdat een punt $P(x, y, z)$ voor de bundel zichtbaar is, is nog steeds dat dit tot de bundel behoort. En het zichtbaar volume – de inhoud van de bundel – is nog steeds convex. Het is immers de doorsnede van twee convexe eenheden. Het verschil is dat opdat P tot de bundel behoort, het nu noodzakelijk is dat deze tegelijkertijd binnen de bundel-piramide én de cel moet liggen. Zo komen we tot het algoritme 4.

Eigenschap 3: een bundel kan enkel invallen op wanden van de cel waartoe hij behoort: Dit volgt rechtstreeks uit eigenschap 2. Als de bundel op een wand buiten de cel zou invallen, dan zou hij moeten uitstrekken tot buiten zijn cel, en dat is in strijd met eigenschap 2. Bemerk dat dit een *enorme* winst is voor het zichtbaarheidsprobleem. Want voor iedere bundel moeten we enkel rekening houden met de wanden van de cel waarin



Figuur 3.10: een bundel wordt begrensd door zijn piramide én zijn cel (eigenschap 2)

Algorithm 4 vernieuwde punt-in-bundel test (eigenschap 2).

```
bool Beam3::contains(P)
{
    if (cell->contains(P))
    {
        for (∀Σ_i)
        {
            if (n_i · OP + d_i < 0) return false;
        }
        return true;
    }
    else
    {
       return false;
    }
}
```

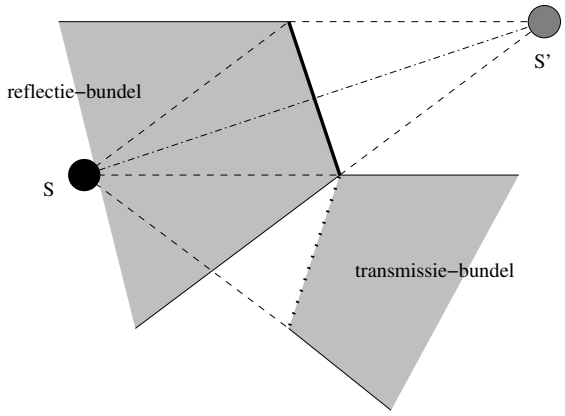
hij zich bevindt. Alle andere wanden spelen geen enkele rol meer. Vergelijk dit met een polygonsoup waar we alle wanden één voor één zouden moeten testen tegenover de bundel.

Eigenschap 4: geen enkel punt in de cel kan in de schaduw liggen van een obstakel: Dit is een rechtstreeks gevolg van het feit dat de cel convex is. Er is geen enkele wand van een convexe cel die een gedeelte van de cel aan het zicht kan onttrekken.

Eigenschap 5: of een wand van de cel zichtbaar is, wordt slechts bepaald door de bundel-piramide: Inderdaad, de wand ligt door constructie reeds binnen de cel, dus hoeft enkel nog getest te worden of die ook binnen de piramide ligt.

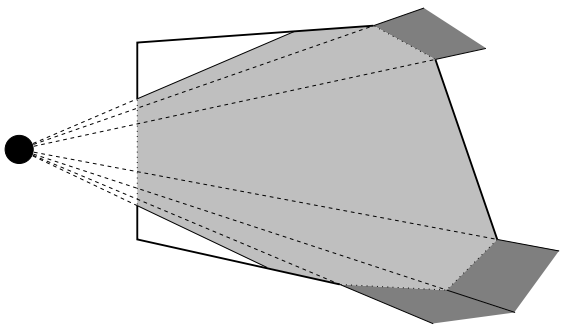
Eigenschap 6: voor iedere wand (portal) van de cel dat zichtbaar is in de bundel wordt een nieuwe bundel gecreëerd, een reflectie-bundel voor volle portals en een transmissie-bundel voor open portals: Iedere wand van de cel wordt getest

tegen over de bundel-piramide, en er wordt bepaald welk deel ervan er zichtbaar is. Bestaat er een zichtbaar gedeelte, dan wordt voor een volle portal een reflectie-bundel gecreëerd zoals beschreven in sectie 2.3.2. Voor een open portal wordt in dat geval een transmissie-bundel gecreëerd, analoog als voor een reflectie-bundel, behalve dat de bron niet gespiegeld wordt. Een reflectie-bundel zal tot dezelfde cel behoren als de originele bundel. Een transmissie-bundel zal tot de cel behoren aan de andere kant van de open portal. Beide situaties zijn geïllustreerd in figuur 3.11.



Figuur 3.11: Volle portals creëren reflectie-bundels, open portals transmissie-bundels (eigenschap 6).

Eigenschap 7: de unie van alle transmissie-bundels is precies gelijk aan het deel dat van de bundel dat voorbij de obstakels kan trekken: Dit is *de* eigenschap die ons convexiteitsprobleem oplost. Al deze bundels zijn immers reeds convex. M.a.w. de concave bundel die voorbij het obstakel trok, is nu reeds voor ons opgesplitst in convexe bundels (figuur 3.12).



Figuur 3.12: open portals lossen het convexiteitsprobleem voor de transmissie-bundels (donkergrijs) op (eigenschap 7.)

Eigenschap 8: als de bron van de bundel buiten de cel ligt, dan treedt de bundel de cel binnen door een deel van één wand, of ten hoogste één volledige

wand: Dit is correct, want zou het door meerdere wanden de cel binnenkomen, dan zou volgens eigenschap 6 de bundel opgesplitst zijn over deze verschillende wanden. De wand door welke de bundel de cel binnentreedt wordt expliciet bijgehouden. Want men wenst niet opnieuw een transmissie/reflectie-bundel te creëren voor deze wand.

3.2.2 Het algoritme

Aan de hand van al deze eigenschappen kunnen we nu een algoritme bouwen dat de bundeltrek implementeert voor convexe cellen, zoals geïllustreerd in 5. We hebben wel met twee zaken nog geen rekening gehouden:

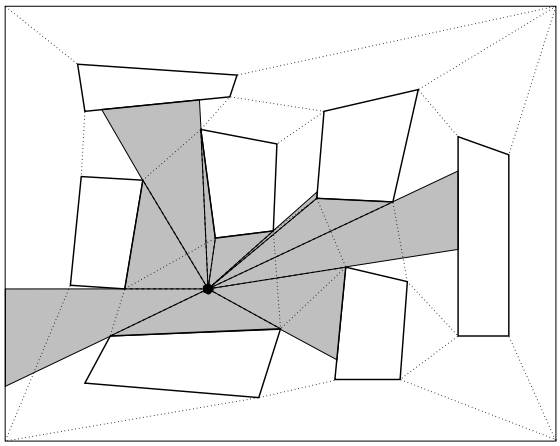
- We hebben tot nu toe steeds verondersteld dat de bundel afgebakend was met snijvlakken, maar we willen eigenlijk omnidirectionele puntbronnen invoeren. De manier waarop we dit kunnen doen is door een bundel te maken met een top (de puntbron) maar geen snijvlakken, en deze bundel in de cel te leggen waarin de puntbron ligt. Inderdaad, alle wanden van de cel zullen nu zichtbaar zijn omdat (a) de cel convex is, en (b) de bundel geen snijvlakken heeft die delen kunnen wegsnijden. In het volgende algoritme 5, wordt deze stap gedaan in `traceOmnidirectional`. Daarna wordt deze aan de eigenlijk bundeltrek functie `trace` toegevoerd. Omdat deze eerste bundel niet door een wand zijn cel is binnengekomen (de bron bevindt zich immers in de cel), zetten we `incoming face` op 0.
- We hebben in het algoritme enkel de bundeltrek in rekening gebracht, maar we moeten ook nog de padgeneratie implementeren die de eigenlijke geluidspaden naar de ontvangers genereert. Om dit te doen moeten we de ontvangers eerst in de wereld plaatsen. We zullen dit doen per cel. Per cel houden we een lijst bij van alle ontvangers (veralgemeend tot objecten) die de cel bevat. Als een bundel een cel binnenkomt, dan testen we welke ontvangers er binnen de bundel liggen, en genereren een geluidspad voor deze. Omdat we reeds weten dat het object binnen de cel ligt, kunnen we het eenvoudigere algoritme 1 gebruiken i.p.v. algoritme 4. De eigenlijke padgeneratie is hier niet van belang, en wordt besproken in secties 2.3.2 en 6.1.

Uiteindelijk komen we tot het algoritme 5. Zoals hierboven vermeld zal `traceOmnidirectional` zorgen voor de verwerking van een omnidirectionele puntbron. `trace` zal vervolgens kijken welke objecten er zichtbaar zijn en naar deze paden genereren. Voor een omnidirectionele bundel zullen dit alle objecten in de cel zijn, wat klopt met het feit dat deze geen snijvlakken heeft die objecten aan het zicht kunnen onttrekken. Daarna zal `trace` voor elke wand dat niet gelijk is aan de wand waardoor de bundel de cel binnenkomt (`incoming face`), kijken of er een stuk van zichtbaar is. Voor omnidirectionele bundels zullen opnieuw alle wanden volledig zichtbaar zijn. Als er inderdaad een zichtbaar stuk overblijft, dan wordt in het geval van een volle portal een reflectie-bundel gecreëerd, en in het geval van een open portal een transmissie-bundel. Deze nieuwe bundel wordt dan

Algorithm 5 het adaptieve bundeltrek algoritme

```
void traceOmnidirectional(source)
{
    cell = find the cell that contains source;
    beam = omnidirectional beam with top in source,
           without any clip planes;
    trace(beam, cell, 0);
}

void trace(beam, cell, incoming face)
{
    // generate paths
    for (each object in cell)
    {
         if (beam contains object)
         {
             generate path to object;
         }
    }

    // generate new beams
    source = top of beam;
    for (each face of cell that is not incoming face)
    {
        clipped face = part of face that is inside beam;
        if (clipped face exists)
        {
            if (face is a portal)
            {
                transmitted beam = construct beam from source
                                   through clipped face;
                neighbour cell = cell at other side of portal;
                trace(transmitted beam, neighbour cell, face);
            }
            else
            {
                virtual source = reflect source against face;
                reflected beam = construct beam from virtual
                                 source through clipped face;
                trace(reflected beam, cell, face);
            }
        }
    }
}
```

recursief verder verwerkt. Een voorbeeld van dit algoritme aan het werk, vind je in figuur 3.13, enkel transmissie werd in rekening gebracht om de duidelijkheid te behouden.



Figuur 3.13: adaptieve bundeltrek aan het werk in een convexe celstructuur

3.3 Winged-pair datastructuur voor convexe cellen

We hebben nu reeds het idee van convexe cellen om het probleem met de concave bundels op te lossen, maar we hebben nog steeds geen goede structuur in de wereld. We zouden kunnen alle wanden (enkelzijdige wanden, open en volle portals) opslaan in een polygon-soup, maar dan zouden we nog steeds zware zoekoperaties nodig hebben om bijvoorbeeld de eerstvolgende portals te vinden waarop een bundel zal invallen. Een quad- of octree, of BSP-tree zouden hierin kunnen helpen, maar we willen beter. We willen weten welke wanden/portals er behoren tot één cel, welke cel er aan de andere zijde van een portal ligt, of welke portal er naast een andere ligt. Dit zijn allemaal vragen die men stelt bij de topologische doorloop van een wereld. We hebben een datastructuur nodig die deze vragen zeer snel kan beantwoorden.

Voor dit eindwerk hebben we geopteerd voor de winged-pair variant zoals kort beschreven in [12, 13]. De winged-pair datastructuur kan worden gezien als een uitbreiding op de half-edge datastructuur [21]. Deze datastructuur zal expliciet alle topologische relaties tussen verschillende elementen in de wereld opslaan, zodat deze zeer snel op te vragen zijn. De prijs die we hiervoor betalen is extra geheugen, maar de snelheidswinst die we in de plaats krijgen compenseert dit ruimschoots.

We zullen de opbouw van de winged-pair datastructuur bespreken aan de hand van de verschillende elementen zoals deze in `bass3` voorkomen.

3.3.1 World3

```
class World3
{
    std::vector<Cell3*> m_cells;
    std::vector<Edge3*> m_edges;
```

```
    std::vector<Vertex3*> m_vertex;
    std::vector<Objects3*> m_objects;
}
```

De wereld bestaat uit verschillende convexe cellen (`Cell3`) die aan elkaar grenzen. De doorsnede van elke twee willekeurige cellen is ledig (eventueel op de wanden van de cellen na). De unie van alle cellen vormt een AABB, alhoewel voor de werking van de bundeltrekker alleen vereist is dat deze unie convex is. Elke cel heeft bijgevolg in elke richting een buurcel, behalve voor de cellen aan de rand van de wereld. Alle cellen van de wereld worden door `World3` bijgehouden in een vector `m_cells`.

`World3` zal ook nog een aantal andere zaken bijhouden, zoals een vector van alle `Edge3`'s, `Vertex3`'s en `Object3`'s. Dit heeft uitsluitend met het interne huishouden van `World3` te maken, en wordt hier niet verder besproken.

3.3.2 Cell3

```
class Cell3
{
    std::vector<Face3*> m_faces;
    std::vector<Object3*> m_objects;
    World3* m_world;
    void* m_medium;
    unsigned m_visitID;
};
```

Een cel wordt begrensd door een aantal convexe polygonen (wanden, portals, `Face3`) en is gesloten. De cel heeft pointers naar zijn wanden gestockeerd in een vector. Een cel kan ook objecten bevatten, waarover verder meer. Ook van deze objecten worden pointers bijgehouden, als ook een pointer naar de wereld waarvan de cel uitmaakt en een pointer naar de eigenschappen van het medium dat door de cel wordt afgebakend. Tot slot wordt er een kental `m_visitID` bijgehouden waarmee algoritmes kunnen herkennen of ze deze cel reeds bezocht hebben. Dit kental wordt door momenteel enkel door het algoritme `Cell3Finder` bij gehouden, zoals beschreven in sectie 4.4.

3.3.3 Face3

```
class Face3
{
    Plane3 m_plane;

    Cell3* m_frontCell;
    Cell3* m_backCell;
```

```
        void* m_frontHandle;
        void* m_backHandle;

        Pair3* m_pair;
    };
```

Elke wand wordt gedeeld door twee cellen. Behalve voor de wanden die aan de rand van de wereld, die behoren toe tot slechts één cell. Als een wand gedeeld wordt, dan wordt die steeds volledig gedeeld. Hierdoor zal een grote (volle of open) wand soms moeten worden opgedeeld in meerdere kleinere en coplanaire wanden, zodat er één volledige (deel)wand kan gedeeld worden met een kleinere buurcel. Een gevolg is dat een cel meerdere wanden kan hebben die coplanair zijn. Dit is niet gebruikelijk in de wiskunde, maar is hier noodzakelijk voor de goede werking.

Elke wand heeft een draagvlak (`Plane3`) en verwijzingen naar de cel(len) waaraan ze is gekoppeld. Het draagvlak heeft een normaalvector en bijgevolg een voor- en achterkant (de voorkant is in de richting van de normaalvector). Op deze manier kunnen we beide cellen onderscheiden als `m_frontCell` en `m_backCell`. Aan een wand kunnen ook eigenschappen worden gekoppeld, en dit kan voor beide zijden afzonderlijk. Aldus hebben we een `m_frontHandle` en een `m_backHandle`.

De algoritmes voor het bundeltrekken beschouwen een cel van binnenuit. Daarom willen we dat de normaalvector van alle wanden van elke cel naar binnen zijn gericht. Dit is echter niet mogelijk omdat de meeste wanden worden gedeeld door twee cellen. We weten wel dat de normaalvector steeds naar `m_frontCell` is gericht. We koppelen hier meteen ook een *polariteit*: als de normaalvector naar een cel toe is gericht, dan noemen we dit positief gepolariseerd; als ze van de cel weg is gericht, dan noemen we dit negatief. Dit heeft altijd als gevolg dat voor de `m_frontCell` de wand positief is gepolariseerd, en voor de `m_backCell` negatief. Dit kunnen we uitbuiten om snel de polariteit van een wand t.o.v. een cel te bepalen, en zo nodig de normaalvector om te draaien. Wat verder zal blijken dat we ook een ander soort polariteit moeten bijhouden, vandaar dat we deze polariteit `facePolarity` zullen noemen.

Elke wand wordt begrensd door een kring van ribben (`Edge3`) en deze kring is gesloten. De ribben worden echter niet onmiddellijk gekoppeld aan een wand, maar via een extra object dat we een paar noemen (`Pair3`). De reden hiervoor is dat we goede topologische informatie in het model willen stoppen, zodat we het model op een eenvoudige manier kunnen doorlopen[1]. Elke wand bestaat dus uit een kring van zulke paren, die de omtrek van de wand eenduidig bepaalt. De wand bevat een pointer naar één van zijn paren, en

[1] Dit is vooral noodzakelijk bij de implementatie van diffractie. Om te weten of er al dan niet diffractie aan de ribbe van een wand kan optreden, moeten alle wanden rond de ribbe worden opgevraagd om te controleren of er een obstakel aan de ribbe grenst (sectie 5.2.2). Dit gebeurt door aan de ribbe een circulaire lijst bij te houden van deze wanden met behulp van `Pair3`'s: een `Pair3` is een dubbelzijdige koppeling tussen ribbe en wand.

verder zijn deze paren gekoppeld in een dubbel-gelinkte en circulaire lijst. Over deze paren verder nog meer.

Een laatste aspect is dat we steeds eisen dat een wand op z'n minst een `m_frontCell` heeft. Een `m_backCell` hoeft niet altijd aanwezig te zijn, denk maar aan de cellen die aan de rand van de wereld liggen.

3.3.4 Edge3

```
class Edge3
{
    Vertex3* m_tail;
    Vertex3* m_head;

    Pair3* m_pair;
};
```

Een ribbe is een gericht lijnstuk tussen twee hoekpunten (`Vertex3`). Het ene hoekpunt is de staart (*tail*) en de andere de kop (*head*). Deze hoekpunten worden gedeeld over meerdere ribben. De ribbe is gericht van staart naar kop.

Er is geen overlap bij ribben toegestaan. D.w.z. dat als twee ribben meer dan één punt gemeen hebben, deze ribben moeten worden opgesplitst zodat het gemeenschappelijk stuk een nieuwe ribbe wordt. Deze ribbe wordt dan volledig gedeeld.

Elke ribbe wordt gedeeld door minstens twee wanden. Elke ribbe houdt ook een kring van paren bij, die weergeeft welke wanden aan deze ribbe zijn gekoppeld. Deze paren zijn dezelfde als die door een wand worden gestockeerd. M.a.w. als een wand een paar bezit die het aan een bepaalde ribbe koppelt, dan zal de ribbe datzelfde paar bezitten met een koppeling naar diezelfde wand. Opnieuw heeft de ribbe een pointer naar één van zijn paren, en verder zijn deze paren in een dubbel gelinkte lijst gekoppeld.

3.3.5 Pair3

```
Class Pair3
{
    Face3* m_face;
    Edge3* m_edge;

    Pair3* m_clock;
    Pair3* m_counterClock;
    Pair3* m_spin;
    Pair3* m_counterSpin;
```

```
    Polarity m_edgePolarity;
};
```

Een paar koppelt een ribbe en een wand, en is steeds uniek. Het is precies in deze paren dat de kracht schuilt om op een eenvoudige manier de wereld te doorkruisen. Een paar is tegelijkertijd in staat om:

- de ribben en hun volgorde vast te leggen voor de omsluiting van een wand.
- voor een ribbe vast te leggen welke wanden deze ribbe delen en in welke volgorde ze moeten worden doorlopen.

De volgorde van de paren voor een wand is tegenwijzerzin t.o.v. zijn normaalvector (dus volgens de rechterhandregel). Een paar heeft zowel een koppeling naar het vorige en het volgende paar. Hiermee kunnen we de paren respectievelijk zowel in wijzerzin als tegenwijzerzin doorlopen. Hiervoor dienen respectievelijk de operators `clock()` en `counterClock()`. Een belangrijk aspect hierbij is dat we steeds willen doen alsof een wand naar de cel toe gericht is[2]. D.w.z. dat we voor de `m_backCell` de pairs in omgekeerde richting moeten doorlopen. Om dit te realiseren kunnen we de cel als parameter meegeven bij het bepalen van het volgende paar.

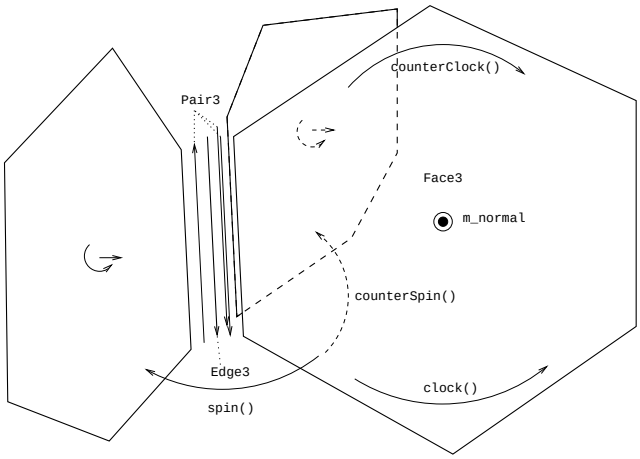
```
Pair3* Pair3::counterClock(Cell3* a_cell) const
{
    return m_face->frontCell() == a_cell ?
           m_counterClock : m_clock;
}
```

De paren rond een ribbe worden ook in tegenwijzerzin t.o.v. de richting van de ribbe gesorteerd. Tegenwijzerzin noemen we hier de spin-richting, en wijzerzin de tegenspin-richting. We kunnen deze paren opvragen respectievelijk met de operators `spin()` en `counterSpin()`. Interessant om op te merken is dat twee opeenvolgende wanden rond één ribbe, steeds een cel gemeenschappelijk hebben. Immers, een ribbe is waar voor een cel twee wanden samenkomen.

3.3.6 Vertex3

```
class Vertex3
{
    Point3 m_position;
}
```

[2]Dit is vooral van belang voor het clipping algoritme en de constructie van bundels (sectie 5.1.3) omdat deze de wand in tegenwijzerzin willen doorlopen v*anuit het standpunt van waarnemer in de cel*. Ook levert dit een zeer eenvoudige *punt-in-cel* test op. Analoog aan de *punt-in-bundel* test ligt een punt dan in de cel als het voor alle wanden ligt.



Figuur 3.14: `Pair3`'s koppelen `Face3`'s en `Edge3`'s

De hoekpunten zijn de punten die verschillende ribben aaneenschakelen. Elke ribbe heeft dan ook twee hoekpunten: de staart en de kop. Een hoekpunt bestaat slechts uit een drie-dimensionaal punt, dat de positie van het hoekpunt aanduidt.

3.3.7 Polarity

Een belangrijk aspect dat nog niet werd vermeld, is dat elke paar ook een polariteit met zich meedraagt. Dit heeft de volgende reden. Als we de ribben rond een wand bekijken, dan willen we dat hun richting (staart-kop) in tegenwijzerzin volgens de normaalvector ligt. Maar door dat ribben gedeeld kunnen worden over verschillende wanden, kunnen we dit niet volhouden. D.w.z. dat sommige ribben in wijzerzin liggen. Een paar waarbij de ribbe in wijzerzin ligt volgens de normaalvector van de wand, noemen we negatief gepolariseerd. Bij tegenwijzerzin positief gepolariseerd. Om het onderscheid te maken met de `facePolarity` zoals hierboven beschreven, zullen we deze polariteit `edgePolarity` noemen.

`facePolarity`: positief als men zich voor een wand bevindt, negatief in het andere geval.

`edgePolarity`: positief als een ribbe in tegenwijzerzin aan een wand hangt, negatief in het andere geval.

Dit wordt pas interessant bij het draaien rond een ribbe. We moeten namelijk steeds in gedachte houden dat we de volledige wereld bekijken vanuit het standpunt van binnen één cel. Voor de eenvoud van de algoritmes, eisen we steeds dat alle normaalvectoren van de wanden van die cel naar binnen gericht zijn, en dat voor elk van die wanden zijn ribben in tegenwijzerzin gericht zijn. Dit heeft als gevolg dat de operator `spin()` steeds een wand zou moeten opleveren dat aan diezelfde cel grenst. M.a.w. we willen dat de operators niet de eigenlijke datastructuur weergeven, maar de virtuele datastructuur die elke cel voor zich

ziet. Dit kan misschien het beste geïllustreerd worden aan de hand van de implementatie van een aantal operatoren (waarvan al één hierboven vermeld is).

facePolarity

De polariteit van een wand t.o.v. de waarnemer kan aan de hand van twee soorten parameters worden bepaald: de cel waarin de waarnemer zich bevindt, of zijn exacte positie. In beide gevallen is de polariteit positief als de waarnemer zich voor de wand bevindt. Belangrijk om op te merken is dat in het eerste geval de cel moet palen aan de wand, m.a.w. de cel moet ofwel de `m_frontCell` ofwel de `m_backCell` van de wand zijn.

```
Polarity Face3::facePolarity(const Cell3* a_cell) const
{
   return m_frontCell == a_cell ? Positive : Negative;
}

Polarity Face3::facePolarity(const Point3& a_point) const
{
    return m_plane.inFront(a_point) ? Positive : Negative;
}
```

Eens je deze polariteit weet, kan je hiermee vragen stellen aan de wand. Als je bv. de normaalvector van een wand opvraagt, dan wil je steeds dat deze naar jou toe is gericht, zodat je je voor de wand bevindt. Je geeft hiervoor jouw polariteit t.o.v. de wand mee. Voor het gebruiksgemak zijn zulke operatoren veelal overladen met versies die een cel of een positie aannemen. Deze bepalen dan intern de polariteit vooraleer het antwoord te geven. Zo hebben we drie functies om de normaalvector van een wand op te vragen.

```
Vector3 Face3::normal(Polarity a_facePolarity) const
{
    return a_facePolarity ? m_plane.normal : -m_plane.normal();
}

Vector3 Face3::normal(const Cell3* a_cell) const
{
    return normal(facePolarity(a_cell));
}

Vector3 Face3::normal(const Point3& a_point) const
{
    return normal(facePolarity(a_point));
}
```

Merk op dat je ook van een wand-ribbe paar de `facePolarity` kunt opvragen, want zo'n paar is met één enkele wand verbonden. Het paar geeft dan de vraag gewoon door aan zijn wand. De `clock()` en `counterClock()` operatoren kunnen nu ook overladen worden om een facePolarity, een cel of een een punt aan te nemen. Dit werkt op analoge wijze als voor `normal()`.

EdgePolarity

De polariteit van een ribbe bepalen is net iets minder eenvoudig.

- Daar waar de `facePolarity` rechtstreeks aan de `Face3` werd opgevraagd, moeten we dit voor de `edgePolarity` aan een `Pair3` opvragen. `EdgePolarity` geeft namelijk de relatie tussen een ribbe en een wand weer. Deze is onafhankelijk van de waarnemer in zoverre de wand onafhankelijk is van de waarnemer. Als de ribbe in tegenwijzerzin is geörienteerd t.o.v. de wand, dan is deze positief, anders is deze negatief. Vandaar dat de `edgePolarity` kan worden opgeslagen in een paar.
- De wand is niet zomaar onafhankelijk van de waarnemer. De `edgePolarity` die in het paar is opgeslagen, is bepaald in de veronderstelling dat de wand positief werd bekeken. Als de wand nu met negatieve polariteit wordt waargenomen, dan wordt wijzerzin tegenwijzerzin en omgekeerd. In dat geval keert ook de `edgePolarity` om.

Uiteindelijk moeten we om de `edgePolarity` te bepalen, ook meegeven hoe we de wand bekijken.

```
Polarity Pair3::edgePolarity(Polarity a_facePolarity) const
{
    return a_facePolarity? m_edgePolarity : -m_edgePolarity;
}
```

Eens we de `edgePolarity` van een ribbe weten, kunnen we ook de operatoren `spin()` en `counterSpin()` implementeren. `spin()` draait rond de ribbe op zoek naar de volgende `Pair3`. Het doet dit in tegenwijzerzin bepaald door de rechterhandregel waarbij de duim van de staart naar de kop van de ribbe is gericht. De kop en staart worden echter "eventjes verwisseld" als de `edgePolarity` negatief is, m.a.w. het draait in wijzerzin om de ribbe.

Een `Pair3` structuur houdt reeds de resultaten van `spin()` en `counterSpin()` bij waarbij positieve `edgePolarity` verondersteld wordt, dus onafhankelijk van de gestockeerde `m_edgePolarity`. De `spin()` operator vertaalt zich zo tot volgende code:

```
Pair3* Pair3::spin(Polarity a_edgePolarity) const
{
```

```
    return m_edgePolarity == Positive ? m_spin : m_counterSpin;
}

Pair3* Pair3::spin(const Cell3* a_cell) const
{
    return spin(edgePolarity(Cell3));
}
```

3.3.8 Pair3 iterators

`Pair3`'s lijmen letterlijk de wereld aan elkaar. Daardoor moeten deze veelvuldig worden doorlopen. Om dit proces wat te vereenvoudigen kan men iterators maken, zoals beschreven in [14]. De meest gekende iterators zijn die uit de `C++` standaardbibliotheek [27] voor het doorlopen van STL containers. `bass3` kent acht `Pair3` iterators: er is telkens een constante en een niet-constante iterator voor de 4 operatoren `clock()`, `counterClock()`, `spin()` en `counterSpin()`. Het verschil tussen constante en niet-constante iteratoren is gelijkaardig aan het verschil tussen `const_iterator` en `iterator` in de `C++` standaardbibliotheek[27].

operator	iterator	constant iterator
`clock()`	`Pair3ClockIterator`	`Pair3ConstClockIterator`
`counterClock()`	`Pair3CounterClockIterator`	`Pair3ConstCounterClockIterator`
`spin()`	`Pair3SpinIterator`	`Pair3ConstSpinIterator`
`counterSpin()`	`Pair3CounterSpinIterator`	`Pair3ConstCounterSpinIterator`

We behandelen hier één iterator in detail: `Pair3ClockIterator`.

```
class Pair3ClockIterator
{
public:
    void reset(Pair3* a_beginPair, Polarity a_polarity = Posi-
tive);
    void reset(Pair3* a_beginPair, const Cell3* a_cell);
    void reset(Pair3* a_beginPair, const Point3& a_point);

    const Pair3ClockIterator& operator++();
    Pair3ClockIterator operator++();
    bool end() const;

    Pair3* get() const;
```

```
    Pair3* operator->() const;
    Pair3& operator*() const;

    bool operator==(const Pair3* a_pair);
    bool operator!=(const Pair3* a_pair);

private:
    Pair3* m_beginPair;
    Pair3* m_currentPair;
    Polarity m_polarity;
};
```

Iedere iterator wordt met een `reset()` geïnitialiseerd om de iteratie te starten aan `a_beginPair`. Er zijn hiervan drie overladen versies om een polariteit aan de iteratie mee te geven, gelijkaardig aan de `clock()` operator zelf. Vervolgens kan men met pre- of postincrement de iteratie doorlopen. Daarbij wordt telkens de `clock()` operator uitgevoerd met de ingestelde polariteit. Dit blijft duren tot de iteratie de oorspronkelijke `Pair3` tegenkomt (alle iteraties gebeuren noodzakelijk op een circulaire lijst). Op dat moment komt de operator `end()` hoog te staan, en wordt `m_currentPair` op *null* gezet. De huidige `Pair3` `m_currentPair3` waarnaar de iterator wijst, kan met verschillende methodes worden opgevraagd, en bovendien kan ook nog worden getest of een iterator al dan niet naar een bepaalde `Pair3` wijst.

Met behulp van deze iteratoren kunnen `Pair3`'s veel compacter en intuïtiever worden doorlopen. Als voorbeeld geven we het kloksgewijs doorlopen van alle `Pair3`'s van een `Face3* face`, zonder en met iterators. Hierbij wordt de polariteit gegeven door een waarnemingspunt `Point3 point`.

```
Pair3* currentPair = face->pair();
do
{
    currentPair->foo();
    currentPair = currentPair->clock(point);
}
while (currentPair != face->pair());

Pair3ClockIterator pit;
for (pit.reset(face->pair(), point); !pit.end(); ++pit)
{
    pit->foo();
};
```

De winst lijkt niet zo groot, maar dat is het zeker wel als je weet dat het vergeten van de de regel `currentPair = currentPair->clock(point);` de voornaamste oorzaak van fouten is in het bovenste voorbeeld. Gelukkig zijn dit soort fouten meestal snel gevonden. Toch biedt het onderste voorbeeld onze voorkeur omdat het veel intuïtiever is.

3.3.9 Vector3, Point3 en Point3h

Traditioneel worden punten en vectoren in een computerprogramma door één soort object in het geheugen opgeslagen. Meestal is dit een vector object, en men gebruikt het ook om punten op te slaan. Deze gewoonte is vrij aannemelijk in een `C`-omgeving, waar zo'n object een structuur van 3 variabelen is.

```
struct Vector3
{
    double x;
    double y;
    double z;
};
```

In `C++` kunnen we echter beter [7]. Door twee verschillende structuren te maken voor vectoren en punten, kunnen we het wiskundig verschil tussen deze ook in onze code tot uiting brengen. We kunnen bijvoorbeeld geen punten schalen optellen *(op barycentrische combinaties na)*. We kunnen geen scalair of vector product nemen tussen twee punten *(dot and cross product)*, en anderzijds kunnen we geen afstand meten tussen twee vectoren. De kerngedachte hier is: een punt is geen vector, maar we gebruiken een plaatsvector om een punt in het werkgeheugen op te slaan. We brengen dit tot uiting door een punt te implementeren in functie van een vector, maar zonder dat het een vector wordt:

```
struct Point3
{
    Vector3 vector;
};
```

Dit lijkt allemaal wat omslachtig, maar de kracht van dit onderscheid wordt pas echt duidelijk als we de sterke *type safety* van `C++` beschouwen. We nemen als voorbeeld een constructor van een vlak `Plane3` dat een normaalvector en één steunpunt van het vlak aan neemt. Als we voor punten en vectoren één structuur gebruiken, dan heeft dit tot gevolg dat een onoplettende programmeur een vlak kan construeren waarbij de normaalvector als steunpunt wordt beschouwd en omgekeerd. De compiler zal dit aanvaarden, en het lange foutzoeken kan beginnen.

```
class Plane3
{
    Plane3(Vector3 a_normal, Vector3 a_support);
};

Vector3 normal(1, 2, 3);
Vector3 support(4, 5, 6);
Plane3 plane(support, normal);
```

Met het onderscheid tussen punten en vectoren zoals deze in `bass3` wordt gemaakt, zal de compiler onderstaand voorbeeld nooit willen compileren, en de programmeur wordt een aantal lange zoekuurtjes bespaard. Over het algemeen geldt dat we *compile-time* fouten boven *run-time* fouten moeten verkiezen [27, 2].

```
class Plane3
{
    Plane3(Vector3 a_normal, Point3 a_support);
};

Vector3 normal(1, 2, 3);
Point3 support(4, 5, 6);
Plane3 plane(support, normal); // compile-time error!
```

Een probleem in dit concept zijn barycentrische combinaties. Het is namelijk toegestaan om punten samen te tellen, op voorwaarde dat de som van alle gewichten gelijk is aan één. Enkel onder deze voorwaarde is het resultaat opnieuw een punt:

$$P = \sum_{k=1}^{n} w_k Q_k \Leftrightarrow \sum_{k=1}^{n} w_k = 1 \tag{3.1}$$

Bij een automatische barycentrische combinatie zullen we de combinatie steeds door de som der gewichten delen, zodat de voorwaarde automatisch vervuld is:

$$P = \frac{\sum_{k=1}^{n} w_k Q_k}{\sum_{k=1}^{n} w_k} \tag{3.2}$$

Dit is een veel voorkomende operatie, en in [7] is dit wat ongelukkig opgelost door de gewone optelling en scalaire vermenigvuldiging voor punten toe te laten, waardoor de voorwaarde niet gegarandeerd is. In `bass3` hebben we een betere oplossing geïmplementeerd door de introductie van een derde structuur `Point3h`. Dit is een homogeen 3D punt, ofwel een 3D punt met ingebouwd gewichtsfactor. Hierdoor kunnen we ongestoord punten accumuleren in een `Point3h`. Op het einde delen we de som door het totale gewicht met

behulp van `affine()` en dit resultaat is opnieuw een `Point3`. Volgend voorbeeld komt overeen met de automatische barycentrische combinatie (3.2).

```
Point3 Q[n];
Real w[n];

Point3h temp; // default initialisation to (0, 0, 0)
for (unsigned k = 0; k < n; ++k)
{
    temp += w[k] * Q[k];
}

Point3 P = temp.affine();
```

Deze methode om onderscheid te maken tussen punten en vectoren komt men eigenlijk veel te weinig tegen. De meeste programmeurs blijven overtuigd dat één enkele structuur volstaat voor alle doeleinden en dat de extra objecten de programma's zouden vertragen. Dit laatste is echter meestal ongegrond, omdat met een *inline* implementatie, de meeste huidige `C++` compilers beide uitvoeringen even snel zullen maken. Toch zijn er anderen die net als hier dit onderscheid wel maken, zoals CGAL [6].

Hoofdstuk 4

Opbouw van de omgeving: preprocessors

Vooraleer we een simulatie kunnen uitvoeren, moeten we de wereld in het geheugen opbouwen. Hiervoor dienen de preprocessors. Aan de hand van invoerdata bouwen ze een `World3` object die geschikt is om aan de eigenlijke bundeltrekker `Beam3Tracer` door te geven. Door deze opbouw niet in de `World3` klasse maar in een aparte preprocessor onder te brengen, is het mogelijk om meerdere preprocessors te ontwerpen die elk een wereld kunnen aanmaken aan de hand van andere invoerdata. Belangrijk hierbij is dat alle functionaliteit die typisch is voor het soort invoerdata in de preprocessor wordt ondergebracht. In het kader van dit eindwerk hebben we twee preprocessors gebouwd.

`Preprocessor2D5`: Deze bouwt een `World3` object op aan de hand van zogenaamde 2.5D informatie.

`PreprocessorSHP`: Bouwt ook een `World3` object op aan de hand van 2.5D informatie, maar haalt deze informatie uit een SHP bestand. Inwendig wordt hiervoor de `Preprocessor2D5` gebruikt.

Het is uiteraard mogelijk om een preprocessor te bouwen die van pure 3D informatie uitgaat en waarop geen beperkingen meer liggen zoals bij `Preprocessor2D5`. Dit lag echter niet in de doelstelling van dit eindwerk. Belangrijk is om te onthouden dat de datastructuur `World3` volledig 3D is.

Verder zijn er ook nog een aantal bewerkingen op `World3` mogelijk die we wensen onder te brengen in dit hoofdstuk, omdat ze niet behoren tot het eigenlijke bundeltrekken. Het zijn er drie:

`World3Optimizer`: Zoals eerder vermeld, maakt `Preprocessor2D5` (en dus ook `PreprocessorSHP`) enkel cellen aan met een driehoekig grondvlak. Dit is echter geen

vereiste voor `World3` en de werking van de `Beam3Tracer`. `World3Optimizer` poogt de wereld te vereenvoudigen door cellen samen te voegen tot grotere convexe cellen. Dit heeft tot gevolg dat er minder portals aanwezig zijn, waardoor de bundeltrekker vlotter zal werken.

`Cell3Finder`: Deze bewerking spoort in de wereld de `Cell3` op die een gegeven punt bevat.

`Cell3Bounder, World3Bounder`: Deze algoritmes zoeken de kleinste AABB die respectievelijk een `Cell3` of een volledige `World3` bevat.

4.1 Preprocessor2D5

Misschien een van de moeilijkste zaken om te schrijven, zijn algoritmes die een datastructuur moeten opbouwen. Zo is ook voor dit eindwerk heel wat tijd gespendeerd aan het werkende krijgen van deze preprocessor. We bekijken eerst vanuit welke data we vertrekken om `World3` op te bouwen, en dan hoe we precies te werk gingen.

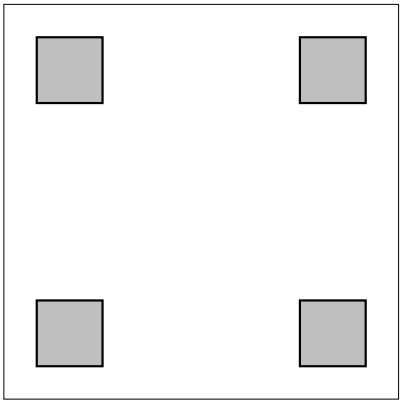
4.1.1 Invoerdata

`Preprocessor2D5` heeft als taak om een `World3` object op te bouwen aan de hand van 2.5D informatie. Met 2.5D bedoelen we dat de informatie in 2D kan worden bekeken - we kunnen er een plattegrond van maken zonder dat er overlappingen zijn - maar dat aan elk punt ook een hoogte wordt toegekend. Op deze manier heeft `Preprocessor2D5` de volgende data voor handen:

- Een hoogtemap. Dit is een functie $z = f(x, y)$ dat van elk 2D punt zegt wat het niveau van het maaiveld is. Dit is de grond waarop de gebouwen (obstakels) zullen worden gebouwd.
- Een aantal obstakels in de vorm van 2D polygonen. Deze polygonen kunnen zowel convex als concaaf zijn, maar niet complex. Deze polygonen beschrijven de gebouwen die op het maaiveld zullen worden geplaatst. Zo'n gebouw heeft verticale muren en een horizontaal dak. Het dak kan eventueel hellend zijn, maar we beschouwen dit nog niet. Aan deze polygonen worden ook nog een aantal eigenschappen toegekend:
  - de hoogte of de z-coördinaat van het dak, of de vergelijking van een vlak in het geval van een hellend dak
  - de akoestische eigenschappen van de buitenmuren van het gebouw, deze zijn gelijk voor alle buitenmuren van één gebouw.
  - de akoestische eigenschappen van de binnenmuren, deze zijn opnieuw gelijk.
  - de akoestische eigenschappen van de buitenkant van het dak

  - de akoestische eigenschappen van de binnenkant van het dak

- Een rechthoekige rand (met de assen parallel aan de coördinaatassen) die de omgeving in het horizontaal vlak begrensd.
- Een begrenzing in de verticale richting: z_{min}en z_{max}. Samen met de horizontale rechthoek vormt dit een balk of een AABB (axis-aligned bounding box, [23], sectie 4.5).



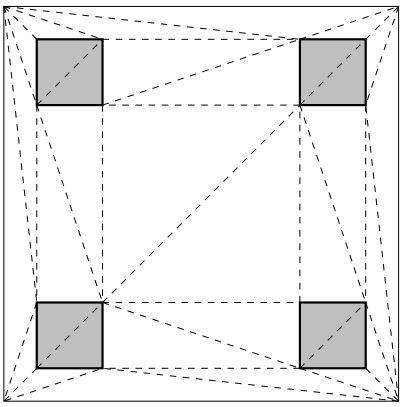
Figuur 4.1: in een 2.5D omgeving zijn de obstakels gegeven door 2D polygonen

Preprocessor2D5 zal aan de hand van deze informatie eerst de wereld in twee dimensies opdelen in convexe cellen. Hiervoor zal het een triangulator gebruiken. Daarna zal het deze 2D cellen optrekken naar drie dimensionale prisma's aan de hand van de hoogte informatie.

4.1.2 2D Triangulatie

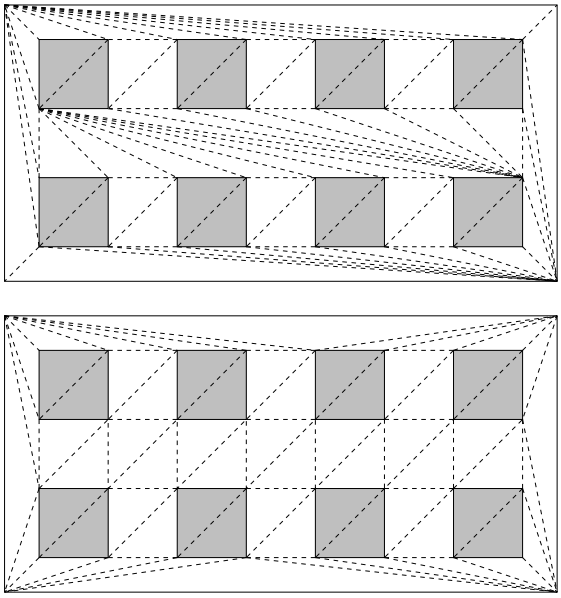
De eerste bewerking die we op de data uitvoeren, is het opsplitsen in convexe cellen. Zelfs als alle obstakels convex zijn, zal de ruimte tussen de obstakels dat doorgaans niet zijn. Een eerste methode om dit te doen, is met een leafy 2D BSP-tree. Inderdaad, eens alle wanden van alle obstakels in de BSP-tree zijn ingevoegd, hebben we alle convexe cellen die we nodig hebben. Een reden om dit niet te doen, is dat een BSP-tree deze wereld meestal niet zo mooi opsplitst. Verschillende wanden worden doorsneden, en bovendien kunnen zeer fijne en langwerpige cellen ontstaan door numeriek precisie, zoals reeds in sectie 3.1.3 werd aangehaald.

Een betere manier is om dit te doen met een triangulatie. Zoals de naam laat vermoeden, zal deze de wereld opdelen in een aaneenschakeling van driehoeken. Het voordeel van deze aanpak is dat er geen wanden moeten doorsneden worden. Alle hoekpunten die nodig zijn voor de triangulatie, zijn reeds voorhanden in de originele data.



Figuur 4.2: 2D Triangulatie van de omgeving

Er zijn verschillende manieren denkbaar om een wereld op te delen in driehoeken. Een bijzondere triangulatie is de zogenaamde *Delaunay triangulatie*. We gaan hier niet in detail op in, maar het belangrijkste is dat een Delaunay triangulatie garandeert dat alle binnenhoeken van de driehoeken zo groot mogelijk gekozen zijn. M.a.w. er is geen andere triangulatie denkbaar waarbij de binnenhoeken nog groter kunnen zijn. Dit is van belang om dat we wensen langwerpige driehoeken te vermijden, omdat deze kunnen leiden tot numerieke problemen. Vergelijk in figuur 4.3 de gegenereerde driehoeken van een willekeurige triangulatie met dat van een Delaunay triangulatie. Bemerk dat we niet alle langwerpige driehoeken kunnen vermijden, maar toch wel een groot aantal.



Figuur 4.3: een willekeurige triangulatie (boven) t.o.v. een Delaunay triangulatie (onder)

Men kan opmerken dat een driehoek niet de grootst mogelijke polygon is om een wereld op te delen in convexe cellen. Inderdaad, men kan in figuur 4.2 verschillende driehoeken

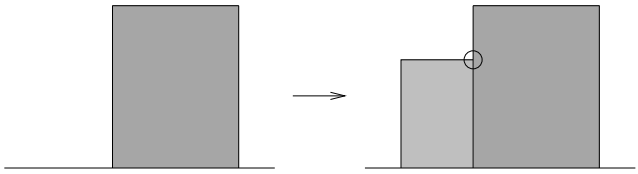
vinden die men kan samen smelten tot een convexe vier-, vijf-, n-hoek. Voor de eenvoud van de volgende stap in de preprocessor, zullen we dit hier niet doen. Wel zullen we dit achteraf in 3D doen op het World3 object dat de preprocessor heeft gecreëerd. Dit bespreken we in sectie 4.3.

4.1.3 Opbouw van 3D cellen

Om een World3 object te verkrijgen, rest ons nu nog de taak om de driehoeken uit sectie 4.1.2 op te trekken naar drie dimensionale veelvlakken, cellen (`Cell3`). Het spreekt voor zich dat al deze cellen driehoekige prisma's zullen zijn: het grondvlak is een driehoek, en ze hebben rechtopstaande wanden. Deze stap lijkt triviaal, maar het is het helemaal niet. Er moeten namelijk per driehoek een aantal cellen boven elkaar geplaatst worden. Herinner dat alle cellen van een World3 object een balk vormen: de AABB van de wereld (axis-aligned bounding box , [23], sectie 4.5). Het horizontale gedeelte van deze balk is reeds uitgetekend in de triangulatie, maar hier moet ook nog het verticale gedeelte worden ingebouwd. Er is hiervoor een z_{min} en een z_{max} voorzien. De situatie is dan als volgt:

- z_{min} zal de wereld langs onder begrenzen en ligt onder het minimum maaiveld. Hierdoor zullen we per driehoek een cel moeten maken tussen z_{min} en het maaiveld. Deze cellen zullen nooit worden bereikt door een bundel of geluidstraal, althans zolang de bron boven het maaiveld blijft. Het maaiveld is wordt immers als ondoordringbaar verondersteld. De enige reden waarom we toch cellen onder het maaiveld stoppen, is om het geheel balkvormig te krijgen. Het maaiveld hoeft immers niet vlak te zijn.
- z_{max} begrenst de wereld langs boven en ligt hoger dan elk obstakel. Per driehoek zal dan een cel gemaakt worden tussen het maaiveld en z_{max}, tenzij boven een gebouw.
- Voor elke driehoek dat deel uitmaakt van een gebouw, moet een cel gemaakt worden van het maaiveld tot aan het dak van het gebouw, en een een cel vanaf het dak tot z_{max}.

Bijgevolg moeten er per driehoek twee tot drie cellen worden aangemaakt. Het eerste idee was om driehoek per driehoek af te lopen en deze cellen met al hun wanden (`Face3`) volledig aan te maken. Dit idee werd snel achterwege gelaten, omdat de wanden (bijna) altijd gedeeld worden door twee cellen. Als reeds één cel is gemaakt, is het moeilijk om bij een tweede cel te detecteren of een wand reeds gemaakt is voor een andere cel, of als deze nieuw moet worden gemaakt. We zouden alle reeds bestaande cellen moeten aflopen op zoek naar een buurcel, en dan de wand opvragen die kan worden gedeeld met de nieuwe cel. Indien zulke wand niet wordt gevonden, moet deze nieuw worden aangemaakt. Bovendien kan het gebeuren dat de bestaande wand moet worden opgesplitst. Een vereiste van de convexe celstructuur is dat elke wand (portal) *volledig* wordt gedeeld door twee buurcellen, en niet slechts voor een stuk. We hebben dit in zijaanzicht geïllustreerd in figuur4.4.

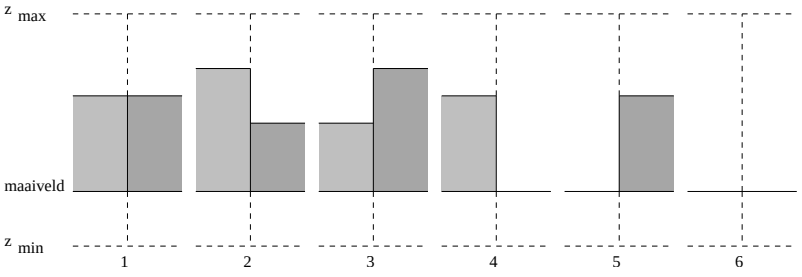


Figuur 4.4: bestaande wanden kunnen worden opgesplitst bij aanbouw van buurcel

Om dit probleem te vermijden, hebben we een andere aanpak gezocht en gevonden. We zullen namelijk eerst alle *"horizontale" (niet-verticale)* wanden en cellen aanmaken, en reeds deze wanden aan de juiste cellen koppelen. Hiermee zijn reeds alle hoekpunten (`Vertex3`) voor de wereld aangemaakt. Inderdaad, er is voor dit soort invoerdata geen enkele hoekpunt dat niet behoort tot een horizontale wand. Dit gegeven zal belangrijk zijn voor de aanmaak van de verticale wanden, dat tevens de volgende stap is.

In deze volgende fase, worden alle lijnstukken van de 2D triangulatie overlopen. Deze komen overeen met een aantal verticale wanden boven elkaar geplaatst. Er zijn een aantal mogelijke situaties, naargelang er gebouwen aan beide zijden staan en de hoogte van deze gebouwen. De zes verschillende mogelijkheden zijn in figuur 4.5 weergegeven.

Uiteindelijk moeten dan ook nog akoestische eigenschappen worden toegekend aan deze wanden. De bovenste en de onderste hebben telkens geen akoestische eigenschappen, het zijn open wanden. Voor de middelste wanden hebben we twee mogelijkheden. Als er aan beide zijden een gebouw is (grijs), dan zal aan beide zijden de binneneigenschappen van deze gebouwen worden toegekend. Dit is het geval voor situatie 1 en voor de onderste helften van situaties 2 en 3. Als er slechts aan één zijde een gebouw is, worden aan zowel binnen- als buiteneigenschappen toegekend. Dit is het geval voor de bovenste helften van situaties 2 en 3 en voor 4 en 5. Concreet betekent dat voor situatie 4 dat de binneneigenschappen aan de linkerzijde worden toegekend, en de buiteneigenschappen van het *linkse* gebouw aan de *rechterzijde*.



Figuur 4.5: zes verschillende situaties voor verticale wanden bij `PreProcessor2D5`. Situaties 1–3 hebben aan beiden zijden een gebouw en deze zijn even hoog voor situatie 1. Situaties 4 en 5 hebben slechts langs één zijde een gebouw en situatie 6 heeft geen enkel gebouw.

4.2 PreprocessorSHP

`PreprocessorSHP` is slechts een wrapper rond `Preprocessor2D5` om deze geschikt te maken voor shapefiles (*.shp) [10]. Dit is een eenvoudige niet-topologisch formaat voor het bewaren van geometrische informatie van geografische structuren. Het is een formaat afkomstig van ESRI en kan bekeken en bewerkt worden in ARCView 2/3. Om shapefiles in te lezen werd gebruik gemaakt van *Shapelib*, een subbibliotheek uit GDAL [15].

Intern heeft `PreprocessorSHP` een `Preprocessor2D5` die het eigenlijk werk doet. De `PreprocessorSHP` leest de verschillende obstakels uit een shapefile, en geeft deze door aan de `Preprocessor2D5` die ze verwerkt. Uiteindelijk geeft deze laatste het resultaat terug aan `PreprocessorSHP`.

Er zijn nog een aantal verschillen in de interface. Zo wordt er geen AABB meegegeven om de grootte van de wereld aan te duiden. De shapefile bevat namelijk reeds een 2D AABB voor de x en y informatie. We moeten dus wel nog de derde dimensie hieraan toevoegen met z_{min} en z_{max}. Omdat de 2D AABB uit de shapefile precies de ingesloten geometrie omsluiten, moeten we de AABB nog wat opblazen. Doen we dit niet, dan komen de gebouwen tot aan de rand van de wereld, en dit wensen we niet. Hiervoor geven we een extra parameter *scale* mee. Bemerk wel dat deze de 3D AABB herschaalt, dus ook z_{min}en z_{max}. Tenslotte kunnen we de informatie uit de shapefile ook nog verplaatsen met een 2D $offset$. Dit dient om de wereld meer naar de oorsprong te verplaatsen zodat we kleinere getallen voor de coördinaten krijgen, wat de nauwkeurigheid verhoogt.

4.3 World3Optimizer

De cellen die aangemaakt worden door `Preprocessor2D5` hebben allen een driehoekig grondvlak wegens de triangulator, en zijn dus driehoekige prisma's. Dit zijn doorgaans echter niet de grootst mogelijke convexe cellen om de wereld in op te delen. `World3Optimizer` zal dan ook proberen afzonderlijke cellen samen te voegen tot grotere convexe eenheden. Tegelijkertijd zal het proberen om wanden samen te smelten tot grotere polygonen. Het voordeel hiervan is dat er minder cellen zullen zijn, minder en grotere portals, waardoor de bundels minder opgesplitst geraken. En dat geeft alleszins een verbetering in nauwkeurigheid. Of het algoritme ook sneller zal zijn hangt van meerdere factoren af, en een vergroting van de cellen zou eventueel tot een vertraging van de bundeltrek kunnen leiden, zoals in sectie 6.2 zal worden uitgelegd.

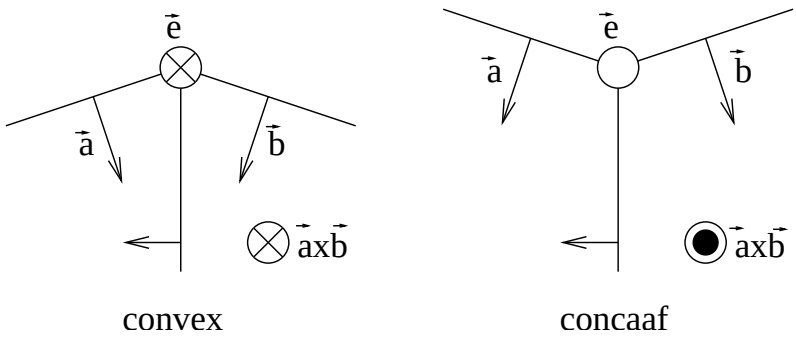
Het algoritme zal alle cellen aflopen en proberen deze samen te voegen met andere cellen. Dit zal het steeds opnieuw doen, tot er geen cellen meer kunnen worden samengevoegd. Als er twee cellen worden samengevoegd, zal het ook proberen wanden samen te voegen op een analoge manier.

Elke cel dat wordt gevraagd zich samen te voegen met buurcellen, zal al haar wanden aflopen. Ze zal daarbij een aantal voorwaarden controleren. En als al deze voorwaarden

zijn vervuld, zal de cel overgaan tot het eigenlijk samenvoegen met de buurcel. Op dat moment zal ze ook proberen om wanden van deze nieuwe cel samen te voegen op een analoge manier. Elke wand zal voor elke buur kijken of het een convex geheel kan vormen.

De voorwaarden voor twee cellen om zich samen te voegen zijn de volgende. Hierbij is `cell` de cel dat wordt gevraagd om zich samen te voegen, `candidateCell` de cel waarmee het zich probeert samen te voegen, en `candidateFace` de portal tussen beiden.

- `candidateFace` moet dubbelzijdig zijn. Anders kan er niet eens een `candidateCell` zijn.
- `candidateFace` moet immaterieel zijn. Het moet een open portal zijn zonder akoestische eigenschappen. `candidateFace` zal namelijk na de samensmelting verdwijnen, en er mag geen akoestische informatie verloren gaan.
- `cell` en `candidateCell` moet met hetzelfde medium gevuld zijn. Na samensmelting kan er slechts één medium overblijven, en we mogen opnieuw geen informatie verliezen.
- aan elke ribbe van `candidateFace` zullen de aanpalende wanden in `cell` en `candidateCell` een convexe hoek moeten maken. Dit betekent dat de binnenhoek kleiner moet zijn dan 180°. Dit is echter minder evident dan het lijkt. Het is in de ruimte namelijk vrij moeilijk om hoeken te bepalen. We kunnen echter een goede test vinden met behulp van een vector product en een scalair product. In figuur 4.6 zijn $\overrightarrow{a}$ en $\overrightarrow{b}$ de normaalvectoren van de aanpalende wanden, en deze zijn naar binnen gericht. De verticale lijn is `candidateFace` en heeft zijn normaal naar links gericht. $\overrightarrow{e}$ is de vector van de ribbe, en is in tegenwijzerzin volgens de normaal van `candidateFace` in het blad gericht. We kunnen nu zien dat als het vector product $\overrightarrow{a} \times \overrightarrow{b}$ ook in het blad is gericht, deze wanden een convexe hoek zullen vormen. In het andere geval is het concaaf. In de ruimte bestaat er echter geen *"blad"*. Daarom gebruiken we een scalair product om te bepalen hoe $\overrightarrow{a} \times \overrightarrow{b}$ gericht is. Als $\overrightarrow{e} \cdot \left(\overrightarrow{a} \times \overrightarrow{b}\right) > 0$, dan zijn beiden gelijk gericht en is de hoek convex.



Figuur 4.6: Bepalen van convexe en concave hoeken in `World3Optimizer`

Als aan alle voorwaarden voldaan zijn, dan worden de wanden van `candidateCell` bij `cell` gevoegd, en wordt `candidateFace` verwijderd. Daarna zal het proberen om een aantal wanden uit de nieuwe cel samen te voegen tot grotere gehelen. Dit gebeurt op een analoge manier, en opnieuw moeten er een aantal voorwaarden worden vervuld:

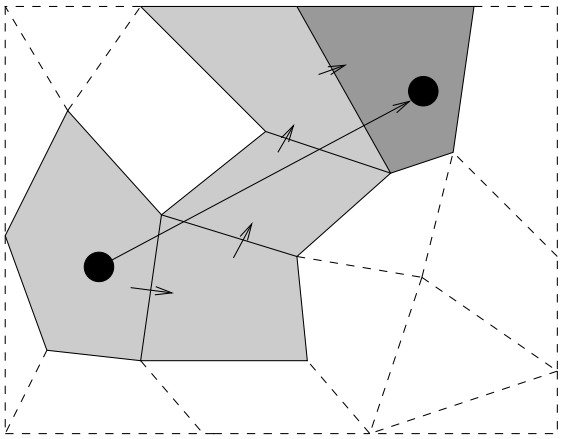
- Beiden wanden moeten coplanair zijn. Zo niet, dan kunnen die nooit samen één polygoon vormen.
- Ze moeten beiden dezelfde eigenschappen hebben en aan dezelfde cellen grenzen.
- Het geheel moet opnieuw convex zijn.

Als aan deze voorwaarden is voldaan, dan worden deze samengevoegd op een analoge manier als hierboven.

4.4 Cell3Finder

Er zijn een aantal situaties waarbij we willen weten in welke cel een gegeven punt zich bevindt. Daarbij denken we aan het invoegen van objecten (ontvangers) in de wereld, en aan het bundeltrekken van een puntbron. De methode die we hiervoor gebruiken is gerelateerd aan *Lawson's Oriented Walk* [20, 30] en aan het algoritme voorgesteld in [24].

Het algoritme start in een willekeurige cel waarin het één arbitrair punt kent. in `bass3` nemen we hiervoor het barycentrum. Het trekt een lijnstuk tussen dit punt en het eindpunt waarvan we willen weten in welke cel het gelegen is. Zolang de huidige cel dit eindpunt niet bevat, worden alle haar wanden afgelopen, op zoek naar deze die met het lijnstuk snijdt. De cel aan de andere zijde van deze wand wordt de nieuwe huidige cel.



Figuur 4.7: `Cell3Finder`: opzoeken van de cel dat een gegeven punt bevat.

In theorie is dit een algoritme dat altijd werkt, in de praktijk kan dit wegens numerieke onprecisie toch falen. Bij testen bleek echter dat het falingspercentage zeer klein is: vijf

falingen op 1 miljard. Nadien hebben we geen falingen meer gedetecteerd, op het triviale geval na dat het punt zich buiten de wereld bevindt. Toch werd een veiligheidsmechanisme ingebouwd om oneindige lussen te voorkomen. Zo zal het algoritme bij elke cel dat het passeert een uniek nummer per zoekopdracht achterlaten: `m_visitID`. Als het algoritme een cel betreedt waar dit nummer reeds werd achtergelaten, verondersteld het dat het in een oneindige lus is terechtgekomen, en geeft aldus een foutmelding.

Niettegenstaande dit een zeer eenvoudig algoritme is, is het toch vrij efficiënt: $O\left(\sqrt[d]{n}\right)$ met n het aantal cellen en d het aantal dimensies[1]. De efficiëntie kan drastisch worden verhoogd als voor opeenvolgende zoekopdrachten de eindpunten zeer dicht bij elkaar gelegen zijn. In dat geval zal een punt veelal in dezelfde cel liggen als de voorgaande. Als we dan iedere nieuwe zoekopdracht starten in de resulterende cel van de voorgaande, dan is de kans groot dat we direct de juiste cel gevonden hebben. In tabel 4.1 zien we dat het algoritme met geheugen nabijgelegen punten in constante tijd kan lokaliseren.

generatie punten	zonder geheugen		met geheugen	
n	11	184	11	184
random punten	$5.1s$	$31s$	$5.4s$	$31s$
nabijgelegen punten	$5.0s$	$30s$	$3.1s$	$3.1s$
identieke punten	$4.8s$	$31s$	$2.6s$	$2.9s$

Tabel 4.1: `Cell3Finder`: prestatie op Pentium 4 @ $2.5GHz$, 1,000,000 zoekopdrachten

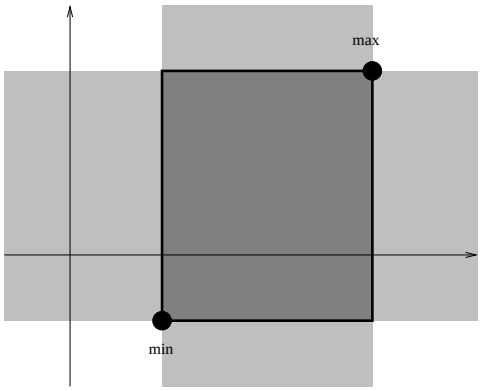
4.5 Cell3Bounder, World3Bounder

Deze algoritmes behoeven weinig uitleg. Ze zoeken de kleinste `AABB3` die respectievelijk een `Cell3` of een volledige `World3` bevat. `AABB3` is een driedimensionale AABB (*axis-aligned bounding box*, een balk met de assen volgens de coördinaatassen) [23]. Het voordeel van zo'n AABB is dat het kan worden beschreven aan de hand van slechts twee hoekpunten min en max, zoals geïllustreerd in figuur 4.8. Bovendien kan zeer snel worden getest of een punt al dan niet binnen de balk is gelegen. Elke regel bakent een band tussen twee uitersten af. Het punt moet tegelijkertijd binnen al deze banden liggen om tot de AABB te behoren.

$$P\begin{pmatrix} x \\ y \\ z \end{pmatrix} \in AABB \Leftrightarrow \begin{cases} x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \\ z_{min} \leq z \leq z_{max} \end{cases}$$

`Cell3Bounder` neemt een lege `AABB3` en voegt daaraan elk hoekpunt van elke ribbe van elke wand van een `Cell3` toe. Past het hoekpunt binnen de `AABB3`, dan hoeft deze niet te wijzigen. Ligt het punt buiten de `AABB3`, dan wordt de `AABB3` uitgebreid zodat deze toch

[1] Doorgaans is $2 \leq d \leq 3$, wegens het 2.5D karakter van vele 3D omgevingen



Figuur 4.8: een AABB wordt beschreven aan de hand van twee hoekpunten

binnen ligt. `World3Bounder` zal alle `AABB3`'s van al zijn cellen nemen en deze samensmelten tot één `AABB3` die alle cellen bevat, en dus ook de wereld.

Hoofdstuk 5

Implementatie van de bundeltrek

De module in `bass3` waar alles om draait is de eigenlijke bundeltrekker `Beam3Tracer`. `Beam3Tracer` zal in een gegeven `World3` omgeving de propagatie van een puntbron simuleren. Het zal daarbij enkel de bundeltrek zelf doen, de padgeneratie wordt overgelaten aan een `Responder`. Ook zal de bundeltrekker enkele vragen stellen aan de `Responder` in verband met de eigenschappen van materialen en de vordering van de simulatie.

5.1 Voorstelling van een bundel: Beam3 en Beam3Stack

5.1.1 Beam3

In het bundeltrek algoritme werken we niet met de volledige bundel van de bron tot de ontvanger. We gebruiken *deelbundels* die steeds in één enkele cel vervat zitten (sectie 3.1.4). Deze deelbundels kunnen veroorzaakt zijn door propagatie door een open portal, door reflectie op een volle wand, of door diffractie op een ribbe. We moeten zo verschillende deelbundels achtereen plaatsen om van de bron tot de ontvanger te komen. Zo'n deelbundel noemen we hier `Beam3`, en `Beam3Stack` zal de deelbundels achtereen plaatsen op een stapel. In het bundeltrek algoritme zal er steeds slechts één zo'n stapel zijn omdat we *"diepte - eerst" (depth-first)* werken. Wat dit betekent, verklaren we in sectie 5.2.

```
class Beam3
{
    Beam3Type m_type;
    Beam3Event m_event;
    unsigned m_generation;

    Point3 m_pointSource;
    LineSeg3 m_lineSource;
    Edge* m_sourceEdge;
```

```
        const Cell3* m_cell;
        const Face3* m_face;
        Polygon3 m_facePart;
        std::vector<ClipPlane3> m_clipPlanes;

        Beam3Stack* m_stack;
        unsigned m_instances;
    };
```

Eerst en vooral houden we in `Beam3` informatie bij over zijn plaats en functie in de `Beam3Stack`. `m_type` vertelt ons de vorm van de bundel, `m_event` vertelt wat de bundel heeft veroorzaakt en `m_generation` houdt bij op welk niveau de bundel in de `Beam3Stack` zit.

`Beam3Type`	beschrijving
`btOmnidirectional`	Omnidirectioneel en nog niet afgebakend door snijvlakken. De bron is puntvormig.
`btPyramid`	Afgebakend door een piramide en heeft nog steeds een enkel punt als bron
`btDiffracted`	Lijnvormige bron, en is afgebakend door een tentvorm.

`Beam3Event`	beschrijving
`beSource`	Veroorzaakt door een puntbron.
`beImmaterial`	Veroorzaakt door propagatie door een open portal, zonder dat dit het geluidspad beïnvloedt.
`beSpecReflection`	Veroorzaakt door speculaire reflectie op een wand. Dit is slechts de eerste bundel na de reflectie. De volgende kan opnieuw `beImmaterial` zijn.
`beDiffraction`	Veroorzaakt door diffractie op een ribbe. Opnieuw slechts de eerste bundel na diffractie.

Tabel 5.1: `Beam3Type` en `Beam3Event`

Vervolgens stockeren we de bron van de bundel. Dit kan zowel een echte als een virtuele bron zijn, en stelt steeds de top van de bundel voor. Dit is waar de snijvlakken van de bundel samen komen. We kunnen zowel puntbronnen `m_pointSource` als lijnbronnen `m_lineSource` bijhouden. Deze lijnbronnen zullen cruciaal zijn voor het implementeren van diffractie. Niet-virtuele lijnbronnen zullen steeds een deel zijn van een ribbe, en voor de verdere nauwkeurigheid van het algoritme houden we ook een pointer `m_sourceEdge` naar die ribbe bij.

Daarna specificeren we wat de binnenkant van de bundel is. Dit is steeds een deel van één cel waarnaar we een pointer `m_cell` bijhouden. Daarnaast zal een bundel – veroorzaakt door `beImmaterial` of `beSpecReflection` – steeds door precies één wand de cel bin-

nenkomen. De (virtuele) bron ligt dan buiten de cel, langs de andere kant van deze wand. We houden een pointer `m_face` naar deze wand bij, en ook een polygoon `m_facePart` bij met het stuk van deze wand dat binnen de bundel ligt. Er zullen nog andere wanden van de cel voor een stuk binnen de bundel liggen, maar dit zijn "uitgaande" wanden, waar de bundel de cel verlaat.

Uiteraard houden we ook de snijvlakken van de bundel bij in `m_clipPlanes`. Deze vormen het manteloppervlak van bundel-piramide.. Alhoewel we het vlak van `m_facePart` ook als een snijvlak zouden kunnen zien, wordt deze hier niet nog eens gestockeerd. Voor elk snijvlak houden we ook twee pointers bij, één naar een `Edge3` en één naar een `Face3`. Beiden zijn normaal *null* en verwijzen nergens naar. Maar soms is wegens constructie met zekerheid geweten dat een ribbe of een wand in het vlak van het snijvlak ligt. Dan zal dit bij worden gehouden via deze pointers. Dit zal cruciaal blijken te zijn voor de implementatie van diffractie. Zo worden deze snijvlakken met een structuur als volgt gestockeerd:

```
struct ClipPlane3
{
    Plane3 plane;
    const Edge3* edge;
    const Face3* face;
};
```

Wat we hier nog wensen te vermelden, is dat voor een `Beam3` niet alle combinaties voor de data geldig zijn. We geven hier een tabel met de mogelijke combinaties. Ze zullen wellicht niet allen duidelijk zijn, maar we komen hierop terug in sectie 5.2.

`Beam3Type`	`Beam3Event`	generatie	bron	wand	snijvlakken
`btOmnidirectional`	`beSource`	$=0$	puntbron	nee	nee
`btPyramid`	`beImmaterial`	>0 (of $=0$[1])	puntbron	ja	ja
	`beSpecReflection`	>0	puntbron	ja	ja
`btDiffracted`	`beDiffracted`	>1	lijnbron	nee	ja
	`beImmaterial`	>2	lijnbron	ja	ja
	`beSpecReflection`	>2	lijnbron	ja	ja

Tabel 5.2: voorwaarden voor een geldige `Beam3`

5.1.2 Beam3Stack

Op elk moment zal er tijdens de bundeltrek één enkele stapel van bundels zijn: één `Beam3Stack`. Deze stapel bevatten alle relevante bundels vanaf de bron tot aan de huidige bundel. Helemaal onderaan de stapel bevindt de eerste bundel en is door de bron veroorzaakt. Ze is dus het type `beSource` en `C`-gewijs zeggen we dat dit de *nulde* generatie bundel is.

Het op de bundel plaatsen en er terug afhalen wordt gereguleerd door een *smart pointer* `Beam3_p`. Net zoals een ruwe `C` pointer (`Beam3*`) verwijst deze naar een `Beam3` object, maar zal bij constructie deze `Beam3` op een gegeven `Beam3Stack` plaatsen. Daarna kan deze `Beam3_p` pointer gekopieerd worden, en zal deze bijhouden hoeveel kopieën er zijn die naar dezelfde `Beam3` verwijzen. Tenslotte zal deze, als de laatste kopie verwijderd wordt, de `Beam3` terug van de stack halen en vernietigen. Om dit te implementeren worden de `m_stack` en `m_instances` leden van `Beam3` gebruikt.

Het grote voordeel van dit mechanisme is in eerste instantie het vermijden van geheugenlekken. Een ruwe `C` pointer kan verdwijnen zonder dat de *pointee* (object naar waar verwezen wordt) verwijderd wordt. Een smart pointer als deze zal steeds zorgen dat het gealloceerde geheugen terug wordt vrijgegeven. Deze techniek wordt door Stroustrup [27] beschreven in het kader van RAII*, resource acquisition is initialization (het verwerven van de resource is initialisatie).* Bovendien zal deze `Beam3_p` er voor zorgen dan een `Beam3` bundel slechts tot één `Beam3Stack` kan behoren.

5.1.3 Constructie van een bundel

class `Beam3` heeft een verscheidenheid aan constructors. Elk van deze zijn gespecialiseerd in het opbouwen van één enkele bundel aan de hand van enkele gegevens. Meer bepaald zullen zij de snijvlakken van de bundel bepalen. Hier volgt een bespreking van deze constructors met hun signatuur.

Omnidirectionele bundel

```
Beam3(const Cell3* a_cell, const Point3& a_source);
```

Deze constructor maakt een omnidirectionele bundel aan. Z'n type is `btOmnidirectional`. Dit is een bundel met een punt als bron en waarvan het zichtbaarheidsgebied nog niet is afgebakend door snijvlakken. Het zal altijd als eerste bundel worden aangemaakt als de bundeltrekker een omnidirectionele bron moet simuleren. Het is dus altijd een bundel van de nulde generatie en `beSource` als event hebben.

De cel van een omnidirectionele bundel zal steeds de top van de bundel bevatten. Zou dit niet zo zijn, dan zou een beperkende piramide kunnen worden geconstrueerd die niet meer omnidirectioneel is en toch de volledige cel bevat. Bij een omnidirectionele bundel zal er dus ook geen sprake zijn van een wand waardoor de bundel de cel binnenkomt.

Piramidale bundel

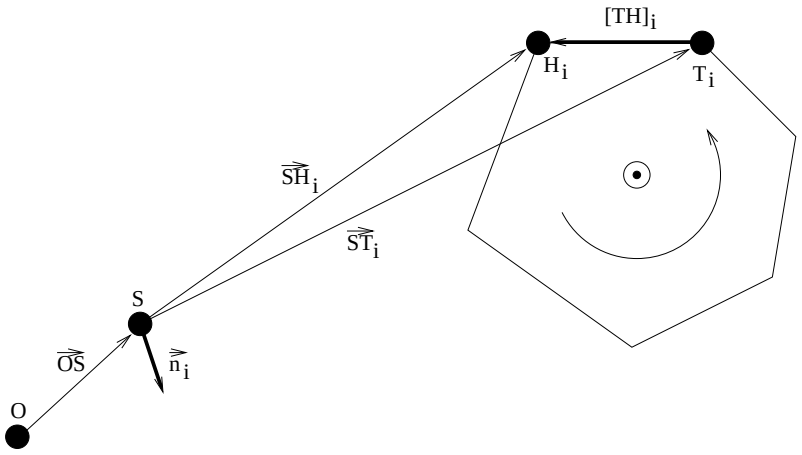
```
Beam3(const Cell3* a_cell, const Point3& a_source,
      const Polygon3& a_crossSection);
Beam3(Beam3Event a_event,
```

```
        const Cell3* a_cell, const Point3& a_source,
        const Face3* a_face);
Beam3(unsigned a_generation, Beam3Event a_event,
        const Cell3* a_cell, const Point3& a_source,
        const ClippedFace3& a_face, Real a_contraction);
```

Een piramidale bundel (`btPyramide`) is niet langer omnidirectioneel en heeft nu snijvlakken die het zichbaarheidsgebied begrenzen. Er zijn drie verschillende constructors, naar gelang de oorzaak van de bundel.

Een eerste constructor wordt gebruikt voor een bron die niet omnidirectioneel is, maar waarvan de zichtbaarheid a priori beperkt is door piramide. Deze wordt gebruikt in de visualisatiedemo als camera (sectie 5.3). Het zicht van zo'n camera is inderdaad beperkt tot het scherm. Verder is deze bundel identiek aan een omnidirectionele bundel: het is steeds van de nulde generatie en heeft als event `beSource`. De cel zal opnieuw de bron bevatten en er is opnieuw geen sprake van een wand waardoor de bundel de cel binnenkomt.

De tweede en derde worden gebruikt voor bundels veroorzaakt door de bundeltrek zelf (de generatie van deze bundels moet bijgevolg steeds groter zijn dan 0 en ze kunnen nooit `beSource` als event hebben). Als een omnidirectionele bundel op een portal (`Face3`) invalt, dan weten we dat de portal steeds volledig zichtbaar zal zijn. Er zijn immers geen snijvlakken van de bundel die het zicht kunnen beperken. In dat geval wordt de tweede constructor gebruikt die de volledige portal aanneemt. Als er daarentegen een piramidale bundel op een portal invalt, dan is het mogelijk dat deze slechts voor een deel zichtbaar is. Dit deel wordt bepaald (`ClippedFace3`) en de derde constructor wordt dan gebruikt. Omdat de tweede constructor enkel gebruikt kan worden als de vorige bundel omnidirectioneel was, zal deze steeds een bundel van generatie 1 creëren. De derde constructor kan bundels van elke generatie verschillend van 0 aanmaken.



Figuur 5.1: constructie van snijvlakken bij piramidale bundels

Algorithm 6 constructie van snijvlakken bij piramidale bundels

S = source of beam, top of pyramid;
for (each edge $[TH]_i$ of portal)
{
    $\overrightarrow{n_i} = \overrightarrow{SH_i} \times \overrightarrow{ST_i}$;
    $d_i = -\overrightarrow{n} \cdot \overrightarrow{OS}$;
    $\Sigma_i \leftrightarrow \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i = 0$;
}

In elke van de drie gevallen wordt uiteindelijk de bron en de doorsnede van de bundel meegegeven aan de constructor. Bij de eerste is deze doorsnede een polygoon (`Polygon3`) die bijvoorbeeld de omtrek van het scherm voorstelt. Bij de tweede is dit een volledige wand (`Face3`), en bij de derde een deel van een wand (`ClippedFace3`). In elk geval is deze doorsnede polygonaal, en de constructor zal snijvlakken creëren door de bron S en elk van zijn ribben. Dit is geïllustreerd in figuur 5.1 en algoritme 6. De ribben worden (ten opzichte van S) in tegenwijzerzin doorlopen. Van de ribbe wordt telkens staart T_i en kop H_i bepaald zodat we precies drie punten van het snijvlak kennen. De normaalvector $\overrightarrow{n_i}$ wordt dan bepaald zodat deze naar de binnenkant van de bundel is gericht. En tenslotte wordt geëist dat S in dit snijvlak ligt, wat ons een waarde voor d_i oplevert. Het resultaat is een vlak dat wordt gestockeerd in de `ClipPlane3` structuur.

Uiteindelijk zijn er nog twee uitbreidingen waarmee moet worden rekening gehouden:

- Bij een `ClippedFace3` zijn er een aantal ribben die veroorzaakt zijn door een snijvlak van de vorige bundel. We kunnen deze situatie herkennen doordat deze ribben van het type `tPlane` i.p.v. `tPair` zijn. Van zo'n ribbe weten we reeds dat het zicht in het vlak van dit snijvlak bevindt. Als ook de bron van de vorige bundel overeenkomt met deze van de nieuwe bundel, dan zal het nieuw snijvlak precies hetzelfde zijn (ze hebben immers minstens drie punten gemeen). In dat geval zullen we het nieuw snijvlak niet opnieuw berekenen zoals hierboven vermeld, maar zullen we het origineel snijvlak kopiëren. Zo zijn we zeker dat ze ook precies gelijk zijn, en dat er geen afrondingsfouten zijn geïntroduceerd door een nieuwe berekening. Dit kan enkel bij de constructie van transmissie-bundels. Bij reflectie-bundels zullen de bronnen immers niet op dezelfde plaats liggen, de nieuwe bron is gespiegeld ten opzichte van de portal. We kunnen echter nog steeds voordeel halen uit de kennis van het origineel snijvlak door deze ook te spiegelen en het resultaat te gebruiken als nieuw snijvlak.

- Als een ribbe van de doorsnede overeenkomt met een originele ribbe (`Edge3`) van een wand (`Face3`), dan zullen we deze onthouden. Dit is het geval voor elke ribbe in de tweede constructor (de doorsnede *is* een `Face3`), en in derde constructor voor alle ribben van het type `tPair`. Naar deze ribbe wordt een pointer bij gehouden in de `ClipPlane3` structuur. Dit zal nodig zijn om het clipping algoritme exacter te laten functioneren.

Diffractie-bundels

Er zijn twee soorten diffractie bundels. Deze die gecreëerd zijn door diffractie aan een ribbe (event `beDiffraction`), en deze gecreëerd door de verdere propagatie van diffractie bundels. Voor beiden bestaan afzonderlijke constructors.

```
Beam3(unsigned a_generation,
      const LineSeg3& a_source, const Edge3* a_edge,
      const Plane3& a_frontPlane, const Plane3& a_backPlane,
      const Face3* a_backFace,
      Real a_coneTail, Real a_coneHead);
Beam3(unsigned a_generation, Beam3Event a_event,
      const Cell3* a_cell,
      const LineSeg3& a_source, const Edge3* a_edge,
      const ClippedFace3* a_face, Real a_contraction,
      Real a_coneTail, Real a_coneHead);
```

Hoe de eerste soort ontstaat door diffractie aan een ribbe is uitgelegd in sectie 2.4.1. `a_edge` verwijst naar de ribbe waaraan de diffractie optreedt en `a_source` is het deel van deze ribbe dat zichtbaar is voor de vorige bundel die de diffractie veroorzaakt. Het is een lijnstuk, wat aanduidt dat we nu met een lijnbron te maken hebben. `a_frontPlane` en `a_backPlane` zijn beide vlakken uit sectie 2.4.1 die de schaduwzone afbakenen. `a_frontPlane` is het vlak tussen de zichtbare zone en de schaduwzone. `a_backPlane` is het vlak tussen de schaduwzone en de achterzijde van het obstakel. `a_backPlane` komt steeds overeen met het vlak van een `Face3`. We zullen een pointer `a_backFace` naar deze wand bijhouden om in het clipping algoritme er voordeel uit te halen (zie sectie 5.4) . Deze pointer wordt opnieuw ook de `ClipPlane3` structuur bewaard.

De tweede soort ontstaat door propagatie van een diffractie-bundel. De constructor hiervoor is zeer gelijkaardig aan de laatste van de piramidale bundels, met dat verschil dat we nu een lijnbron i.p.v. een puntbron in rekening moeten brengen. Concreet betekent dit dat voor elke ribbe $[TH]_i$ er twee snijvlakken kunnen worden geconstrueerd: één voor elk van de eindpunten (S_T en S_H) van `a_source`. Telkens moet dat snijvlak worden geselecteerd waarbij het andere eindpunt achter dat vlak ligt (zie sectie 2.4.1). We doen dit door eerst het vlak voor S_T te berekenen. Als S_H dan voor dit vlak ligt, dan herberekenen we een nieuw vlak met S_H. Als S_H in het vlak door S_T ligt, dan zijn beide vlakken gelijk en is het om het even met welk eindpunt het vlak wordt geconstrueerd. Extra aandacht moet gaan naar de situatie waarbij T_i, H_i en S_T colineair zijn. In dit geval is het niet mogelijk om één vlak door deze drie punten te construeren ($\overrightarrow{n}$ zal $\overrightarrow{0}$ worden). We moeten dan automatisch S_H gebruiken. Verder gelden dezelfde uitbreidingen als bij piramidale bundels.

Algorithm 7 constructie van snijvlakken bij diffractie-bundels

S_T = tail of a_source;
S_H = head of a_source;
for (each edge $[TH]_i$ of portal)
{
    $\overrightarrow{n_i} = \overrightarrow{S_T H_i} \times \overrightarrow{S_T T_i}$;
    $d_i = -\overrightarrow{n} \cdot \overrightarrow{OS_T}$;
    if ($\overrightarrow{n_i} \cdot \overrightarrow{OS_H} + d_i > 0$ or $\overrightarrow{n} = \overrightarrow{0}$)
    {
        $\overrightarrow{n_i} = \overrightarrow{S_H H_i} \times \overrightarrow{S_H T_i}$;
        $d_i = -\overrightarrow{n} \cdot \overrightarrow{OS_H}$;
    }
    $\Sigma_i \leftrightarrow \overrightarrow{n_i} \cdot \overrightarrow{OP} + d_i = 0$;
}

5.2 Implementatie van de bundeltrek: Beam3Tracer

Nu we weten hoe de omgeving wordt voorgesteld, en hoe verschillende bundels worden bewaard in het geheugen, kunnen we overgaan tot de behandeling van de component van `bass3` waar het allemaal om te doen is: `class Beam3Tracer`. Het bevat het algoritme dat instaat voor het eigenlijke bundeltrekken. Het start vanuit een puntbron en genereert bundels tot bepaalde criteria worden bereikt.

5.2.1 De bundeltrek functies.

`Beam3Tracer` volgt ongeveer dezelfde stappen als het algoritme 5 in sectie 3.2.2. Er zijn drie verschillende *trace* functies die een bundel `Beam3_p` aannemen en deze verder door rekenen: `traceOmnidirectional`, `tracePyramid` en `traceDiffracted`. Ze nemen elk één van de drie types `btOmnidirectional`, `btPyramid`, `btDiffracted` aan. De verschillen tussen de drie zitten voornamelijk in het uitbuiten van voorkennis over het type. Ruwweg volgen ze alle drie dezelfde stappen. Een generiek grondplan van deze functies is geïllustreerd in algoritme 8.

Eerst en vooral worden alle ontvangers in de bundel behandeld door `sendObjectHits`. Deze zal voor elke zichtbare ontvanger een `objectHit` naar de `Responder` sturen (we zullen deze `Responder` onmiddellijk behandelen in sectie 5.3). De `Responder` stuurt een boolse waarde terug dat aanduidt of de ontvanger al dan niet relevant is. `sendObjectHits` geeft op zijn beurt ook een boolse waarde af, en die is *waar* als er minstens één `objectHit` *waar* is of als er geen enkel object in de bundel ligt[2]. Op deze manier krijgt `traceX` informatie

[2]Als er geen enkel object in de bundel ligt, dan wordt er geen enkele `objectHit` naar de `Responder` verzonden. Dat wil zeggen dat de `Responder` via geen kans om via een `objectHit` te melden of de bundel al dan niet relevant is. We mogen de bundel dan hier niet afbreken, omdat we niet over voldoende informatie over de relevantie van de bundel beschikken. Daarom geeft in dat geval `sendObjectHits` automatisch *waar* af, zodat de bundeltrek wordt verder gezet.

Algorithm 8 grondplan van trace functie in Beam3Tracer

```
void Beam3Tracer::traceX(Beam3_p a_beam)
{
    if (sendObjectHits(a_beam) == false) return;

    for (each face of cell)
    {
        if (face is black) continue;

        clip face to beam;
        if (face is completely invisible) continue;

        if (sendFaceHit(face, a_beam) == false) continue;

        if (face is open portal)
        {
            Beam3_p newBeam = construct transmission beam;
            traceX(newBeam);
            if (diffraction is possible)
            {
                diffractOnFace(face, newBeam);
            }
        }

        if (face is closed portal)
        {
            Beam3_p newBeam = construct reflection beam;
            traceX(newBeam);
        }
    }
}
```

over de relevantie van de bundel. Als `sendObjectHits` *waar* af geeft, dan wordt de bundel als relevant beschouwd en wordt de bundeltrek verder gezet. In het andere geval wordt de recursie hier afgebroken.

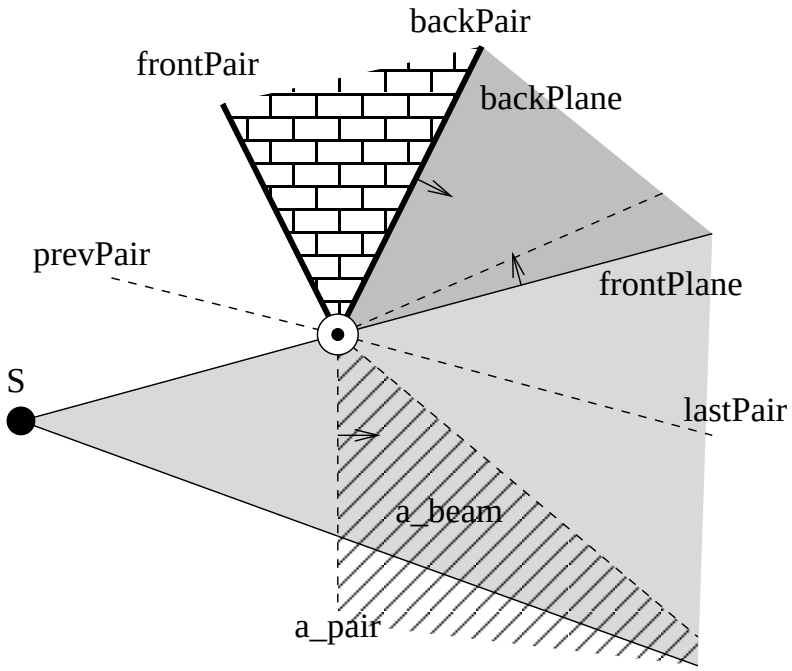
Daarna probeert `traceX` om nieuwe bundels aan te maken. Het doet dit voor elke wand van de cel. Als een wand geen portal is (zoals aan de rand van de wereld), of als het een alles absorberende wand is, dan beschouwen we deze als *black* en wordt er geen nieuwe bundel gecreëerd. Vervolgens wordt bepaald welk deel van de wand zichtbaar is door de bundel. Als de wand volledig onzichtbaar is, kan er geen nieuwe bundel worden gecreëerd door deze wand, en gaat `traceX` over naar de volgende wand. Daarna brengen we de `Responder` op de hoogte van een `FaceHit`, en deze zal vertellen of de wand al dan niet relevant is. Indien ze dit niet is, dan wordt er opnieuw geen bundel gecreëerd. Tenslotte zal voor een open portal een transmissie-bundel worden gecreëerd en gecontroleerd of er diffractie kan ontstaan. Bij een volle portal zal er reflectie optreden, en wordt er een

reflectie-bundel gecreëerd.

Diffractie kan enkel aan open portals optreden, en enkel als er aan één van de ribben een obstakel grenst. Of deze voorwaarde voor een portal is vervuld, wordt met een vlag in de wand bijgehouden. Deze wordt getest alvorens verder te gaan met diffractie. Zo kunnen we dure operaties vermijden bij portals waaraan helemaal geen diffractie kan optreden. `diffractOnFace` zal voor elke ribbe van de portal dat overeenstemt met een originele `Pair3` ribbe[3] proberen om een diffractie-bundel te construeren. Het doet dit door voor deze ribben de functie `diffractOnPair`[4] aan te roepen.

5.2.2 Diffractie

Hoe diffractie kan worden geïmplementeerd m.b.h.v. convexe cellen, werd reeds besproken in sectie 2.4.1. We gaan hier verder in op de implementatiedetails, specifiek voor `Beam3Tracer`. Vooraleer we overgaan tot een bespreking van het algoritme (9) zullen we eerst een aantal punten vooropstellen.



Figuur 5.2: diffractie aan een ribbe in bovenaanzicht, centraal staat de ribbe waaraan diffractie optreedt.

- We gaan uit van een bundel `Beam3_p a_beam`, een ribbe `Pair3 a_pair` waaraan diffractie kan optreden, en `LineSeg3 a_pairPart` die het deel van de ribbe bevat

[3]Indien de portal van het type `Face3` is, dan is elke ribbe een `Pair3` ribbe. Indien de portal een `ClippedFace3` is, dan zijn het enkel de ribben van het type `tPair` die in aanmerking komen.

[4]We geven niet direct de `Edge3` door aan `diffractOnPair`, maar de `Pair3` van de portal die leidt naar `Edge3`. Zo behouden we nog kennis van de `Face3` waaruit de portal bestaat. Vandaar de naam `diffractOnPair`.

dat zichtbaar is door `a_beam`. We veronderstellen dat er tenminste een deel van de ribbe zichtbaar is, anders zou er geen diffractie kunnen optreden.

- Een `Pair3` verwijst naar zowel een `Edge3` als een `Face3`d. Als parameter is `a_pair` gebruikt om een ribbe `Edge3` aan te duiden. In het algoritme zullen we daarentegen de `Pair3`'s als wanden (`Face3`) behandelen. De ribbe zelf wordt aangeduid met `Edge3 edge`. We doen dit omdat we alle wanden rond de ribbe nodig hebben, en hiervoor is het middel bij uitstek de lijst van `Pair3`'s rond de ribbe. In figuur 5.2 is `edge` centraal geplaatst, loodrecht op het blad. Rond `edge` liggen dan de verschillende wanden, aangeduid met *pairs*.

- `a_beam` is niet de bundel die invalt op de portal waaraan diffractie ontstaat, maar de transmissie-bundel is die reeds door de portal is geconstrueerd. Dit is omdat we de zichtbaarheid van de bron door de portal wensen te kennen. Dit is enkel mogelijk met de transmissie-bundel, omdat deze snijvlakken door de randen van de portal heeft geconstrueerd. De cel van `a_beam` is ingekleurd met een diagonale arcering. Dit heeft belangrijke consequenties naar de oriëntatie toe. We bekijken nu `a_pair` langs achter (de normaalvector is naar rechts getekend). Dit betekent dat `edge` uit het blad is gericht, en dat spin en counterspin omheen de ribbe overeenkomt met respectievelijk tegenwijzerzin en wijzerzin.

- Als we testen of iets al dan niet in `a_beam` ligt, zullen we niet eisen dat het ook tot de cel van `a_beam` moet behoren (schuine arcering). We zullen enkel eisen dat het tot de bundel-piramide behoort (lichtgrijs zone). We doen dit omdat het probleem van diffractie zich over meerdere cellen uitstrekt.

In het algoritme 9 wordt eerst en vooral `prevPair` bepaald. Dit is de ribbe die we in counterspin richting vinden van `a_pair`. Een eerste voorwaarde opdat er diffractie kan optreden aan de ribbe is dat `prevPair` buiten de bundel ligt (`prevPair is not seen by a_beam`). Stel dat `prevPair` toch binnen de bundel zou liggen, dan betekent dit dat om tot de huidige bundel te komen, we reeds een bundel doorheen `prevPair` hebben gemaakt. De bron S van `a_beam` ligt immers aan de andere zijde van `prevPair`. Dat betekent dat er reeds op dit moment diffractie is optreden aan de ribbe. `prevPair` grenst inderdaad ook aan `edge`. Als dat zo is, dan mogen we niet nog eens diffractie veroorzaken. Bedenk dat de open portals fysisch niet aanwezig zijn, we mogen slechts één maal diffractie rond dezelfde `edge` veroorzaken.

Vervolgens wordt naar het obstakel gezocht die de diffractie veroorzaakt. Daarvoor wordt aan `a_pair` gestart en zowel in spin als in counterspin richting gezocht naar de eerste `Pair3` die een volle portal als wand heeft. In spin richting wordt dit `backPair`, en in counterspin richting `frontPair`. Kan één van beiden niet worden gevonden[5], dan kan er

[5]Het kan eigenlijk niet gebeuren dat slechts één van beiden wordt gevonden. Ofwel worden zowel `frontPair` als `backPair` gevonden, ofwel geen van beiden. Toch wordt voor de veiligheid op beiden afzonderlijk getest, omdat we eisen dat *beiden* aanwezig zijn voor diffractie.

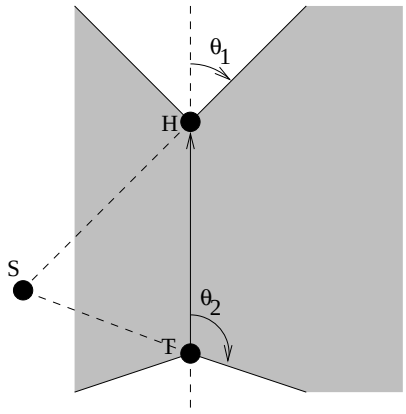
Algorithm 9 diffractie aan een ribbe

```
void Beam3Tracer::diffractOnPair(Beam3_p a_beam, const Pair3* a_pair,
                                 const LineSeg3& a_pairPart)
{
    edge = a_pair's edge;

    prevPair = pair at counterspin side of a_pair;
    if (prevPair is seen by a_beam) return;

    find frontPair;
    if (frontPair not found) return;
    if (frontPair is seen by a_beam) return;

    find backPair;
    if (backPair not found) return;
    if (backPair is seen by a_beam) return;

    determine cosTail and cosHead;
    frontPlane = plane through source and a_pairPart;
    backPlane = plane of backPair's face;

    find lastPair;
    newBeam = diffracted beam with frontPlane,
              backPlane, cosTail and cosHead;
    for (all cells between lastPair and backPair)
    {
        set newBeam's cell;
        traceDiffracted(newBeam);
    }
}
```

geen diffractie optreden aan de ribbe. Ook moet dit obstakel volledig buiten `a_beam` liggen om diffractie te kunnen veroorzaken. Zou dit niet zo zijn, dan zou het obstakel de bundel tegenhouden. Als dus `frontPair` of `backPair` in de bundel ligt, dan wordt de functie verlaten.

Eens dit alles bepaald is, is het zeker dat er diffractie optreedt aan de ribbe. Dan moeten beide kegels bepalen waartussen alle mogelijke diffractie stralen vervat zitten (zie sectie 2.4). Hun toppen zijn reeds bekend (de eindpunten van `a_pairPairt`) zodat enkel nog hun halve openingshoeken θ_1 en θ_2 berekend moeten worden, en meer bepaald de cosinus van deze hoeken. Figuur 5.3 toont ons dat $\theta_1 = \overrightarrow{TH} \wedge \overrightarrow{SH}$ en $\theta_2 = \overrightarrow{TH} \wedge \overrightarrow{ST}$. Tenslotte zijn nog de twee snijvlakken van de schaduwzone nodig: `frontPlane` en `backPlane`. `frontPlane` is het vlak door `a_pairPart` en de bron, en `backPlane` is het draagvlak van `backPair`.

Met deze gegevens wordt dan een diffractie-bundel aangemaakt, maar de cel waartoe ze behoort wordt nog niet ingevuld. Dit komt omdat de bundel in verschillende cellen moet



Figuur 5.3: bepaling van de kegels bij diffractie

starten. Voor elke cel rond `edge` die voor een gedeelte in de schaduwzone ligt, moet de diffractie-bundel worden ingezet. Daarvoor dient `lastPair`. Het is de laatste `Pair3` vanaf `a_pair` in spin richting die niet in de schaduw ligt. Eventueel kan dit `a_pair` zelf zijn. Alle cellen tussen `lastPair` en `backPair` liggen tenminste voor een deel in de schaduw, en voor elk van deze cellen wordt de diffractie-bundel door gerekend met `traceDiffracted`.

5.3 Afhandeling van resultaten: Responder

Eén van de doelstellingen van `bass3` was om grote flexibiliteit te behouden. Dat is nodig als je de bundeltrekker in variërende toepassingen wil kunnen inzetten. Zo hebben we de bass3 reeds gebruikt om een visualisatie-demo te geven van de *"Zuid"* omgeving. Om dit te bereiken hebben we het *strategy* patroon toegepast [2, 14] die alle probleemspecifieke onderdelen van de bundeltrekker `Beam3Tracer` scheidt. Wat zijn deze onderdelen?

- Eigenschappen van materialen. `Beam3Tracer` kent geen eigenschappen van materialen, enkel de geometrie van de omgeving. Hij kan dus niet uit zichzelf weten of er reflectie optreedt aan een portal en zal dit aan de `Responder` moeten vragen. De `Responder` controleert de eigenschappen van het materiaal van de portaal, en geeft een boolse waarde als antwoord.

- Stopcriteria. Het is noodzakelijk om er voor te zorgen dat `Beam3Tracer` niet eindeloos door gaat. Daarvoor dienen we stopcriteria op te leggen, en die zijn afhankelijk van de toepassing. Daarom wordt deze verantwoordelijkheid ook in de `Responder` ondergebracht. Een voorbeeld van zo'n stopcriterium is dat er slechts 8 reflecties mogen voorkomen in één `Beam3Stack`. Vooraleer `Beam3Tracer` een reflectie zal uitvoeren, zal hij aan de `Responder` vragen of hij dit mag doen. De `Responder` controleert de `Beam3Stack` en geeft opnieuw een bools antwoord.

- Afhandeling van de resultaten. Tijdens het bundeltrekker vind de `Beam3Tracer` zichtbare ontvangers en portals. Wat hij hiermee moet doen, is afhankelijk van de toepassing. In de visualisatie-demo (waar er geen ontvangers zijn) is enkel de detectie van portals van belang. De volle portals moeten op het scherm worden afgebeeld. In andere toepassingen zijn niet de portals maar de ontvangers van belang. Dit is het geval voor de hoofdtoepassing waarvoor `bass3` is geschreven. Voor elke ontvanger dat wordt gedetecteerd moet een geluidspad, een attenuatie en een faseverschil worden berekend. Deze toepassing wordt verder behandeld in hoofdstuk 6.

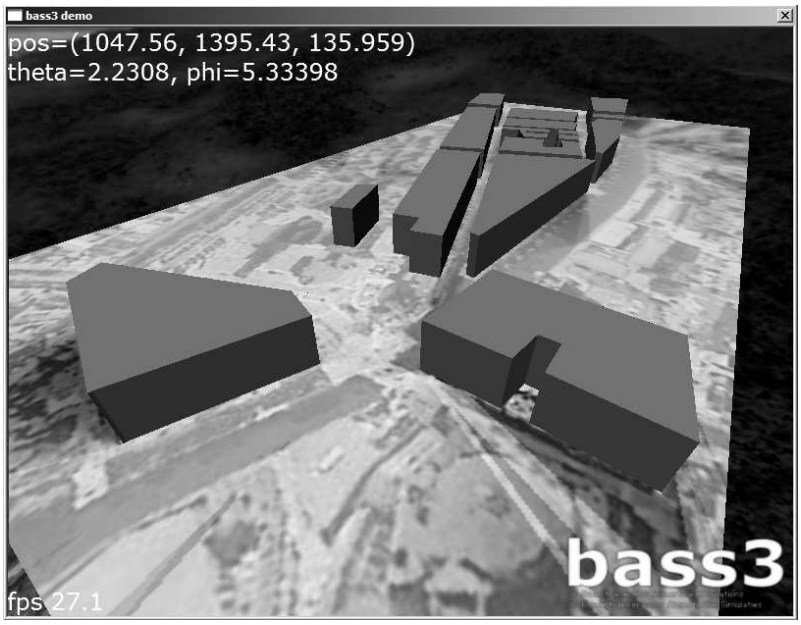
De interface van de `Responder` ziet er als volgt uit:

```
class Responder
{
public:
    virtual bool objectHit(const Beam3Stack* a_beamStack,
                           const Object3* a_object);
    virtual bool faceHit(const Beam3Stack* a_beamStack),
                         const Face3* a_face,
                         const Polygon3& a_faceArea);
    virtual bool isDiffractive(const Beam3Stack* a_beamStack,
                               const Face3* a_face);
    virtual bool isReflective(const Beam3Stack* a_beamStack,
                              const Face3* a_face);
    virtual bool isTransparent(const Beam3Stack* a_beamStack,
                               const Face3* a_face);
};
```

`objectHit` en `faceHit` vangen detecties van ontvangers en portals op. Bij `faceHit` wordt ook nog het zichtbare gebied van de portal `a_faceArea` door gegeven. Beiden hebben een boolse *return* waarde. Hiermee geven ze feedback aan de `Beam3Tracer`. *Waar* betekent dat de detectie relevant is, en *vals* dat de detectie van geen waarde is. Als een `faceHit` *vals* oplevert, dan weet `Beam3Tracer` dat het de bundeltrek doorheen deze portal kan stoppen. De return waarde van `objectHit` heeft dezelfde functie, maar de `Beam3Tracer` zal nu slechts beslissen op het resultaat van alle `objectHit`'s in één cel (zie sectie 5.2.1).

`isDiffractive`, `isReflective` en `isTransparent` zijn functies om `Beam3Tracer` te informeren over de eigenschappen van materialen. `isDiffractie` zal *waar* afgeven als aan de portal diffractie kan optreden, `isReflectie` als er reflectie kan optreden, en `isTransparent` als er transmissie mogelijk is. Bovendien krijgt de `Responder` nu de kans om stopcriteria op te leggen. Door het aantal reflecties in `a_beamStack` te tellen, kan het bepalen of er al dan niet nog een reflectie is toegestaan.

Een uitgewerkte `Responder` wordt besproken in hoofdstuk 6.



Figuur 5.4: de *"Zuid"* omgeving in een visualisatie-demo van `bass3`

5.4 Stabiliteit

Als men geometrische algoritmes numeriek gaat uitvoeren, dan is de grootste vijand de eindige precisie waarmee getallen in een computer geheugen worden opgeslagen. Geometrische algoritmes zijn meestal ontwikkeld in de veronderstelling dat alle getallen met oneindige precisie kunnen worden voorgesteld, in een zogenaamd *Real RAM*. Theoretisch wordt de consistentie tussen numerieke en combinatorische data gegarandeerd door het onderliggend algoritme. Maar in de praktijk, als men vlottende komma getallen gaat gebruiken, dan zullen afrondingsfouten deze consistentie snel doorbreken. Dit kan leiden tot verkeerde beslissingen in het algoritme, waardoor de uitkomst foutief kan zijn. Bijvoorbeeld een test die bepaalt of een punt al dan niet in een gegeven vlak ligt, zal controleren of de uitkomst van een vergelijking nul is. Allerlei afrondingsfouten kunnen ervoor zorgen dat deze niet precies gelijk is aan nul, terwijl het wel zo had moeten zijn als we expliciet het vlak door het punt hebben gecreëerd. Men moet dan ook speciale aandacht hebben voor dit soort fouten.

Eén manier om dit op te lossen is door de invoering van zogenaamde *exacte geometric computation* (EGC). De fundamentele taak van EGC is om correcte beslissingen te nemen op basis van algebraïsche vergelijken. Het steunt daarbij op verschillende principes met aan de basis het gebruik van arbitrair grote gehele en rationale getallen (die steeds exact zijn), en vlottende komma getallen waarvan de mantisse uitbreidbaar is om de exacte oplossing te bevatten.

Een ding hebben alle methodes die EGC invoeren gemeen: ze zijn enorm belastend wat betreft de performantie. Het rekenen met zeer grote gehele getallen, met rationale getallen,

of met vlottende komma getallen met arbitraire mantisse vergen zowel veel geheugen als processortijd. En dat is zeker niet gewenst.

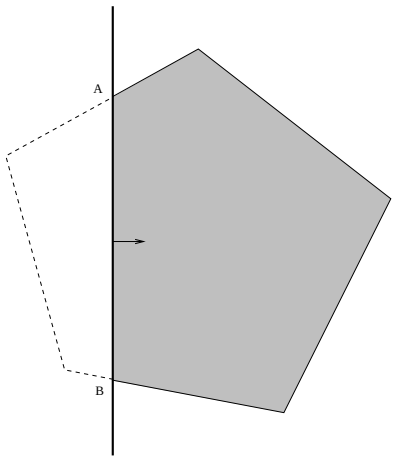
In `bass3` hebben we gepoogd om EGC te vermijden, door het toepassen van een een aantal technieken die hierna. Dit is gelukt, maar we hebben toch de weg vrij gehouden om EGC te introduceren in `bass3` door een nieuwe naam `Real` te introduceren. Deze wordt overal gebruikt waar we wensen een reëel getal voor te stellen, en is standaard ingesteld als een eenvoudige `double`.

```
typedef double Real;
```

Hier volgen vier methodes die in bass3 werden gebruikt om foutieve beslissingen ten gevolge van afrondingsfouten te beperken.

5.4.1 Voorkennis over het resultaat

Als je op voorhand weet welk resultaat een bewerking moet uitkomen, vertrouw dan niet op de uitkomst van de bewerking, maar gebruik het resultaat uit de voorkennis. Op deze manier ben je er zeker van dat er geen nieuwe afrondingsfouten worden geïntroduceerd.

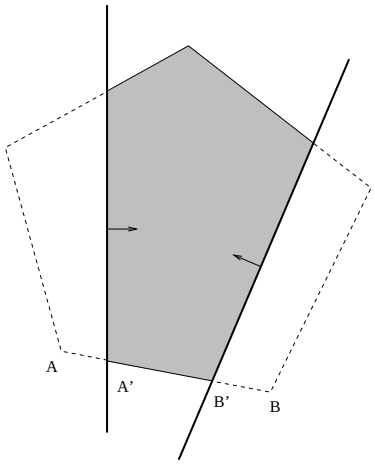


Figuur 5.5: Hergebruik van snijvlakken (doorsnede)

Deze techniek wordt in `bass3` gebruikt bij de generatie van snijvlakken van een bundel. In figuur 5.5 is een vijfhoekige portal afgebeeld waarvan een stuk wordt weggesneden door een snijvlak, wat leidt tot een ribbe $[AB]$. De resulterende portal is in het grijs ingekleurd. De bundel die in transmissie door deze portal moet worden geconstrueerd zal een snijvlak aanmaken voor elke ribbe van het resultaat, en dus ook door $[AB]$. Dit snijvlak heeft reeds twee punten gemeen met het origineel snijvlak die $[AB]$ veroorzaakte. Bovendien hebben ze nog een derde niet-colineair punt gemeen: de bron S. Dit betekent dat deze twee snijvlakken precies aan elkaar gelijk moeten zijn. Als we het nieuwe snijvlak berekenen aan de hand van A, B en S, dan kunnen afrondingsfouten er voor zorgen dat deze niet

precies aan elkaar gelijk zijn. Om dit te vermijden onthouden we het origineel snijvlak en kopiëren we dit waar nodig.

5.4.2 Gebruik van originele data



Figuur 5.6: gebruik van originele data (doorsnede)

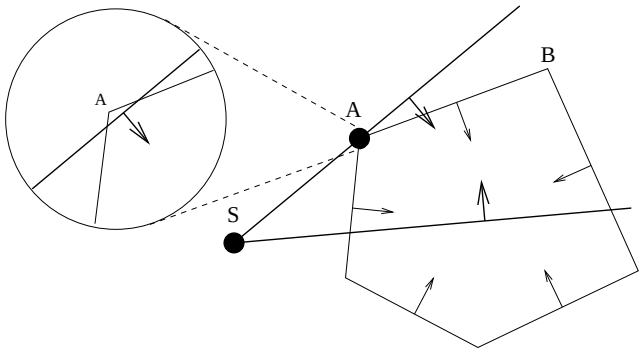
Een andere techniek is het gebruik maken van originele data i.p.v. resultaten uit berekeningen. Op deze manier beperk je opnieuw de afrondingsfouten. Deze techniek is uiteraard niet overal toepasbaar. Eén plaats in `bass3` waar dit wel gebruikt kan worden, is opnieuw de generatie van snijvlakken. In figuur 5.6 wordt ribbe $[AB]$ door twee snijvlakken in gekort tot $[A'B']$. Als we hierdoor een snijvlak construeren, dan zouden we normaal gebruik maken van de punten A', B' en de bron S. A' en B' zijn echter ontstaat uit berekeningen, en we weten dat de punten A en B ook in het snijvlak moeten liggen (ze zijn immers colineair met A' en B'). Daarom zullen we voor de constructie van dit snijvlak toch gebruik maken van de originele punten A en B, en de bron S.

5.4.3 Topologische relaties

De sterk gekoppelde winged-pair structuur laat ons toe om een aantal testen uit te voeren zonder numerieke berekeningen. In `bass3` passen we deze techniek twee maal toe, opnieuw in verband met de snijvlakken van een bundel.

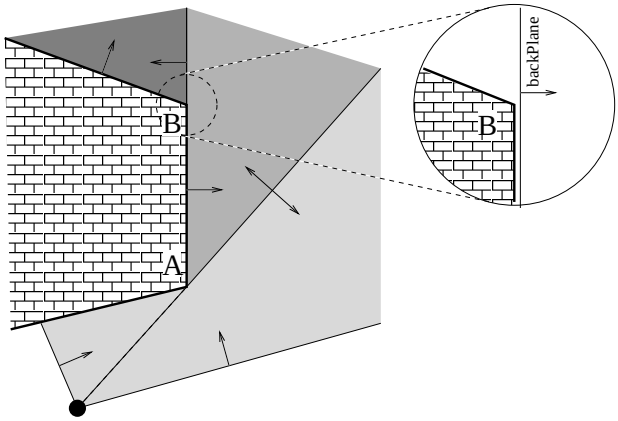
In figuur 5.7 is een bundel getekend die tot een vijfhoekige cel behoort. Voor die bundel is er een snijvlak geconstrueerd door bron S en ribbe A (de ribbe staat loodrecht op het blad). De bundel zal dit snijvlak bijhouden in een `ClipPlane3` structuur. In die structuur zal het ook een pointer naar A bijhouden.

Welk voordeel levert ons dit? We bekijken de situatie waarbij de wand AB getest wordt ten opzichte van het snijvlak SA. Door een afrondingsfout kan ribbe A net achter het



Figuur 5.7: topologische relaties (bovenaanzicht)

snijvlak liggen (zie inzet). Dit is in tegenstelling met de constructie van het snijvlak: SA zou *door* A moeten gaan. Als we geen extra maatregelen nemen, dan zal een smalle strook van AB worden weggesneden: de ribbe A. Dit is niet wenselijk omdat we we dan alle topologische informatie rond A verliezen. Als ribbe A niet meer tot de resulterende portal behoort, dan kan bijvoorbeeld nooit diffractie aan deze ribbe optreden. Als we echter eerst testen of een ribbe van de wand gelijk is aan de ribbe van het snijvlak, dan merken we dat dit het geval is voor A. Op dat moment weten we dat er slechts twee situaties mogelijk zijn: de wand ligt *volledig* voor het snijvlak, of er *volledig* op of achter. In het eerste geval zeggen we dat ze volledig zichtbaar is, in het tweede geval volledig onzichtbaar. En dat is precies wat we verwachten als het snijvlak door SA gaat.

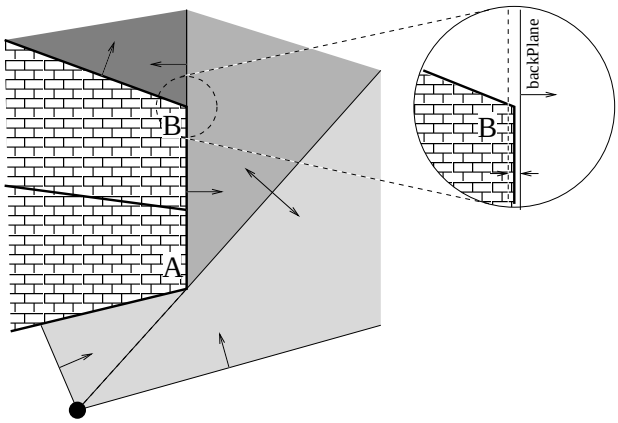


Figuur 5.8: topologische relaties bij dubbele diffractie

Hetzelfde doen we ook bij de constructie van de `backPlane` bij diffractie (zie sectie 5.2.2). We weten dat dit vlak door elke ribbe van de wand van `backPair` gaat. Door deze wand te stockeren in `ClipPlane3`, kunnen we later net dezelfde test doen als hierboven. In figuur 5.8 is geïllustreerd dat ribbe B (loodrecht op het blad) achter de `backPlane` ligt. Nochthans zou volgens de constructie de `backPlane` door B moeten gaan. Dit zou tot gevolg hebben dat er aan ribbe B geen diffractie zou optreden. Maar doordat ribbe B

gekoppeld is aan de wand van `backPair`, zal deze toch gedetecteerd worden als zijnde *"binnen"* de `backPlane`, waardoor toch diffractie zal optreden.

5.4.4 Invoeren van ε-toleranties



Figuur 5.9: invoeren van toleranties

Ondanks bovenstaande maatregelen, kunnen we toch snel in de problemen lopen. Dit is vooral te merken bij diffractie. Niet alle situaties zijn immers op te vangen door topologische relaties bij te houden. In figuur 5.9 behoort B niet meer tot de `backPair`. Het obstakel is immers in twee gesplitst, en de invloed van `backPair` loopt slechts tot aan de scheiding. Om B toch nog te detecteren als zijnde "binnen" de `backPlane`, voeren we een kleine tolerantie ε in. Dit is een waarde die zeer klein is t.o.v. de variaties in de geometrie, maar die toch nog groter is dan de verwachte afrondingsfout. Met deze tolerantie classificeren we B reeds voor de backPlane, als het maar niet verder dan ε achter het vlak ligt.

$$\overrightarrow{n} \cdot \overrightarrow{OB} + d > -\varepsilon \tag{5.1}$$

Alhoewel het invoeren van deze tolerantie alle resterende problemen heeft opgelost, moet men er toch bij opletten. Met name omdat ε-toleranties niet langer de transiviteit van de gelijkheidstest garanderen. Stel dat we toelaten dat twee punten aan elkaar gelijk worden bevonden, als de afstand tussen beiden kleiner is dan ε, dan is het niet langer zo dat als $A = B$, en $B = C$ dat dan automatisch $A = C$. Het gebruik van ε-toleranties in `bass3` stelt echter geen problemen als deze.

$$A = B \Leftrightarrow \|A - B\| < \varepsilon \tag{5.2}$$

Hoofdstuk 6

Integratie in ABT: ResponderRayEvent

In dit hoofdstuk wordt de integratie van `bass3` in `ABT` besproken. `ABT` is een 2.5D bundeltrekker binnen de onderzoekgroep *Acustica*. `ABT` beschikt over een uitgebreid akoestisch model, maar heeft de beperking dat de ingebouwde bundeltrek slechts 2.5 dimensionaal werkt. Het doel van dit eindwerk was om een 3D bundeltrekker af te leveren die inplugbaar is op het akoestisch model van `ABT`. Deze 3D bundeltrekker is `bass3`, en we kunnen dit inpluggen op `ABT` door een geschikte `Responder` te schrijven: `ResponderRayEvent`.

6.1 Implementatie

6.1.1 Voorstelling van een geluidstraal: RayEvent

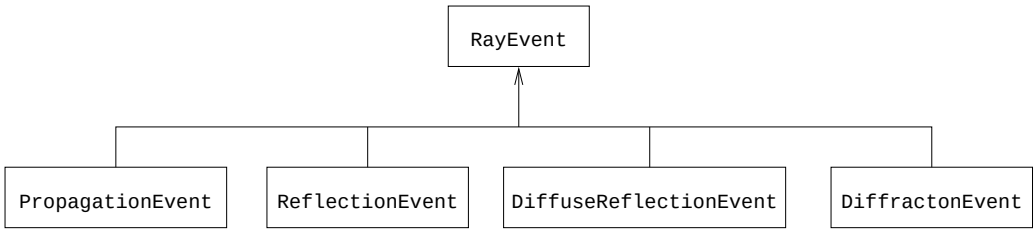
Het akoestisch model van ABT verwacht geluidstralen in de vorm van een aaneenschakeling van `RayEvent`'s. `RayEvent` is de basisklasse van alle soorten gebeurtenissen die over de lengte van een geluidstraal kunnen voorkomen. Hiervan zijn er vier specifieke RayEvents afgeleid (figuur 6.1):

PropagationEvent Propagatie van geluid door een medium voor. Deze propagatie gebeurt over een recht lijnstuk tussen twee punten.

ReflectionEvent Speculaire reflectie van geluid op een wand voor. Gegeven zijn het medium van de wand en de hoek t.o.z. van de normaal waarmee de straal invalt op de wand.

DiffuseReflectionEvent Diffuse reflectie van geluid op een wand voor. Deze gebeurtenis wordt echter nog niet ondersteund.

DiffractionEvent Diffractie van een geluidstraal aan een ribbe.



Figuur 6.1: hiërarchie van `RayEvent`

6.1.2 Padgeneratie

De taak van `ResponderRayEvent` is om een aaneenschakeling van bundels (`Beam3Stack`) om te zetten tot `RayEvents`, waarbij het zowel speculaire reflectie als diffractie in rekening brengt. Deze vertaling gebeurt in twee stappen: eerst wordt het kortste pad tussen bron en ontvanger bepaald, daarna wordt een reeks `RayEvent`'s gegenereerd.

Propagatie

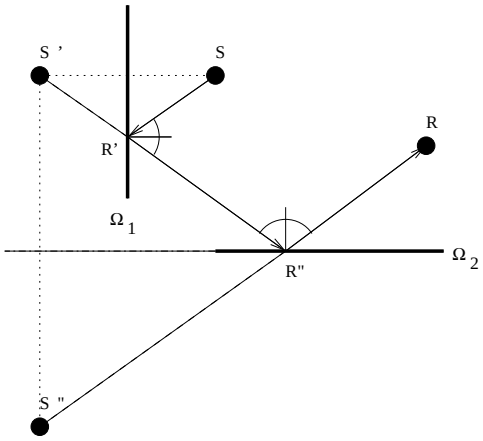
Figuur 6.2: padgeneratie zonder reflectie en diffractie

Het eerste deel van de padgeneratie zoekt naar de ene geluidstraal uit de bundel die bron en ontvanger verbindt. Als er noch speculaire reflectie, noch diffractie optreedt in de bundel, is deze taak zeer eenvoudig: het korste pad is een rechte tussen bron en ontvanger (figuur 6.2). Er zal dan ook slechts één `RayEvent` worden geproduceerd: een `PropagationEvent` die bron en ontvanger verbindt.

Speculaire reflectie

Als er enkel speculaire reflectie optreedt, kunnen we dit oplossen met behulp van het spiegelbronmodel (sectie 2.1). Men start aan de bron S en spiegelt deze t.o.z. alle reflecterende wanden (figuur 6.3). Zo krijgt men de virtuele bronnen S' en S''. We noemen dit proces *het ontplooien van de reflectie*. Het geluidspad is nu ontplooid over de reflecterende wanden tot een recht pad $S''R$ tussen S'' en R. Daarna wordt het snijpunt R'' van dit pad met het vlak van de laatste reflecterende wand Ω_2 bepaald. Ook R' wordt bepaald als het snijpunt van $S'R''$ met het vlak van Ω_1, en zo vinden we het uiteindelijke geluidspad $SR'R''R$.

Belangrijk is dat R' en R'' punten van de Ω_1 en Ω_2 moeten zijn. Als één van deze punten buiten de wand valt, dan kan de geluidstraal niet *op* de wand reflecteren, en



Figuur 6.3: padgeneratie bij speculaire reflectie

is er geen geldig geluidspad mogelijk. Als er enkel speculaire reflectie aanwezig is zullen deze snijpunten altijd binnen de wand vallen, omdat de bundeltrek de zichtbaarheid voor speculaire reflecties perfect kan oplossen. Als er ook diffractie is gemodelleerd, dan is dit niet langer het geval, en moet dit expliciet worden getest.

Tenslotte wordt elk van de lijnstukken die deel uitmaken van het geluidspad vertaald naar een `PropagationEvent`, afgewisseld door `ReflectionEvent`'s die de reflecties zelf voorstellen.

Diffractie.

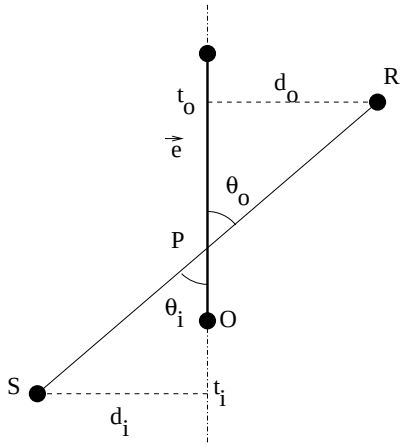
Als er diffractie optreedt, dan is de padgeneratie minder eenvoudig. We zullen het probleem eerst behandelen zonder dat er speculaire reflectie optreedt. Daarna zullen we ook reflecterende wanden in rekening brengen.

Als er N diffractie ribben zijn, dan zal het geluidspad bestaan uit $N+1$ rechte lijnstukken die kop aan staart de bron met ontvanger verbinden. De lijnstukken ontmoeten elkaar telkens in één punt P_j $(j = 0 \ldots N-1)$ van een diffractie ribbe. Deze punten zijn zodanig gekozen dat het resulterende geluidspad zo kort mogelijk is. Het is dus de taak van de padgeneratie om deze punten te vinden.

Een ribbe wordt voorgesteld met een beginpunt O_j en een vector $\overrightarrow{e}_j$. $\|\overrightarrow{e}_j\|$ is de lengte van een ribbe. Zo kan P_j met één scalair getal t_j worden voorgesteld:

$$P_j = O_j + t_j \overrightarrow{e}_j \qquad t \in [0,1] \tag{6.1}$$

Als er slechts één diffractie ribbe is, kan dit punt analytisch worden bepaald. Gegeven zijn bron S, ontvanger R en ribbe O, $\overrightarrow{e}$. Het doel is om het punt $P = O + t\overrightarrow{e}$ te vinden zodat $\left\|\overrightarrow{SP}\right\| + \left\|\overrightarrow{PR}\right\|$ minimaal is. Bovendien vertelt de Uniforme Diffractie Theorie dat de hoeken die $\overrightarrow{SP}$ en $\overrightarrow{PR}$ met de ribbe maken aan elkaar gelijk zijn: $\theta_i = \theta_o$ (zie sectie 2.4).



Figuur 6.4: Bepaling van P_i op een diffractie ribbe

We vinden dit punt door eerst R rond de ribbe te roteren zodat S, R en de ribbe in één vlak liggen. Dit kan omdat volgens de UDT elke straal op de kegel met openingshoek θ_o een uitgaande straal kan zijn (figuur 2.12). Als we een oplossing vinden waarbij S, R en de ribbe coplanair zijn, dan bekomen we ook een oplossing voor het oorspronkelijk probleem.

We bekomen dan de vlakke tekening uit figuur 6.4. P wordt bepaald door het snijpunt van rechte SR met de ribbe. Dit snijpunt is volledig bepaald door de orthogonale projecties van S en R op de draagrechte van de ribbe (dit geeft ons t_i en t_o, de parameters van deze projecties) en de afstanden van S en R tot deze draagrechte (d_i en d_o).

$$t_i = \frac{\overrightarrow{OS} \cdot \overrightarrow{e}}{\|\overrightarrow{e}\|} \tag{6.2}$$

$$t_o = \frac{\overrightarrow{OR} \cdot \overrightarrow{e}}{\|\overrightarrow{e}\|} \tag{6.3}$$

$$d_i = \|S - (O + t_i \overrightarrow{e})\| \tag{6.4}$$

$$d_o = \|R - (O + t_o \overrightarrow{e})\| \tag{6.5}$$

Met behulp van de stelling van Thales vinden we een parameter voor het punt P:

$$t = \frac{t_i d_o + t_o d_i}{d_i + d_o} \tag{6.6}$$

Bemerk dat het eigenlijk niet nodig is om R te roteren. Immers, elke rotatie van het punt R rond de diffractie ribbe, zal dezelfde parameter t_2 opleveren als het geprojecteerd wordt. Dit valt mooi samen met de UDT die stelt dat een invallende straal $\overrightarrow{SP}$ resulteert in een kegel van stralen.

Als er meerdere diffractie ribben tussen de bron en ontvanger voorkomen, kunnen we dit niet langer zomaar analytisch oplossen. Er moeten namelijk N parameters $t_{0\ldots N-1}$ tegeli-

Algorithm 10 Iteratieschema voor padgeneratie bij meervoudige diffractie

```
void solveDiffraction(S, R, O_{0...N-1}, e⃗_{0...N-1}, t_{0...N-1}, ε_L, k_max)
{
    // initialisation
    P_{-1} = S;
    P_N = R;
    for (j = 0; j < N; ++j)
    {
       t_j = 1/2;
       P_j = O_j + t_j e⃗_j;
    }
    L_old = pathLength(P_{-1...N});

    // find solution
    k = 0;
    while (k < k_max)
    {
        // find next approximation
        for (j = 0; j < N; ++j)
        {
            t_j = adjustT(P_{j-1}, O_j, e⃗_j, P_{j+1});
            P_j = O_j + t_j e⃗_j;
        }

        // check approximation
        L = pathLength(P_{-1...N});
        if (|L - L_old| < ε_L)  return true; // solution is found
        L_old = L;
    }
    return false; // solution isn't found
}

Real pathLength(P_{-1...N})
{
    return ||P_{-1}P_O⃗|| + ||P_0P_1⃗|| + ... + ||P_{N-1}P_N⃗||;
}

Real adjustT(P_i, O, e⃗, P_o)
{
    t_i = (OP_i⃗ · e⃗) / ||e⃗||;
    t_o = (OP_o⃗ · e⃗) / ||e⃗||;
    d_i = ||P_i - (O + t_i e⃗)||;
    d_o = ||P_i - (O + t_o e⃗)||;
    return (t_i d_o + t_o d_i) / (d_i + d_o);
}
```

jkertijd worden bepaald om een oplossing te vinden waarbij de opeenvolgende lijnstukken het korste pad voorstellen. We gebruiken daarom een iteratief schema waarbij we een initieel set van parameters t_j steeds verbeteren tot deze een voldoende oplossing bereikt.

Als initiële set van parameters gebruiken we $t_{0\ldots N-1} = \frac{1}{2}$. Alle hoekpunten P_j liggen dan precies in de helft van de diffractie ribben. Daarna wordt per iteratie iedere parameter t_j afzonderlijk aangepast om het korste pad tussen het voorgaande en het volgende hoekpunt P_{j-1} en P_{j+1} te bepalen[1]. Daarbij worden het voorgaande en het volgende hoekpunt vastgehouden. Deze nieuwe parameter t_j wordt dan onmiddellijk gebruikt om ook de volgende parameter t_{j+1} te verbeteren. We doen dit steeds opnieuw, iteratie na iteratie, tot de variatie in de padlengte t.o.z. van de vorige waarde gedaald is beneden een tolerantie ε_L. We spreken dan van convergentie. We leggen ook een maximaal aantal iteraties k_{max} vast zodat het iteratieschema steeds wordt beëindigd, ook als er geen convergentie optreedt. Dit schema is geïmplementeerd in algoritme 10.

Als er naast diffractie ook nog reflectie optreedt, dan moeten we een combinatie maken van het spiegelbronmodel en deze methode. Eerst wordt het probleem *ontplooid* zodat het probleem vrij van reflecties is. Er blijven nog enkel de bron, de ontvanger en de diffractie ribben over. Hierop passen we dan bovenstaande methode toe voor het bepalen van alle hoekpunten op de diffractie ribben en de lijnstukken daartussen. Tenslotte zal voor elk van deze resulterende lijnstukken het spiegelbronmodel verder worden toegepast om de reflecterende wanden opnieuw in rekening te brengen.

Het kan gebeuren dat een oplossing wordt gevonden waarbij één of meerdere parameters buiten het toegelaten bereik vallen: $t_j \notin [0, 1]$. In dat geval ligt het hoekpunt buiten de diffractie ribbe, en dat levert een ongeldig pad op. Dit kan gebeuren omdat we geen perfectie oplossing gebruiken voor het zichtbaarheidsprobleem bij diffractie in de bundeltrek. Daarom moeten we expliciet op deze situatie testen, en alle paden waarbij een parameter uit het bereik valt verwerpen. Verder moeten we ook expliciet controleren dat elk reflectie punt binnen de reflectie wand ligt, en dat alle stralen *door* de transmissie portals gaan, om die zelfde reden.

6.2 Resultaten / performantie

Met behulp van deze `ResponderRayEvent` hebben we de bundeltrekker getest en vergeleken met de bundeltrekker uit `ABT`. Hierbij werd niet het volledige akoestisch model van `ABT` gebruikt omdat de rekenlast daarvan veel hoger is dan de bundeltrekker zelf, en we wensen enkel de bundeltrekker te testen.

6.2.1 Akoestisch model

Als vereenvoudigd model werden volgende formules gebruikt:

[1] In dit schema beschouwen we S en R ook als hoekpunten en we definiëren $P_{-1} = S$ en $P_N = R$.

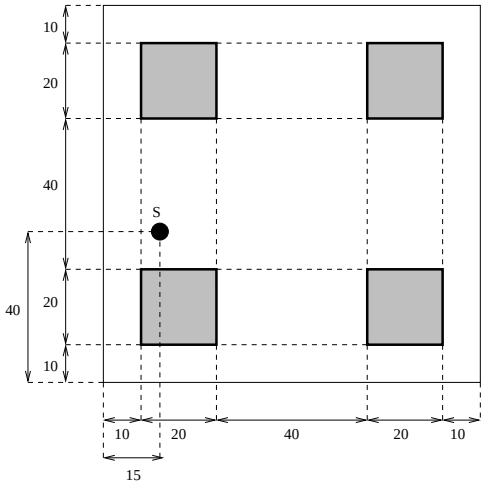
- propagatie coëfficiënt: $P(L) = \frac{1}{2\pi L^n}$ met L de totale lengte van het geluidspad. Voor 3D simulaties is fysisch gezien $n = 2$, maar omdat deze het geluid te snel verzwakt over een korte afstand, werd ook $n = 1$ gebruikt om een betere visuele indruk van de resultaten te krijgen.

- speculaire reflectie coëfficiënt: $R(\theta) = \frac{Z\cos\theta - \rho_0 c}{Z\cos\theta + \rho_0 c}$ met Z een frequentie onafhankelijke impedantie van het materiaal, θ de hoek van de invallende straal t.o.z. de normaal en $\rho_0 c$ de karakteristieke impedantie van lucht.

- diffractie coëfficiënt: $D(\theta) = (\cos\theta)^m$, met θ de hoek tussen de uitgaande straal en de ribbe, en m een arbitraire waarde. Deze formule is niet gesteund op fysische fenomen, maar werd arbitrair bepaald. Visuele controle toonde aan dat de verzwakking aannemelijk is voor geluid met een gemiddelde frequentie van $250Hz$. m werd hiervoor vastgelegd op $m = 10$.

6.2.2 Testomgevingen

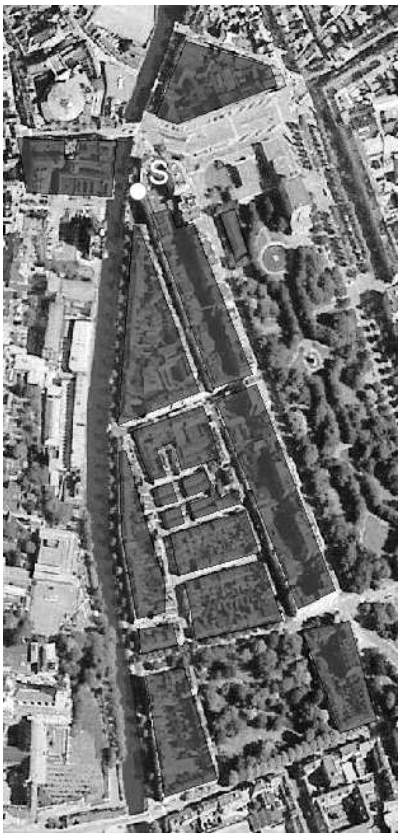
Er werden twee testomgevingen gebruikt: A en ZUID.



Figuur 6.5: testomgeving A

Omgeving A is $100m$ op $100m$ groot, en bestaat uit vier obstakels van $20m$ op $20m$ groot en $15m$ hoog. De bron ligt op het punt $(15m, 40m, 1m)$ en de $n \times n$ ontvangers liggen op een regelmatig rooster ter hoogte van $1.4m$.

Omgeving ZUID is een eenvoudige voorstelling van de Zuid omgeving in Gent. Ze is $424m$ op $887m$ groot en de gebouwen hebben opnieuw een hoogte van $15m$. De bron ligt op het punt $(149m, 668m, 10m)$ en de ontvangers liggen op $1.4m$ hoogte.



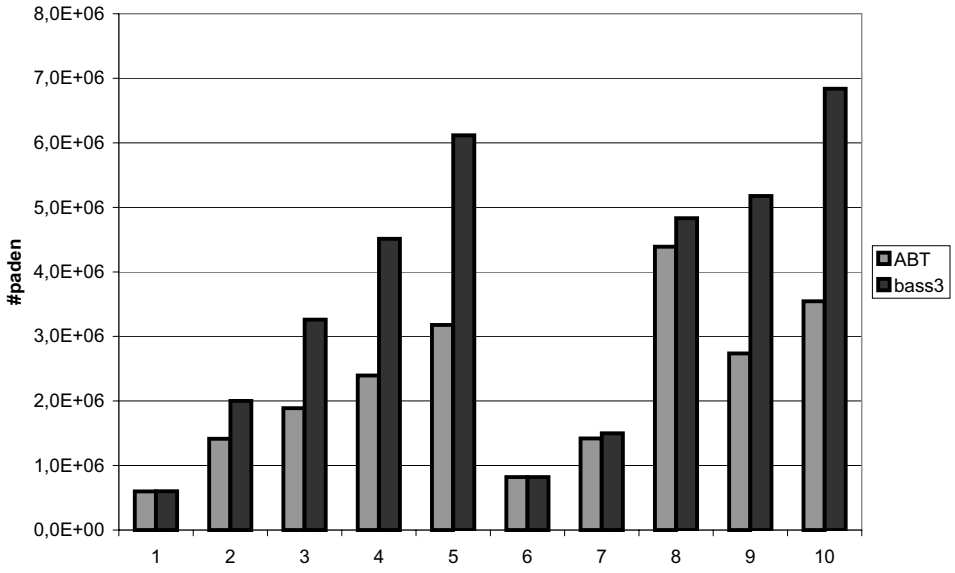
Figuur 6.6: omgeving ZUID

6.2.3 Vergelijking tussen bass3 en ABT

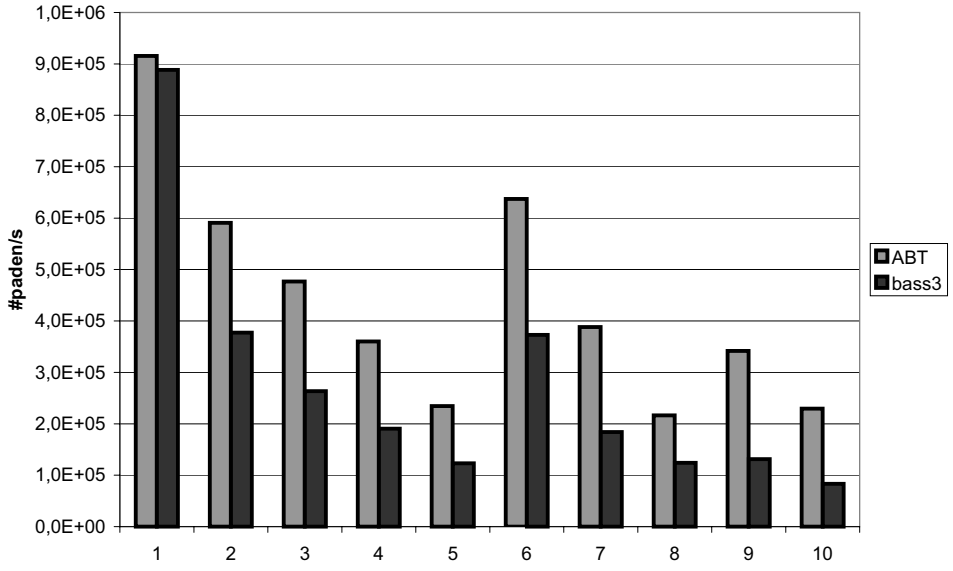
Een eerste reeks simulaties werden verricht om de performantie van `bass3` te vergelijken met de bundeltrekker van `ABT`. Tien verschillende simulaties werden op omgeving A uitgevoerd en vergeleken. Bij deze tien simulaties werden telkens andere limieten ingesteld voor het maximum aantal reflecties en diffracties per straal:

1. enkel propagatie
2. enkelvoudige reflectie
3. 2-voudige reflectie
4. 4-voudige reflectie
5. 8-voudige reflectie
6. enkelvoudige diffractie
7. 2-voudige diffractie
8. 3-voudige diffractie

9. enkelvoudige diffractie + 4-voudige reflectie
10. enkelvoudige diffractie + 8-voudige reflectie



Figuur 6.7: vergelijking tussen `bass3` en `ABT`: aantal gegenereerde geluidspaden



Figuur 6.8: vergelijking tussen `bass3` en `ABT`: aantal gegenereerde geluidspaden per seconde (AMD Athlon XP2200+ (1.7GHz), 1536MB RAM @ 266MHz)

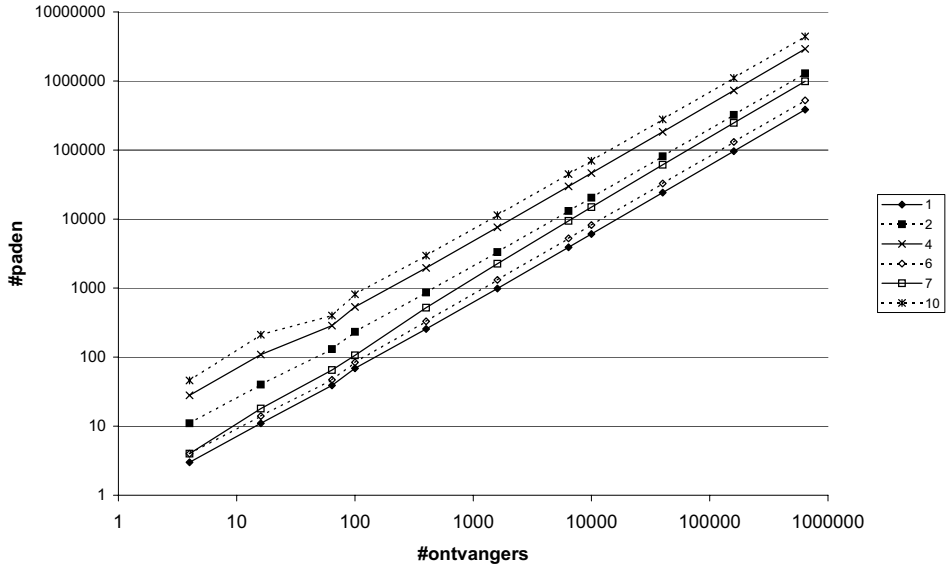
Uit de resultaten blijkt dat `bass3` en `ABT` ongeveer evenveel geluidspaden genereren zolang er geen reflectie is ingeschakeld (grafiek 6.7). Met reflectie blijkt `bass3` ongeveer twee maal zoveel paden als `ABT` te genereren. Dit komt omdat in `bass3` het maaiveld ook reflectief is in tegenstelling tot `ABT`. Een reflectief grondvlak leidt onmiddellijk tot bijna

een verdubbeling van het aantal paden. Deze goede overeenkomst tussen `bass3` en `ABT` is uiteraard een goed teken, omdat beiden hetzelfde probleem oplossen.

Als we enkel progagatie of enkelvoudige diffractie in rekening brengen, dan blijken beiden zelfs exact hetzelfde aantal paden te produceren. Dit komt omdat het probleem dan tot een twee dimensionaal probleem wordt gereduceerd waarbij `bass3` en `ABT` op dezelfde manier op reageren. Inderdaad, bij enkelvoudige diffractie kunnen we geen stralen hebben over de daken van obstakels, omdat we daar minimum een tweevoudige diffractie voor nodig hebben. De reden waarom `bass3` iets meer paden genereert bij tweevoudige en drievoudige diffractie is omdat `bass3` en `ABT` diffractie over het dak van een gebouw anders afhandelen. `ABT` moet dit als speciaal geval oplossen, terwijl `bass3` dit volledig drie dimensionaal kan afhanden, wat leidt tot iets meer paden.

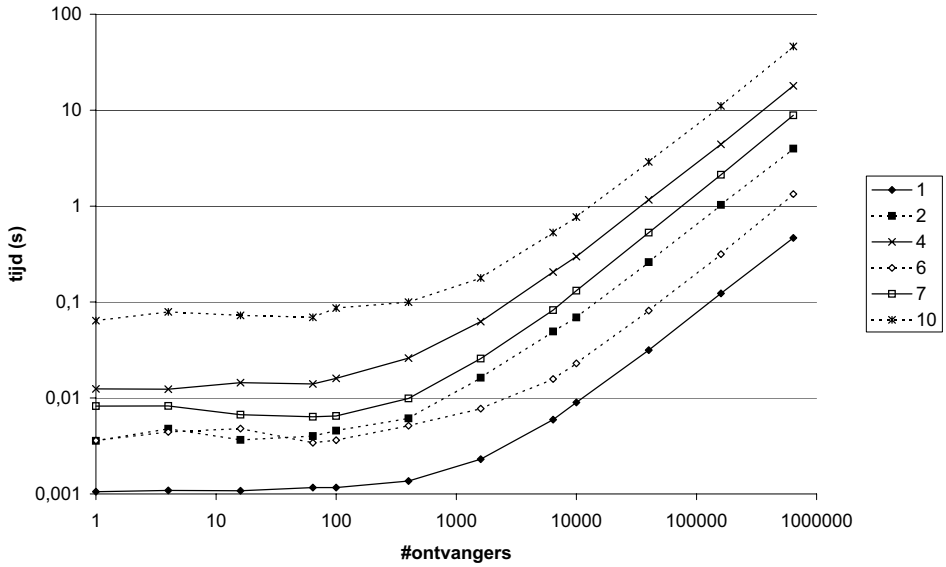
Ook wat de snelheid betreft blijkt `ABT` sneller te zijn dan `bass3`, `ABT` kan meer paden per seconde genereren dan `bass3` (grafiek 6.8). Dit heeft heel veel te maken met het feit dat veel geometrische problemen sterk kunnen geoptimaliseerd worden als men zich beperkt tot twee dimensies. `ABT` kan hier voordeel uithalen, `bass3` moet de problemen volledig drie dimensionaal aanpakken. Enkel voor propagatie blijken `ABT` en `bass3` aan elkaar gewaagd te zijn.

6.2.4 Invloed van het aantal ontvangers



Figuur 6.9: vergelijking bij verschillend aantal ontvangers: aantal gegenereerde paden.

De meeste processortijd in de bundeltrekker gaat naar de padgeneratie. We hebben bovenstaande simulaties 1, 2, 4, 6, 7 en 10 in omgeving A opnieuw uitgevoerd met `bass3`, en telkens voor een verschillend aantal ontvangers $n \times n$, met n =1, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400 en 800. Uit figuren 6.9 en 6.10 blijkt dat het aantal paden lineair toeneemt met



Figuur 6.10: vergelijking bij verschillend aantal ontvangers: totale simulatietijd (`bass3`, Pentium 4 @ 2.5GHz, 512MB RAM).

het aantal ontvangers, en dat ook de totale simulatie tijd lineair toeneemt vanaf ongeveer 1000 ontvangers. Hieruit kunnen we besluiten dat vanaf 1000 ontvangers, de meeste processortijd naar de padgeneratie gaat. Tot aan deze grens heeft de bundeltrekker zelf nog het grootste (constante) aandeel hierin.

6.2.5 Invloed van het de omgeving en aantal cellen

complexiteit van de omgeving

In deze test werden de omgevingen A en ZUID met elkaar vergeleken met telkens 400×400 ontvangers. ZUID blijkt iets sneller te zijn als het gaat om pure reflectie, maar moet duidelijk het ondenrspit delven als diffractie wordt ingeschakeld (tabel 6.1).

Dat ZUID wint wat betreft reflectie is te verklaren doordat de obstakels eerder onregelmatig zijn geschikt. Bij A zijn de ontvangers zeer mooi uitgelijnd waardoor we reflectie tussen twee evenwijdige wanden krijgen die in principe eindeloos voort kan duren. Bij ZUID wordt reflectie m.a.w. sneller geëllimineerd.

Bij diffractie is het aantal ribben waar diffractie kan optreden van belang. Bij ZUID zijn dit er opmerkelijk meer waardoor de simulatietijd enorm stijgt. Vooral als bovendien ook nog reflectie wordt ingeschakeld, dan ziet de bundeltrekker nog veel meer potentiële diffractie ribben.

	1	2	3	4	5	6	7	9	10
A	0.12	1	2.3	4.4	8.8	0.3	2.2	5.8	11
ZUID	0.15	0.7	1.6	3.2	6.4	1.9	20.4	54	229

Tabel 6.1: 160000 ontvangers (simulatietijden in seconden) (P4@2.5GHz, 512MB RAM).

driehoekige versus n-gonale cellen

We moeten ons ook de vraag stellen welke invloed het optimaliseren van de convexe celstructuur heeft, waarbij cellen worden samengevoegd tot grotere convexe cellen. Het antwoord is volledig afhankelijk van het feit of de performantie bepaald wordt door de bundeltrek (een klein aantal ontvangers) of door de padgeneratie (een groot aantal ontvangers).

In het eerste geval is de optimalisatie voordelig voor de performantie. Als er minder cellen zijn, dan zijn er minder portals, en hoeven er minder bundels gegeneerd worden. Dit geeft ons een snelheidswinst (tabel 6.2).

We vermoeden dat in tweede geval de optimalisatie de nadelig zijn. Grotere cellen zullen meer ontvangers bevatten over een groot volume. Deze moeten allemaal getest worden ten opzichte van de bundel, zelfs als de bundel maar een klein gedeelte van de cel bestreikt. Kleinere cellen bevatten een meer lokale verzameling van ontvangers, waarbij de kans dat een ontvanger tot de bundel behoort nu groter is. Bovendien zal de bundeltrekker beslissen om verder te gaan op basis van het resultaat van *alle* ontvangers in de cel. Bij een grote cel kan de bundeltrekker beslissen om door te gaan, zelfs als slechts een klein aantal ontvangers een positief resultaat geven. Bij kleinere cellen zullen er naar verhouding meer positieve resulaten zijn (tenminste als er positieve resulaten zijn). We kunnen zeggen dat kleinere cellen beslissingen nemen op een lokaler niveau.

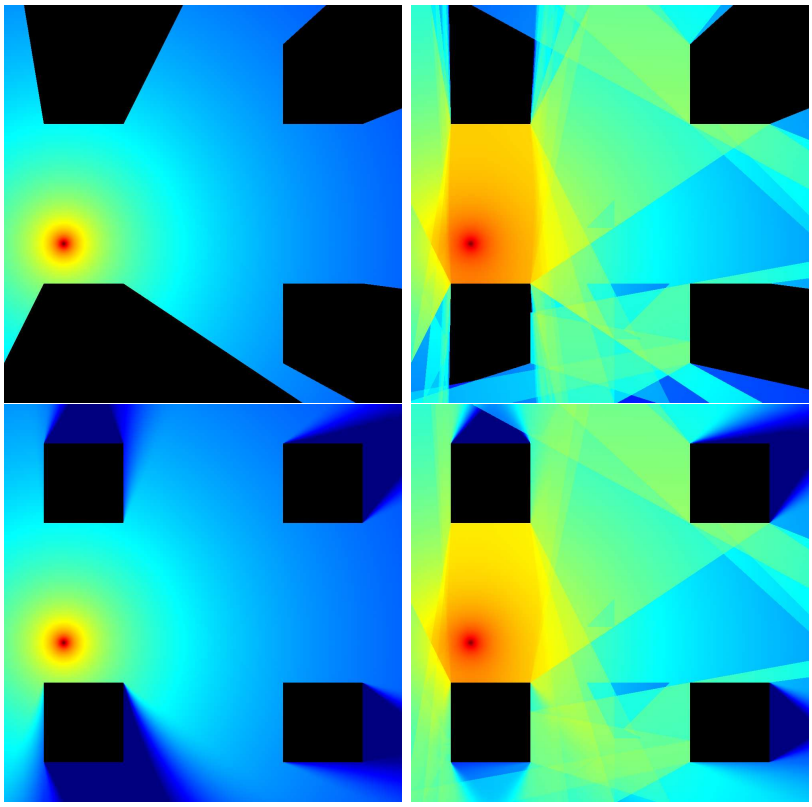
Testresultaten tonen echter geen significante vertraging aan (tabel 6.3). Hieruit blijkt opnieuw dat de programmeur doorgaans slecht is in het schatten van de impact van een algoritme, en dat de enige betrouwbare resulaten uit testen komen.

simulatie	1	2	3	4	5	6	7	9	10
driehoekig	1.6	6.4	13	26	55	4.4	8.3	57	127
n-gonen	1.2	4.3	8.5	16	31	3.6	7.0	38	92

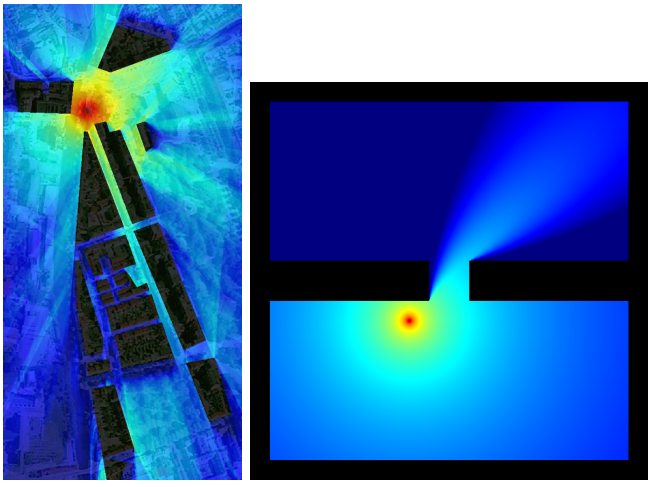
Tabel 6.2: 100 ontvangers (simulatietijden in microseconden) (P4@2.5GHz, 512MB RAM).

simulatie	1	2	3	4	5	6	7	9	10
driehoekig	0.5	3.9	9.2	17	36	1.2	5.7	24	49
n-gonen	0.5	4.0	9.2	18	37	1.3	8.8	24	46

Tabel 6.3: 640000 ontvangers (simulatietijden in seconden) (P4@2.5GHz, 512MB RAM).



Figuur 6.11: Simulaties in omgeving A: linksboven: enkel propagatie (instelling 1), rechtsboven: 8-voudige reflectie (instelling 5), linksonder, 3-voudige diffractie (instelling 8), rechtsonder: 2-voudige diffractie + 4-voudige reflectie (instelling 10).



Figuur 6.12: links: ZUID omgeving met 2-voudige diffractie en 4-voudige reflectie, rechts: twee kamers met absorberende wanden zijn verbonden met een smalle opening, hierbij wordt het effect van diffractie geïllustreerd.

Hoofdstuk 7

Besluit

In dit eindwerk werd het ontwerp en de implementatie van een 3D polygonale bundeltrekker voor akoestische simulaties besproken. Hierbij werden als eisen voorop gesteld dat de bundeltrekker in een volledige drie-dimensionale polygonale omgeving moet kunnen functioneren, en dat deze speculaire reflectie en diffractie moet ondersteunen. Verder werd gestreefd naar een flexibele opbouw zodat de bundeltrekker in variërende toepassingen kan worden ingezet. Het resultaat is `bass3`, een object-geörienteerde implementatie van de bundeltrekker, geschreven in `C++`.

Eerst werden de verschillende methodes uit de geometrische akoestiek besproken waaruit blijkt dat polygonale bundeltrek superieur is wat betreft performantie en nauwkeurigheid t.o.z. van andere methoden als het spiegelbronmodel, stralentrek, conische en piramidale bundeltrek. Anderzijds kan stralentrek wel worden ingezet in niet-polygonale omgevingen en kunnen diffractie en diffuse reflectie eenvoudiger worden gerealiseerd. Hoe polygonale bundeltrek kan ingezet in niet-polygonale omgevingen of hoe diffuse reflectie kan worden gerealiseerd werd niet behandeld. Voor de implementatie van diffractie werd besproken hoe de uniforme diffractie theorie – geschikt voor stralentrek – kan worden omgevormd voor gebruik in polygonale bundeltrek.

Een aantal datastructuren voor de opslag van de omgevingen werden besproken, en daarbij werd het concept van een convexe celstructuur geïntroduceerd. Er werd aangetoond dat deze structuur een zeer eenvoudig bundeltrek algoritme toelaat waarbij zichtbaarheid- en convexiteitsproblemen worden opgelost door de structuur. De winged-pair datastructuur werd voorgesteld als kandidaat om deze convexe celstructuur voor te stellen. T.o.z. van andere kandidaten als een polygonsoup, quadtree of BSP-tree, heeft deze het voordeel dat deze de topologische informatie van de celstructuur volledig kan bevatten.

In het kader van het eindwerk werd een preprocessor geïmplementeerd die 2.5D omgevingen kan omvormen tot een geschikte 3D celstructuur. Voor de opdeling van deze omgeving in convexe cellen werd aangetoond waarom een triangulatie werd gebruikt, en niet een BSP-tree.

Voor de implementatie van de bundeltrekker zelf, werd besproken hoe deze met afrondingsfouten ten gevolge van vlottende komma getallen omgaat, en hoe we hebben kunnen vermijden dat we *exact geometric computation* (EGC) moesten toepassen. Tegelijkertijd werd wel de weg vrij gehouden voor een eventuele introductie van EGC. Anderzijds werd aangetoond hoe het gedrag van de bundeltrekker wordt gestuurd door een strategy pattern, en hoe deze de bundeltrekker flexibel houdt zodat het in variërende toepassingen kan worden ingezet. Eén strategy werd geïmplementeerd voor de koppeling met `ABT`, een 2.5D bundeltrekker in de onderzoeksgroep *Acustica*, en hierbij werd padgeneratie bij diffractie besproken.

Zowel de implementatie van speculaire reflectie en diffractie is met succes gebeurt. De snelheid van de bundeltrekker werd getest t.o.z. `ABT`, en blijkt ongeveer twee maal zo traag te zijn. Dit is voornamelijk te wijten aan het feit dat `bass3` volledig drie dimensionaal moet werken, waar `ABT` optimalisaties voor het twee dimensionale probleem kan doorvoeren. Verder blijken `bass3` en `ABT` ongeveer hetzelfde aantal paden te genereren, op reflectie t.o.z. van het maaiveld na. De simulatieduur bij `bass3` wordt bij een groot aantal ontvangers voornamelijk bepaald door de padgeneratie, en stijgt lineair met het aantal ontvangers. Voor een klein aantal ontvangers wordt de bundeltrek zelf belangrijker, dat een constant aandeel in neemt volgens het aantal ontvangers.

Alhoewel het theoretisch mogelijk is om `bass3` in te zetten voor auralisatie toepassingen, blijkt dit in de praktijk niet zomaar mogelijk te zijn, omdat de bundeltrekker niet snel genoeg is voor dit soort van real-time toepassingen[1]. Verder onderzoek is nodig om `bass3` zo aan te passen dat deze ook voor auralisatie toepassingen geschikt wordt. Ook kan nog worden onderzocht hoe diffuse reflectie kan worden geïmplementeerd in `bass3` en hoe pure 3D omgevingen kunnen worden ingevoerd[2].

[1] Tenminste, als men wenst reflectie en diffractie in rekening te brengen. De visualisatiedemo doet dit niet en haalt zo toch een framerate van ongeveer 30–60fps op een Pentium4 @ 2.5GHz.

[2] De huidige preprocessors gaan uit van 2.5D data, en zijn niet zonder meer geschikt om *binnenomgevingen* van bv. kantoorgebouwen te verwerken.

Bijlage A

Overzicht implementatie: bass3

`bass3` is ontwikkeld in `C++`, met behulp van Microsoft Visual C++ 6.0 SP5 (MSVC65). Het is een object-geörienteerd ontwerp waarbij zoveel mogelijk aan de ISO standaard werd gehouden[1][18]. Hierbij werd wel rekening gehouden met de afwijkingen in MSVC65. Deze hebben vooral invloed op het gebruik van *templates*, en het bekende *scope* probleem bij `for` lussen [17].

Het grootste deel van de code in `bass3` is onder gebracht in de namespace `bass3`, op een paar uitzonderingen na waarbij code uit andere projecten werd herbruikt: `gis` bevat de code voor het inlezen van shapefiles (sectie 4.2), en `qe` bevat de constrained delaunay triangulator (sectie 4.1.2).

Voor de extensies van de bronbestanden werd gekozen voor `hpp` voor de *headers*, `cpp` voor de *implementations*, en `inl` voor de *inline implementations*. Declaraties en implementaties werden op deze manier strikt uit elkaar gehouden.

Een aantal headers zijn voor de algemene ondersteuning in `bass3` belangrijk:

`bass3.hpp` De enige header die door client code hoeft te worden binnen getrokken. Deze bevat alle nodige declaraties om `bass3` te kunnen gebruiken.

`bass3_common.hpp` Een header die wordt binnen getrokken door elke andere header, en algemene declaraties bevat.

`bass3_configure.hpp` Hiermee kunnen een aantal zaken in `bass3` tijdens compile-time worden geconfigureerd met een aantal macro's. Hier kan o.a. worden gespecifieerd of `bass3` moet gebruik maken van inline *implementations*, of normaal vectoren moeten worden genormaliseerd, en welke debug faciliteiten moeten worden ingeschakeld.

`bass3_compilers.hpp` Deze header moet een aantal verschillen tussen compilers oplossen. Momenteel is enkel de ondersteuning voor MSVC65 in gevuld.

[1] Een punt van discussie zijn naamloze structures. Deze werd gebruikt voor snelle toegang tot de `Vector3` members in de `Point3` structure (sectie 3.3.9). Vele compilers waaronder MSVC++65 ondersteunen deze constructie, terwijl het toch niet wordt toegelaten door de standaard.

`bass3_typedefs.hpp` In deze header worden een aantal integers met gegeven byte-lengte gedeclareerd.

`debug.hpp` Introduceert een aantal debug faciliteiten die in het `bass3` project zijn gebruikt.

In `bass3` is ook de ondersteuning voor een *memory debugger* ingebouwd. Daarvoor werd MMGR van Paul Nettle gebruikt [26]. Dit is een *freeware memory manager* die in de broncode wordt in gebed. Voor deze memory debugger moeten wel alle standaard headers worden uitgesloten, en daarvoor dienen de constructies `#include BASS3_NO_MEMORY_DEBUGGER` / `#include BASS3_MEMORY_DEBUGGER` rond deze headers.

We geven een overzicht van de componenten in `bass3`. Alle componenten zijn te vinden in de `bass3` namespace, tenzij anders vermeld. De bronbestanden die een componente `Foo` bevatten, beginnen met dezelfde naam (maar met kleine letters), en eindigen op `hpp`, `inl` of `cpp`: `foo.hpp`, `foo.inl` en `foo.cpp`.

`AABB3` Een 3D axis-aligned bounding box.

`Beam3` Een 3D bundel gebruikt door de bundeltrekker `Beam3Tracer`.

`Beam3_p` Een smart pointer naar een `Beam3` bundel.

`Beam3Stack` Een stapel van `Beam3` bundels.

`Beam3Tracer` De eigenlijke 3D bundeltrekker van `bass3`.

`Callback...` Constructies voor een verbeterd callback mechanisme. Kan niet alleen functie pointers maar ook specifieke methodes op specifieke objecten als callback gebruiken.

`Cell3` Een 3D polygonale convexe cel, zoals gebruikt in de `World3` structuur.

`Cell3Bounder` Een algoritme dat een `AABB3` construeert rond een `Cell3`.

`Cell3Finder` Een algoritme dat in een `World3` structuur de `Cell3` opspoort dat een gegeven punt bevat.

`ClippedEdge3` Een ribbe in een `ClippedFace3` portal.

`ClippedFace3` Het zichtbare gedeelte van een `Face3` door een `Beam3`. Een `ClippedFace3` is polygonaal en gebruikt `ClippedEdge3` om z'n ribben voor te stellen.

`ComposedCellHandle` Een composite handle dat de nodige akoestische handles i.v.m. een obstakel kan bundelen.

`ComposedEdgeHandle` Een composite handle nodig in `TriangulatorDelaunay`. Houdt naast de akoestische handle nog extra informatie vast.

`Edge2` Abstracte basisklasse van een 2D ribbe in `Triangulator`

`Edge2Delaunay` Conctrete `Edge2` in `TriangulatorDelaunay`

`Edge3` Een 3D ribbe, zoals gebruikt in de `World3` structuur.

`Face3` Een 3D wand/portal, zoals gebruikt in de `World3` structuur.

`Image32` Een 32bit ARGB bitmap klasse.

`Line3` Een 3D geometrische lijn.

`LineSeg2` Een 2D geometrisch lijnstuk

`LineSeg3` Een 3D geometrisch lijnstuk

`MatlabOStream` Een stream klasse voor uitvoer naar Matlab M-bestanden.

`NonCopyable` Mixin basisklasse om de kopieerbaarheid van een object te beperken.

`Object3` Representeert een object/ontvanger in de `World3` structuur op een gegeven locatie.

`Pair3` Een 3D ribbe/wand paar, zoals gebruikt in de `World3` structuur.

`Pair3ConstIterators...` Iterators voor gebruik met `Pair3`.

`Pair3Iterators...` Iterators voor gebruik met `Pair3`.

`Plane3` Een 3D geometrisch vlak

`Point2` Een 2D geometrisch cartesiaans punt.

`Point2h` Een 2D geometrisch homogeen punt.

`Point3` Een 3D geometrisch cartesiaans punt.

`Point3h` Een 3D geometrisch homogeen punt.

`Polarity` Polarititeit voor gebruik in de `World3` structuur.

`Polygon2` Een 2D geometrische polygoon

`Polygon3` Een 3D geometrische polygoon

`Preprocessor2D5` Een preprocessor die een `World3` structuur opbouwt vanaf 2.5D informatie.

`PreprocessorSHP` Een preprocessor die een `World3` structuur opbouwt vanaf een shapefile.

`Real` Wrapper rond een formaat dat een reëel getal kan voorstellen.

`Responder` Abstracte basisklasse van alle `Beam3Stack` responders.

`ResponderBridge` Concrete `Responder` die de methodes doorvoort naar Callback's.

`ResponderRayEvent` Concrete `Responder` die bass3 kan inpluggen op ABT.

`SimpleSHPReader` (namespace gis) Eenvoudige shapefile reader.

`Triangle2` Abstracte basisklasse van een 2D driehoek in `Triangulator`.

`Triangle2Delaunay` Concrete `Triangle2` in `TriangulatorDelaunay`.

`Triangulator` Abstracte bassisklasse van het triangulatie algoritme, voor gebruik in `Preprocessor2D5`.

`TriangulatorDelaunay` Concrete `Triangulator` die een Constrained Delaunay Triangulation implementeert.

`Vector2` Een 2D geometrische vector.

`Vector3` Een 3D geometrische vector.

`Vector4` Een 4D geometrische vector.

`Vertex3` Een 3D hoekpunt zoals gebruikt in de `World3` structuur.

`World3` Een 3D winged-pair structuur voor de opslag van een 3D polygonale convexe cel-structuur.

`World3_p` Een smart pointer naar `World3` met reference counting.

`World3Bounder` Een algoritme dat een AABB3 construeert rond een World3 object.

`World3Optimizer` Een algoritme dat de convexe cellen probeert samen te smelten tot grotere convexe cellen, en dat vlaggen opzet te bevordering van de implementatie van diffractie in `Beam3Tracer`.

`World3Tester` Een algoritme dat een aantal testen uitvoert op een `World3` object om deze te valideren.

Bibliografie

[1] ABRASH M. (1997). *Michael Abrash's Graphics Programming Black Book, Special Edition*. The Coriolis Group, Inc.

[2] ALEXANDRESCU A. (2001). *Modern C++ Design, Generic Programming and Design Patterns Applied*. Addison-Wesley

[3] AMANATIDES J. (1984) *Ray Tracing with Cones*. Computer Graphics 18, 3, 129–135.

[4] BIKKER J. (1999). *Real-time 3D Clipping (Sutherland-Hodgeman)*. flipCode development tutorials, http://www.flipcode.com/articles/.

[5] BOTTELDOOREN D. (2002) *Acustica* (cursusnota's). Gent, Faculteit Toegepaste Wetenschappen. Universiteit Gent.

[6] *CGAL db Kernel Reference Manual*. http://www.cgal.com

[7] DE GREVE B. (2002). *3D Geometry Primer, chapter II*. flipCode columns, http://www.flipcode.com/articles/.

[8] DE MUER T. (2001). *Datastructuren voor het snel weergeven van de akoestische omgeving bij auralisatie* (afstudeerwerk). Gent, Faculteit Toegepaste Wetenschappen. Universiteit Gent.

[9] DRUMM I.A. & LAM Y.W. (2000). *The adaptive beam-tracing algorithm*. Journal of the Acoustical Society of America, 107, 1405–1412.

[10] ESRI (1998). *ESRI Shapefile Technical Description*. http://www.esri.com.

[11] FARINA A. (1995). *RAMSETE – A New Pyramid Tracer for Medium and Large Scale Acoustic Problems*. Euro-Noise 95 Conference, Lyon.

[12] FUNKHOUSER T.A. (1999). *A visibility algorithm for hybrid geometry- and image-based modeling and rendering*. Computers & Graphics – UK, 23, 719–728.

[13] FUNKHOUSER T.A., CARLBORN I., ELKO, G., PINGALI G., SONDHI M. & WEST J. (1998). *A beam tracing approach to acoustic modeling for interactive virtual environments*. Computers & Graphics (SIGGRAPH '98), Orlando FL, 21–32.

[14] GAMMA E., HELM R., JOHNSON R. & VLISSIDES J. (1995) . *Design Patterns, de Nederlandse editie, Elementen van herbruikbare objectgeoriënteerde software*, Addison Wesley Professional.

[15] *GDAL - Geospatial Data Abstraction Library*. http://www.remotesensing.org/gdal.

[16] HUYGHENS C. (1912). *Traité de la Lumière*. Londen, Macmillan & Co.

[17] HYSLOP J. & SUTTER H. (2000). *Conversations: So Who's the Portable Coder?* C++ Experts Forum. http://www.cuj.com/experts/.

[18] ISO/IEC 14882 (1998). *Programming Languages – C++*. New York, American National Standards Institute.

[19] JOY K. I., LEGAKIS J. & MACCRACKEN R. (2002). *Data structures for multiresolution representation of unstructured meshes*. In: Farin G., Fagen H. & Hamann B. (eds.) Hierarchical approximation and geometrical methods for scientific visualization. Heidelberg, Germany, Springer-Verlag.

[20] LAWSON C. L. (1977). *Software for C^1 surface interpolation*. In: Rice J. R. (ed.) Mathematical Software III. New York, Academic Press.

[21] McGUIRE M. (2000). *The half-edge data structure*. flipCode development tutorials, http://www.flipcode.com/articles/.

[22] MICROSOFT CORPORATION. *DirectX 9.0 SDK Documentation*. http://www.msdn.microsoft.com

[23] MOLLER T. & HAINES E. (2000). *Real-Time Rendering*. A.K. Peters Ltd.

[24] MUCKE E. P., SAIAS I. & ZHU B. (1996). *Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations*. Proc. 12th ACM Symp. on Computationall Geometry (So CG'96), 274–283.

[25] MUANGWINE S. J. & HORNE R. E. N. (Eds.) (1998) *The Colour Image Processing Handbook*, First edition. London, Chapman & Hall.

[26] NETTLE P. (2002). *MMGR*. Fluid Studio's. http://www.fluidstudios.com.

[27] STROUSTRUP B. (2000). *De programmeertaal C++, derde editie*, Addison Wesley.

[28] TELLER S. J. (1992). *Visibility Computations in Densely Occluded Environments*. PhD thesis, Computer Science Div., Univ. of California, Berkely.

[29] TSINGOS N., FUNKHOUSER T., NGAN A. & CARLBOM I. (2001). *Modeling Acoustics in Virtual Environments Using the Uniform Theory of Diffraction*. SIGGRAPH 2001. Los Angeles.

[30] WELLER F. (1998) *On the Total Correctness of the Lawson's Oriented Walk Algorithm*, The 10th Internation Canadian Conference on Computational Geometry, Montréal, Canada, August 10–12.