



UNIVERSITEIT  
GENT

---

Faculteit Toegepaste Wetenschappen  
Vakgroep Informatietechnologie  
Voorzitter: prof. dr. ir. P. LAGASSE

# Modelleren en implementeren van een ray tracer met behulp van functionele talen

door Kenneth HOSTE

Promotor: prof. dr. ir. R. BOUTE  
Thesisbegeleider: lic. H. VERLINDE

Scriptie ingediend tot het behalen van de graad van licentiaat in de informatica

Academiejaar 2004–2005

## **Voorwoord**

Hierbij wil ik mijn vriendin Joke bedanken voor haar morele steun en bijstand, en ook mijn ouders en grootouders wil ik danken voor hun morele en financiële steun. Zonder hen was het niet mogelijk geweest om deze scriptie en tevens mijn studies tot een goed einde te brengen.

Graag wil ik ook Prof. Boute en Hannes Verlinde danken voor hun advies bij het opstellen van het model en het oplossen van de problemen die daarbij naar voor kwamen. Ook Andy Georges wil ik bedanken voor de praktische tips bij de implementatie en het nalezen van de scriptie.

## **Toelating tot bruikleen**

De auteur geeft de toelating deze scriptie tot consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.

Kenneth Hoste, mei 2005

# Modelleren en implementeren van een ray tracer met behulp van functionele talen

door  
Kenneth HOSTE

Promotor: prof. dr. ir. R. BOUTE  
Thesisbegeleider: lic. H. VERLINDE

Scriptie ingediend tot het behalen van de graad van licentiaat in de informatica

Academiejaar 2004–2005

Faculteit Toegepaste Wetenschappen  
Universiteit Gent

Vakgroep Informatietechnologie  
Voorzitter: prof. dr. ir. P. LAGASSE

## Samenvatting

Met behulp van wiskundige formalisme Funmath stellen we een formeel model op voor een eenvoudige ray tracer. Hierbij maken we gebruik van de generische functionalen die beschikbaar zijn, en maken we dankbaar gebruik van het feit dat tupels beschouwd worden als functies. Op basis van dit model leiden we dan een implementatie af in de functionele programmeertaal Haskell. We voorzien ook een uitbreiding op het model, namelijk het gebruik van de Perlin noise techniek om materialen als hout en marmer voor te stellen. Daarnaast implementeren we ook een eenvoudige parser en een tekstuele en grafische gebruikersinterface.

**Trefwoorden:** ray tracing, 3d, functionele talen, Funmath, Haskell

# Inhoudsopgave

<b>1</b>	<b>Inleiding</b>	<b>1</b>
1.1	Ray tracing . . . . .	2
1.1.1	Het algoritme . . . . .	2
1.1.2	Toepassingen . . . . .	3
1.2	Funmath . . . . .	3
1.3	Haskell . . . . .	4
<b>2</b>	<b>Modelleren van de ray tracer in Funmath</b>	<b>6</b>
2.1	Funmath . . . . .	6
2.1.1	Funcies . . . . .	7
2.1.2	Basisconstructies . . . . .	7
2.1.3	Generische functionalen . . . . .	10
2.1.4	Calculatieve stijl . . . . .	13
2.2	Ray tracing . . . . .	14
2.2.1	Het algoritme . . . . .	14
2.2.2	Wiskundige benadering . . . . .	15
2.2.3	Enkele vereenvoudigingen . . . . .	19
2.3	Stapsgewijze opbouw van het model . . . . .	20
2.3.1	Begrippen . . . . .	20
2.3.2	Functie definities . . . . .	23
2.3.3	Enkele eigenschappen . . . . .	33
<b>3</b>	<b>Implementatie in Haskell</b>	<b>39</b>
3.1	Algemeen . . . . .	39
3.1.1	Haskell syntax . . . . .	40
3.2	De module <code>HRayMath</code> . . . . .	42
3.3	De module <code>HRayEngine</code> . . . . .	47
3.4	De module <code>HRayPerlin</code> . . . . .	54

3.5	De module HRayOutput . . . . .	57
<b>4</b>	<b>Inputspecificatie en gebruikersinterface</b>	<b>59</b>
4.1	Inputspecificatie . . . . .	59
4.2	Tekstuele en grafische gebruikersinterfaces . . . . .	60
4.2.1	Tekstuele gebruikersinterface . . . . .	60
4.2.2	Grafische gebruikersinterface (GUI) . . . . .	62
<b>5</b>	<b>Enkele afbeeldingen</b>	<b>65</b>
5.1	Schaduw van een bol op andere bol . . . . .	65
5.2	Reflectie van bollen in een vlak . . . . .	66
5.3	Transparantie . . . . .	66
5.4	Perlin noise materialen . . . . .	67
5.5	Combinatie van effecten . . . . .	67
<b>6</b>	<b>Conclusies</b>	<b>68</b>
6.1	Verder onderzoek . . . . .	69
6.2	Verwant onderzoek . . . . .	70
<b>A</b>	<b>Haskell implementatie</b>	<b>71</b>
A.1	HRayMath . . . . .	71
A.2	HRayEngine . . . . .	73
A.3	HRayPerlin . . . . .	76
A.4	HRayOutput . . . . .	79
A.5	HRayParser . . . . .	79
A.6	HRayMain . . . . .	84
A.7	HRayGUI . . . . .	85
A.8	PixbufInternals . . . . .	88
	<b>Bibliografie</b>	<b>91</b>

# Hoofdstuk 1

## Inleiding

In deze scriptie presenteren we de ontwikkeling van een ray tracer, waarbij we vooral aandacht besteden aan de structuur ervan. Om een structuur te bekomen die voldoet aan alle eisen (elegant, duidelijk, uitbreidbaar, robuust, ...), maken we in een eerste fase gebruik van Funmath [1, 2, 3], een wiskundig formalisme dat toelaat om het model voor de ray tracer op een abstract niveau te definiëren. Hierbij gaan we declaratief te werk, en kunnen we enkele gewenste eigenschappen van het model bewijzen voor we overgaan tot de eigenlijke implementatie.

In een tweede fase steunen we op het model om een implementatie van de applicatie uit te werken in de functionele programmeertaal Haskell [4, 5, 6, 7]. Deze implementatie volgt grotendeels rechtstreeks uit het model, maar we moeten wel rekening houden met de beperkingen van Haskell t.o.v. Funmath. Om de resulterende ray tracer gebruiksvriendelijk te maken, geven we ook een inputspecificatie en voorzien we een eenvoudige gebruikersinterface, zowel tekstueel als grafisch [8, 9].

In het eerste hoofdstuk bespreken we kort de begrippen ray tracing, Funmath en Haskell ter inleiding. In hoofdstuk 2 bekijken we Funmath wat uitgebreider, en stellen we daarmee een model op voor de ray tracer. We bewijzen ook enkele eenvoudige eigenschappen van het model. Dit model gebruiken we dan in hoofdstuk 3 om de applicatie te implementeren in Haskell, nadat we ook de programmeertaal Haskell wat uitgebreider besproken hebben. In hoofdstuk 4 bespreken we de inputspecificatie en beide gebruikersinterfaces, die toelaten om de ray tracer effectief te gebruiken. Enkele resultaten die we bekomen m.b.v. de geïmplementeerde ray tracer, bekijken we in hoofdstuk 5. In het laatste hoofdstuk besluiten we met een overzicht van de resultaten.

## 1.1 Ray tracing

In de wereld van de computergrafiek wordt er steeds meer belang gehecht aan realistische afbeeldingen. Een krachtig en eenvoudig, maar rekenintensief algoritme als ray tracing wordt daarvoor steeds meer gebruikt, vooral omdat de rekenkracht van computers snel toeneemt en het rekenintensieve aspect dus minder beperkingen stelt.

Ray tracing is een algoritme om van een 3-dimensionale wereld een 2-dimensionale afbeelding te maken [10, 11, 12]. Die wereld bestaat uit objecten (die elk hun eigen materiaal of texture hebben) en een belichting, en wordt bekeken door een fictieve camera. Aan de hand van enkele instellingen, zoals de afstand tussen het oogpunt en de 3-dimensionale wereld, de kijkhoek van de camera (m.a.w. hoeveel van de wereld er zichtbaar is) en de gewenste resolutie van de afbeelding, kan de gebruiker de gewenste afbeelding bekomen.

Een groot voordeel van het ray tracing algoritme is dat effecten als schaduwen, reflectie en transparantie vrij eenvoudig te bekomen zijn, terwijl dat bij andere algoritmen, zoals splatting en scan conversion, vaak heel wat moeilijker is. Ook de bijkomende berekeningen om de kwaliteit van een afbeelding te verbeteren (bv. d.m.v. anti-aliasing, waarbij de scherpe randen van objecten verzacht worden) kunnen vrij eenvoudig in het ray tracing algoritme verwerkt worden.

Zoals reeds aangegeven, is de vereiste rekenkracht bij het uitvoeren van het algoritme het grootste nadeel ervan. Bij andere algoritmen worden bepaalde berekeningen gedeeld over meerdere pixels, terwijl bij ray tracing bij elke pixel met een schone lei wordt gestart. Daardoor gaat er veel informatie verloren die opnieuw moet berekend worden. Er bestaan technieken die proberen om het rekenwerk dat nodig is tot een minimum te beperken, door het bepalen van snijdingen met objecten sneller te laten verlopen (gebruik maken van bounding volumes, efficiënte berekeningen voor ingewikkelde objecten), minder snijdingen te berekenen dan het klassieke algoritme (hierarchy van bounding volumes, ruimtelijke onderverdeling van de wereld, directionele technieken) en door stralen samen te bundelen (beam tracing, pencil tracing) [10]. In deze scriptie zullen we ons echter beperken tot het zuivere ray tracing algoritme.

### 1.1.1 Het algoritme

Het realistische karakter van het ray tracing algoritme wordt verklaard door het vrij nauwkeurig simuleren van het natuurkundige model. In de natuur krijgt een voorwerp kleur doordat er lichtstralen op invallen en het materiaal waaruit het object bestaat een deel van het licht opslorpt, en de rest terugkaatst. Daarnaast zorgen ook andere invloeden (waaronder reflectie van andere voorwerpen in het bekeken voorwerp en (gedeeltelijke)

transparantie van het voorwerp) voor een bijdrage tot de kleur van het voorwerp.

De vereenvoudiging die noodzakelijk is om het algoritme werkbaar te maken, bestaat erin dat enkel die punten die zichtbaar zijn, behandeld worden. Dit gebeurt door stralen te volgen vanuit het oogpunt doorheen de 3D-wereld, en snijdingen te maken van die stralen met de objecten in de 3D-wereld. Die snijdingen worden dan gebruikt om de kleur van de objecten in een bepaald punt te bepalen, en om te controleren of een object zich in de schaduw van een ander object bevindt. Omdat een straal kan beschouwd worden als een wiskundige rechte, kunnen berekeningen gemaakt worden die rekening houden met de hoeken die de rechte maakt met andere rechten (zoals de normaal in het snijpunt), om zo de sterkte van het licht en de reflectie- en brekingsrichting te bepalen. Het natuurkundig model wordt dus eigenlijk omgekeerd: de stralen vertrekken niet vanuit de lichtbronnen, maar vanuit het oogpunt, en worden gevolgd tot aan de lichtbronnen, om op die manier de kleur te bepalen van ieder zichtbaar punt.

In hoofdstuk 2 wordt het algoritme in detail besproken, en definiëren we een formeel model voor een ray tracer.

### 1.1.2 Toepassingen

Er bestaan verschillende toepassingen van het ray tracing algoritme, die allerlei uitbreidingen bevatten. Een bekend voorbeeld is POVRay, een open-source applicatie volledig gebaseerd op het ray tracing algoritme. Ieder jaar wordt een wedstrijd uitgeschreven waarbij de ontwikkelaars van de applicatie de gebruikers uitdagen om het potentieel van POVRay zoveel mogelijk te benutten, wat resulteert in verbazingwekkend realistische afbeeldingen [13].

Een andere belangrijke toepassing van het algoritme komt voor in de wereld van animatiefilms. Zowel Pixar (de animatiestudio achter computeranimatie-klassiekers als Toy Story en Finding Nemo) met hun Renderman applicatie [14], als Blue Sky (die de animatiefilms Ice Age en Robots produceerden) met CGI Studio [15], maken dankbaar gebruik van de kracht van het ray tracing algoritme.

## 1.2 Funmath

Funmath is een declaratief, wiskundig formalisme dat toelaat om op een abstract niveau de functionaliteit van een applicatie te modelleren, en om eigenschappen van dat model te bewijzen. De specificatie kan dan dienen als basis voor de eigenlijke implementatie in een functionele programmeertaal, zoals Haskell.

Het Funmath formalisme [1, 2, 3] is een concrete versie van de Functional Mathematics aanpak, waarbij wiskundige objecten (waar mogelijk) systematisch gedefinieerd worden als functies. Deze aanpak heeft als voordeel dat de verzameling generische functionalen, waarvan we er enkele zullen bespreken, beschikbaar zijn voor elk van deze objecten. Hierdoor moeten de rekenregels voor generische functionalen slechts één keer vastgelegd worden, en bovendien worden op die manier de dubbelzinnigheden en inconsistenties in de klassieke notaties uit de weg geruimd.

In hoofdstuk 2 bespreken we uitgebreid het formalisme: we leggen uit wat functies zijn, wat de basisconstructies zijn van het formalisme, wat generische functionalen zijn en hoe ze kunnen gebruikt worden. Met behulp van de basisconstructies en generische functionalen stellen we dan het model samen dat zal dienen als specificatie voor de eigenlijke implementatie.

### 1.3 Haskell

Haskell is een luie, puur functionele en sterk getypeerde programmeertaal die erg verschillend is van imperatieve programmeertalen zoals Java en C/C++. Door de functionele stijl is de code die geproduceerd wordt korter, duidelijker en makkelijker onderhoudbaar. Bovendien leunt die stijl dichter aan bij de natuurlijke taal van de wiskunde, zodat het neerschrijven van de gewenste functionaliteit minder moeite vergt dan bij imperatieve programmeertalen. Door de sterke typering worden minder makkelijk fouten gemaakt, zodat ook de productiesnelheid groter wordt.

Een programma in Haskell bestaat uit één enkele uitdrukking, en wordt uitgevoerd door de uitdrukking te evalueren. Hierdoor ligt de nadruk op wat er moet bereikt worden, en niet hoe dat doel bereikt kan worden. Haskell ondersteunt polymorfisme, waardoor het type-systeem minder restrictief is dan dat van andere programmeertalen zoals Ada en Pascal. Het luie karakter van Haskell zorgt ervoor dat deeldrukkingen enkel geëvalueerd worden indien ze ook effectief worden gebruikt .

Een belangrijke vereiste voor het schrijven van makkelijk te onderhouden code is het gebruik van abstracties. De ondersteuning van functies van hogere orde of functionalen in Haskell draagt hiertoe bij: functies kunnen als argument doorgegeven worden aan andere functies, kunnen het resultaat zijn van een functie, kunnen opgeslagen worden als datastructuur, enz.

Het abstracte karakter van Haskell kan echter ook een nadeel zijn: bij applicaties waarbij de prestatie erg belangrijk is, zal, afhankelijk van de implementatie (compiler), een imperatieve taal zoals C beter presteren, door manipulatie van de volgorde van evaluatie

en speciale technieken bij geheugenbeheer. Dergelijke laag-niveau programmering komt de duidelijkheid van het programma echter niet ten goede, en duidelijkheid is dan weer een van de sterkte punten van Haskell.

In de laatste jaren is het gebruik van Haskell sterk toegenomen. Een veel gebruikte versie is de Haskell 98 standaard, maar er bestaan verschillende uitbreidingen. Hieronder vervat zijn bindingen met grafische bibliotheken zoals OpenGL en GTK+, krachtige taaluitbreidingen zoals arrows, O'Haskell (object-gerieerde uitbreiding) en Template Haskell (uitbreiding om type-veilige, compileerbare meta-programmatie toe te laten). Ook krachtige applicaties als darcs (een systeem om bvb. code of documentatie, gedeeld door verschillende mensen, up to date te houden), VOP (een applicatie om POV-Ray beschrijvingen te renderen in OpenGL) en interpreters voor de lambda en combinatie calculus, werden ontwikkeld in Haskell.

De meest gebruikte tools om Haskell programma's te schrijven zijn Hugs en GHC. Hugs is een interpreter die vooral bedoeld is voor mensen die hun eerste stappen zetten met Haskell, terwijl GHC een krachtige compiler is. Deze laatste is zelf in Haskell geschreven, levert snelle programma's op en biedt ondersteuning voor verschillende architecturen. Veel uitbreidingen op de Haskell 98 standaard werken enkel met GHC, waardoor deze compiler erg belangrijk is voor het ontwikkelen van omvangrijke programma's. Omdat een interactieve manier van werken, zoals ondersteund wordt door Hugs, heel handig is bij het ontwikkelen van applicaties, wordt er met GHC een interactieve tegenhanger GHCi meegeleverd.

# Hoofdstuk 2

## Modelleren van de ray tracer in Funmath

In dit hoofdstuk stellen we de formele specificatie op voor de ray tracer. Daartoe bespreken we eerst uitgebreid de belangrijkste concepten van het gebruikte Funmath formalisme, waaronder de verschillende basisconstructies en enkele generische functionalen. We leggen het ray tracing algoritme uit en bekijken de wiskundige begrippen en meetkundige bewerkingen die we zullen nodig hebben. Deze informele definitie zetten we dan stap voor stap om naar een formele specificatie [16, 17, 18], waarvan we ook enkele eigenschappen zullen aantonen.

### 2.1 Funmath

Omdat functies zeer belangrijk zijn in het Funmath formalisme [1], bekijken we eerst hoe we een functie kunnen definiëren. Daarna bespreken we de verschillende basisconcepten die beschikbaar zijn. Hierbij besteden we bijzondere aandacht aan de tupeldenotatie, wegens de krachtige eigenschappen van dit concept. We vermelden verschillende operatoren die kunnen gebruikt worden in combinatie met tupels, en geven aan hoe de behandeling van tupels als functies verschillende voordelen oplevert. We beschrijven de definitie van enkele belangrijke generische functionalen die zullen gebruikt worden in het model dat we zullen opstellen, samen met enkele toepassingen ervan. Tenslotte bespreken we nog de calculatoire stijl die we hanteren bij het bewijzen van eigenschappen.

### 2.1.1 Functies

Een functie  $f$  wordt gedefinieerd met behulp van een domeinaxioma van de vorm  $\mathcal{D} f = S$  of  $x \in \mathcal{D} f \equiv p$ , en een beeldaxioma van de vorm  $x \in \mathcal{D} f \Rightarrow f x = e$ . Hierbij is  $S$  een verzameling,  $p$  een propositie en  $e$  een uitdrukking.

Een belangrijk begrip is functiegeïjkheid, dat als volgt formeel wordt uitgedrukt:

$$f = g \Rightarrow \mathcal{D} f = \mathcal{D} g \wedge (e \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f e = g e) \quad (2.1)$$

Om de gelijkheid aan te tonen, kunnen we gebruik maken van de volgende inferentieregel:

$$\frac{(q \Rightarrow) \mathcal{D}(f) = \mathcal{D}(g) \wedge (x \in \mathcal{D}(f) \cup \mathcal{D}(g) \Rightarrow f x = g x)}{(q \Rightarrow) f = g} \quad (2.2)$$

Een functie van hogere orde of functionaal is een functie waarvan het argument of het resultaat een functie is. Indien we dergelijke functionalen op een generische manier definiëren, kunnen ze onveranderd gebruikt worden in verschillende contexten, zonder de argumenten beperkingen op te leggen.

We bekijken nu reeds enkele triviale gevallen: de constante functie  $\bullet$ , de lege functie  $\varepsilon$  en de eenpunts-functie  $\mapsto$ :

$$\begin{aligned} X \bullet e &= v : X . e \\ \varepsilon &= \emptyset \bullet e \\ d \mapsto e &= \iota d \bullet e \end{aligned}$$

Hierbij is  $X$  een willekeurige verzameling en  $e$  een uitdrukking waarin  $v$  niet vrij voorkomt. De singleton operator  $\iota$  wordt gedefinieerd door het axioma  $y \in \iota x \equiv y = x$ . De abstractie-notatie  $v : X . e$  en niet-triviale generische functionalen worden later besproken.

### 2.1.2 Basisconstructies

In het Funmath formalisme zijn er slechts 4 basisconstructies: identifier, functie-abstractie, functie-applicatie en tupeldenotatie. Door deze constructies te combineren, kunnen we elke wiskundige theorie synthetiseren zonder dubbelzinnigheden.

Identifiers worden gedeclareerd door bindingen van de vorm  $i : X \wedge p$ , waarbij de zogenaamde filter  $\wedge p$  optioneel is, en waarmee we vastleggen dat  $i \in X \wedge p$ . Omdat de relationele operator  $\in$  tot dubbelzinnigheden leidt als we die ook gebruiken voor binding, schrijven we  $i : X \wedge p$  voor de declaratie en  $i \in X \wedge p$  voor de propositie waaraan de gedeclareerde  $i$  voldoet. Bekende symbolen, zoals klassieke notaties voor getallenverza-

melingen ( $\mathbb{N}$ ,  $\mathbb{R}$ ,  $\mathbb{B}$ ), bewerkingen ( $+$ ,  $-$ ) en relaties ( $<$ ,  $>$ ,  $\Rightarrow$ ) worden als constanten beschouwd. Definities van de vorm **def** *binding* introduceren identifiers met een globaal bereik, die we constanten noemen.

Een functie-abstractie is van de vorm *binding*.*e*, en drukt een functie uit. Wanneer we de abstractie  $v : X \wedge p . e$  verkort noteren als  $f$ , dan wordt het domeinaxioma gegeven door  $d \in \mathcal{D}f \equiv d \in X \wedge p[d]_d^v$  en het beeldaxioma door  $d \in \mathcal{D}f \Rightarrow f d = e[d]_d^v$ , waarbij  $e[d]_d^v$  staat voor  $e$  waarin  $v$  op uniforme wijze vervangen wordt door  $d$ . Varianten hierop zijn  $e \mid v : X$  wat staat voor  $v : X . e$  en  $v : X \mid p$  wat staat voor  $v : X \wedge p . v$ . Deze laten toe om, samen met de notatie voor functiebereik  $\{—\}$ , de klassieke verzamelingnotaties formeel en ondubbelzinnig te synthetiseren, zoals in  $\{2 \cdot x \mid x : \mathbb{N}\}$  en  $\{x : \mathbb{R} \mid x > 0\}$ .

De standaard notatie voor functie applicatie is prefix, zoals bvb.  $f x$ . Andere mogelijke notaties zijn infix (bvb.  $x + y$ ) en superfix (bvb.  $M^T$ ). De niet-standaard affix notatie van een functie wordt met behulp van streepjes ( $—$ ) aangegeven bij de specificatie ervan. We merken op dat functie-applicatie links-associatief is, zodat  $f a b$  gelijk is aan  $(f a) b$ , en dat partiële applicatie toegelaten is voor elke infix operator  $\star$ , zodat  $a \star$  en  $\star b$  monadische functies zijn waarbij  $(a \star) b = a \star b$  en  $(\star b) a = a \star b$ .

Een tupeldenotatie is een lijst van minstens twee uitdrukkingen, gescheiden door komma's, zoals bvb.  $(e, e', e'')$ . Tupels, en in het bijzonder sequenties (tupels waarbij alle elementen hetzelfde type hebben), worden gezien als functies.

Het domein van een tupel met lengte  $n$  is  $\square n$ , waarbij we de operator  $\square$  als volgt kunnen definiëren:

$$\mathbf{def} \square : \mathbb{N}' \rightarrow \mathcal{P} \mathbb{N} \mathbf{with} \square n = \{m : \mathbb{N} \mid m < n\}$$

waarbij  $\mathbb{N}' := \mathbb{N} \cup \iota \infty$  en  $\mathcal{P} S$  de verzameling is van alle deelverzamelingen van een gegeven verzameling  $S$ . De verzamelingnotatie  $\{—\}$  zullen we later formeel definiëren als functiebereik. We merken op dat  $\square 0 = \emptyset$  en  $\square \infty = \mathbb{N}$ .

Het beeld van een lijstje als functie is een element van dat lijstje, bijvoorbeeld

$$\mathcal{D}(e, e', e'') = \square 3 \quad \wedge \quad (e, e', e'') 0 = e \wedge (e, e', e'') 1 = e' \wedge (e, e', e'') 2 = e''$$

Voor veelzijdige functies zoals tupels en sequenties zijn veel nuttige operatoren gedefiniëerd, waarvan we de belangrijkste vermelden:

- de lengte-operator ( $\#$ ) met als axioma  $\mathcal{D}x = \square (\# x)$  voor elk tupel  $x$
- het lege tupel (of de lege sequentie)  $\varepsilon$ , gedefiniëerd als  $\varepsilon = \emptyset \bullet e$

- de singleton operator ( $\tau$ ), met als axioma  $\tau x = 0 \mapsto x$ , voor elke  $x$
- de shift operator ( $\sigma$ ), met als axioma's  $\#(\sigma x) = \#x - 1$  en  $\sigma x n = x(n + 1)$
- de concatenatie operator ( $- ++ -$ ) is gedefinieerd voor elk tupel  $x$  en  $y$ , met als domein axioma's:

$$\begin{aligned} \#(x ++ y) &= \#x + \#y \\ \forall i : \mathcal{D}(x ++ y) . (x ++ y) i &= (i < \#x) ? x i \dagger y(i - \#x) \end{aligned}$$

- de prefixing ( $- \succ -$ ) en postfixing ( $- \prec -$ ) operators, met als axioma's  $a \succ x = \tau a ++ x$  en  $x \prec a = x ++ \tau a$

Sequentietypes worden gedefinieerd als verzamelingen van functies, waarbij de machtsoperator ( $\uparrow$ ) als volgt uitgebreid wordt over verzamelingen: voor elke verzameling  $A$  en  $n : \mathbb{Z}'$  definiëren we  $A \uparrow n = \square n \rightarrow A$ , waarbij we  $A \uparrow n$  vaak verkort noteren als  $A^n$ . Op die manier bekomen we volgende classificatie van sequentietypes:

$A^n$  is de verz. van sequenties van lengte  $n$  over  $A$ , dus  $A^n = \square n \rightarrow A$

$A^*$  is de verz. van eindige sequenties (of lijsten) over  $A$ , dus  $A^* = \bigcup n : \mathbb{N} . A^n$

$A^\infty$  is de verz. van oneindige sequenties (of stromen) over  $A$ , dus  $A^\infty = \mathbb{N} \rightarrow A$

$A^\omega$  is de verz. van sequenties over  $A$ , dus  $A^\omega = A^* \cup A^\infty$

Om een interval aan te duiden, gebruiken we de notatie  $[a, b]$ . Een interval is steeds een deelverzameling van  $\mathbb{R}$ , en kunnen we als volgt definiëren:

$$[a, b] = \mathbb{R} \downarrow ((\geq a) \wedge (\leq b))$$

Hierbij is de filter  $\downarrow$  voor verzamelingen gedefinieerd als  $X \downarrow P = \{x : X \cap \mathcal{D}P \mid Px\}$ , waarbij  $P$  een predikaat is en we  $X \downarrow P$  verkort kunnen noteren als  $X_P$ .

Conditionele uitdrukkingen zijn van de vorm  $c ? b \dagger a$  (en kunnen we lezen als “indien  $c$ , dan  $b$ , zoniet  $a$ ”). We kunnen ze als volgt definiëren met behulp van tupeldenotatie:

$$c ? b \dagger a = (a, b) c$$

Met deze definitie levert de conditionele uitdrukking  $a$  op als  $c = 0$  (vals) en  $b$  als  $c = 1$  (waar), wat overeenkomt met de intuïtieve betekenis.

Een predikaat is een functie waarvan het beeld een element is van  $\mathbb{B}$  (dit is  $\{0, 1\}$ ). De kwantoren  $\forall$  en  $\exists$  zijn predikaten over predikaten, en zijn voor een willekeurig predikaat  $P$  gedefinieerd door

$$\forall P \equiv P = \mathcal{D}P \bullet 1 \quad \text{en} \quad \exists P \equiv P \neq \mathcal{D}P \bullet 0$$

Het bereik van een functie  $f$  wordt genoteerd als  $\mathcal{R}f$ , met  $\{f\}$  als alternatieve notatie, en wordt gedefinieerd door

$$y \in \mathcal{R}f \equiv \exists(x: \mathcal{D}f . y = fx)$$

Het samenvoegen van verzamelingen gebeurt zoals gewoonlijk met behulp van de unie  $\cup$ , die we kunnen definiëren met behulp van het axioma  $x \in X \cup Y \equiv x \in X \vee x \in Y$ . Voor een familie  $T$  van verzamelingen definiëren we  $\bigcup$  door

$$x \in \bigcup T \equiv \exists i: \mathcal{D}T . x \in Ti$$

Dit kunnen we zien als een veralgemening van de infix-operator  $\cup$ , gezien  $\bigcup(X, Y) = X \cup Y$  voor verzameling  $X$  en  $Y$ . We noemen  $\bigcup$  daarom de elastische uitbreiding van  $\cup$ . Analoog kunnen we  $\sum$  definiëren en aantonen dat  $\sum(x, y) = x + y$  voor getallen  $x$  en  $y$ .

Algemeen, stel dat  $F$  de elastische uitbreiding is van een infix-operator  $\star$ . Dan gebruiken we  $F$  als definitie voor de *variadische applicatie* van  $\star$ , waarmee functie-applicatie bedoeld wordt waarin  $\star$  afwisselt met argumenten, bvb.  $e \star e' \star e''$ . We definiëren dergelijke uitdrukkingen dan als

$$e \star e' \star e'' = F(e, e', e'')$$

Voorbeelden hiervan zijn  $X \cup Y \cup Z$  en  $x + y + z$ , die we respectievelijk definiëren als  $\bigcup(X, Y, Z)$  en  $\sum(x, y, z)$ .

### 2.1.3 Generische functionalen

Een functionaal noemen we generisch als die onveranderd gebruikt kan worden in verschillende contexten, zonder restricties te moeten opleggen aan de argumenten. We bekijken enkele belangrijke generische functionalen die we later nodig zullen hebben in ons model.

#### Functie-compositie ( $— \circ —$ )

In de traditionele definitie van de functie-compositie  $f \circ g$  vereist dat  $f$  en  $g$  voldoen aan  $\mathcal{R}g \subseteq \mathcal{D}f$ , opdat  $f \circ g$  gedefinieerd zou zijn. Wij definiëren  $f \circ g$  door middel van

de abstractie  $f \circ g = x : \mathcal{D}g \wedge gx \in \mathcal{D}f . f(gx)$  of equivalent door gescheiden domein en beeld axioma's:

$$\begin{aligned} x \in \mathcal{D}(f \circ g) &\equiv x \in \mathcal{D}g \wedge gx \in \mathcal{D}f \\ x \in \mathcal{D}(f \circ g) &\Rightarrow (f \circ g)x = f(gx) \end{aligned}$$

Met deze definitie is de eis  $\mathcal{R}g \subseteq \mathcal{D}f$  niet langer nodig, omdat het domein van het resultaat gedefinieerd wordt via het domein van beide argumenten. De functie-compositie  $f \circ g$  is op die manier enkel gedefinieerd voor elementen die tot het domein van  $f \circ g$  behoren, zodat er geen beperking op de argumenten  $f$  en  $g$  nodig is.

### Directe uitbreiding ( $\widehat{\quad}$ )

De directe uitbreidingsoperator  $\widehat{\quad}$  wordt ingevoerd om klassieke operatoren uit te breiden over functies. Voor eender welke dyadische operator  $\star$  definiëren we  $\widehat{\star}$  zodat, voor functies  $f$  en  $g$ ,  $f \widehat{\star} g$  een functie is gedefinieerd door:

$$f \widehat{\star} g = x : \mathcal{D}f \cap \mathcal{D}g \wedge (fx, gx) \in \mathcal{D}(\star) . (fx) \star (gx)$$

Een illustratief voorbeeld hiervan is de directe uitbreiding van de gelijkheid, die toelaat om functies puntsgewijs te vergelijken via de beeldpunten.

$$(f \widehat{=} g) = x : \mathcal{D}f \cap \mathcal{D}g . fx = gx$$

Hierbij steunen we op  $(fx, gx) \in \mathcal{D}(=)$  omdat  $=$  universeel is.

Merk op dat de gelijkheid van de domeinen hier geen vereiste is, en er een significant verschil bestaat tussen functie-gelijkheid en gelijkheid van de beelden. We kunnen de eerder gedefinieerde functie-gelijkheid uitdrukken m.b.v. de directe uitbreiding van de gelijkheid:

$$f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge \forall (f \widehat{=} g)$$

### Half-directe uitbreiding ( $\overleftarrow{\quad}$ en $\overrightarrow{\quad}$ )

Wanneer we wensen slechts één van de argumenten van een dyadische operator  $\star$  uit te breiden tot een functie  $f$ , hebben we voldoende aan de half-directe uitbreiding  $f \overleftarrow{\star} e$  of  $e \overrightarrow{\star} f$ , die we als volgt kunnen definiëren:

$$\begin{aligned} f \overleftarrow{\star} e &= f \widehat{\star} (\mathcal{D}f \bullet e) \\ e \overrightarrow{\star} f &= (\mathcal{D}f \bullet e) \widehat{\star} f \end{aligned}$$

We geven een eenvoudig voorbeeld:

$$\begin{aligned}
3 \overset{\rightarrow}{\vdash} (2, 4) &= \langle \mathbf{def} \overset{\rightarrow}{\vdash} \rangle && (\mathcal{D}(2, 4) \bullet 3) \overset{\widehat{\rightarrow}}{\vdash} (2, 4) \\
&= \langle \mathbf{def} \overset{\widehat{\rightarrow}}{\vdash} \rangle && i : \square 2 . (\square 2 \bullet 3) i + (2, 4) i \\
&= \langle \mathbf{def} \square \rangle && (\square 2 \bullet 3) 0 + (2, 4) 0, (\square 2 \bullet 3) 1 + (2, 4) 1 \\
&= \langle \mathbf{def} \bullet, (a, b) c = c ? a \dagger b \rangle && 3 + 2, 3 + 4 \\
&= \langle \text{uitwerken} \rangle && 5, 7
\end{aligned}$$

We zien dat we een nieuwe lijstje bekomen dat bestaat uit het originele lijstje waarbij bij elk element 3 werd opgeteld.

### Functie-versmelting ( $\cup$ )

Twee functies  $f$  en  $g$  kunnen samengesmolten worden met behulp van de functie-versmelting  $\cup$ , zonder beperkingen op te leggen aan deze functies. De resulterende functie  $f \cup g$  wordt gedefinieerd door de volgende axioma's:

$$\begin{aligned}
x \in \mathcal{D}(f \cup g) &\equiv x \in \mathcal{D}f \cup \mathcal{D}g \wedge (x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f x = g x) \\
x \in \mathcal{D}(f \cup g) &\Rightarrow (f \cup g) x = x \in \mathcal{D}f ? f x \dagger g x
\end{aligned}$$

De functie-versmelting wordt dikwijls gebruikt in combinatie met de eenpunts-functie  $\mapsto$ , wat toelaat om tupels expliciet als functie te schrijven. Een tupel  $(a, b)$  (met als domein  $\square 2$ ) kunnen we dan schrijven als  $(0 \mapsto a \cup 1 \mapsto b)$ . Hierbij beschouwen we het tupel als een samentrekking van twee eenpunts-functies: één die de waarde 0 afbeeldt op  $a$  en één die de waarde 1 afbeeldt op  $b$ .

### Veralgemeend functioneel cartesiaans product ( $\times$ )

Het veralgemeend functionele cartesiaans product laat  $\times$  toe om verfijnde functie-typering te specificeren. Deze functionaal werkt in op een familie verzamelingen  $F$  (zodat  $F$  een functie is en  $F x$  voor elke  $x$  in  $\mathcal{D}F$  een verzameling) en wordt gedefinieerd door:

$$\times F = \{f : \mathcal{D}F \rightarrow \bigcup F \mid \forall x : \mathcal{D}f \cap \mathcal{D}F . f x \in F x\}$$

Het klassieke cartesiaans product  $x, y \in X \times Y \equiv x \in X \wedge y \in Y$  kunnen we dan definiëren als  $X \times Y = \times (X, Y)$ . Ook de functie-typering  $X \rightarrow Y$  kunnen we nu formeel definiëren als  $\times (X \bullet Y)$ . Soms noteren we  $\times v : X . Y_v$  verkort als  $X \ni v \rightarrow Y_v$ , waarbij  $Y$  een functie is van  $v$ , en  $Y_v$  hetzelfde is als de functie-applicatie  $Y v$ . dat  $Y$  afhankelijk *kan* zijn van  $v$ .

### Linker reductie ( $\leftarrow \rightarrow \_ \rightarrow$ )

Deze functionaal werd gedefinieerd om een lijst te reduceren onder een infix operator  $\star$  met een identiteitselement  $e$  [3]. De linker reductie functionaal  $\rightarrow$  werkt in op 3 argumenten: een dyadische functie  $\star$ , een element  $e$  en een lijstje. Het type van deze functie van hogere orde en de definitie zijn gegeven door:

$$\begin{aligned} \mathbf{def} \rightarrow & : (X \rightarrow Y \rightarrow X) \rightarrow X \rightarrow Y^* \rightarrow X \\ \mathbf{with} \quad (\star) & \rightarrow_e \varepsilon = e \\ \wedge \quad (\star) & \rightarrow_e (a \succ l) = \star \rightarrow_{(e \star a)} l \end{aligned}$$

waarbij  $X$  en  $Y$  willekeurige types zijn. Om de werking van deze functionaal te verduidelijken, werken we een eenvoudig voorbeeld uit. Om de som van de elementen van een lijstje te bepalen, kunnen we gebruik maken van de elastische uitbreiding  $\sum$  van de som. We kunnen dit echter ook als volgt uitdrukken m.b.v. de linker reductie functionaal:

$$\sum = (+) \rightarrow_0$$

We illustreren dit met een voorbeeld:

$$\begin{aligned} (+) \rightarrow_0 (4, -1, 2) &= \langle \mathbf{def} \rightarrow \rangle \quad (+) \rightarrow_{(0+4)(-1,2)} \\ &= \langle \mathbf{def} \rightarrow \rangle \quad (+) \rightarrow_{((0+4)+-1)} (2) \\ &= \langle \mathbf{def} \rightarrow \rangle \quad (+) \rightarrow_{((0+4+(-1))+2)} \varepsilon \\ &= \langle \mathbf{def} \rightarrow \rangle \quad 0 + 4 + (-1) + 2 \\ &= \langle \text{uitwerken} \rangle \quad 4 + (-1) + 2 \\ &= \langle \mathbf{def} \sum \rangle \quad \sum (4, -1, 2) \end{aligned}$$

### 2.1.4 Calculationele stijl

Bij het bewijzen van eigenschappen en het uitwerken van voorbeelden hanteren we steeds een calculationele stijl. Hierbij wordt de verantwoording van stappen duidelijk aangegeven met behulp van  $\langle \text{verantwoording} \rangle$ . Op deze manier zijn bewijzen en uitwerkingen steeds van de vorm:

$$\begin{aligned} p_0 & R_0 \langle \text{verantwoording}_0 \rangle p_1 \\ & R_1 \langle \text{verantwoording}_1 \rangle p_2 \end{aligned}$$

Hierbij zijn  $R_i$  de schakels ( $\leq, =, \Rightarrow, \equiv$ ) en  $p_i$  de verschillende stappen in de uitwerking.

Bij deze stijl wordt herhaling van uitdrukkingen vermeden, en ligt de nadruk op de redenering die wordt gevolgd. Met behulp van verschillende theorema's wordt de calculationele stijl waarbij uitdrukking elkaar opvolgen geformaliseerd [1].

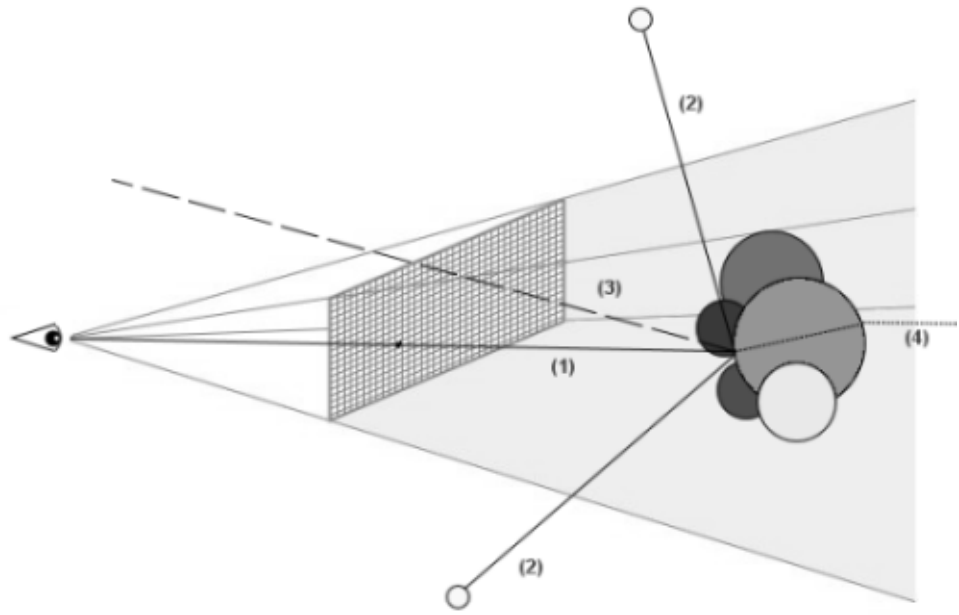
## 2.2 Ray tracing

Voor we de formele specificatie opstellen voor een ray tracer, is het noodzakelijk inzicht te krijgen in de werking van het algoritme. We bekijken eerst het algoritme in pseudo-code, waarna we de wiskunde achter het algoritme bespreken [10, 11, 12]. We vermelden ook enkele conventies die we hanteren, die er toe bijdragen dat het model niet nodeloos veel (technische) details bevat.

### 2.2.1 Het algoritme

Het principe van het ray tracing algoritme is vrij eenvoudig, en is daardoor beknopt te beschrijven in onderstaande pseudo-code. Omdat het algoritme gebaseerd is op stralen die doorheen de wereld ‘geschoten’ worden, kan het ook grafisch duidelijk voorgesteld worden. De verschillende stappen van het algoritme zijn genummerd en aangegeven op figuur 2.1, waarop ook duidelijk het oogpunt, het venster, de belichting en de objecten getoond worden. Het grijze gebied duidt de ruimte aan die zichtbaar is door het venster.

```
voor elke pixel van afbeelding
  construeer straal vanuit oogpunt door pixel (1)
  (*) bepaal dichtstbijzijnde snijding van straal met 3D-wereld
  indien geen snijding
    kleur pixel met achtergrondkleur
  anders
    voor elke lichtbron in 3D-wereld
      construeer straal vanuit snijpunt naar lichtbron (2)
      indien snijding van straal met object
        geen kleurbijdrage door deze lichtbron (schaduw)
      anders
        bepaal kleurbijdrage van lichtbron, houd die bij
    als oppervlak reflectief
      construeer reflectie-straal, bepaal bijdrage recursief (*) (3)
    als oppervlak transparant
      construeer gebroken straal, bepaal bijdrage recursief (*) (4)
    bepaal kleur van pixel aan de hand van som van bijdragen
```



Figuur 2.1: Het ray tracing algoritme stap voor stap

## 2.2.2 Wiskundige benadering

### Basisbewerkingen

We bespreken enkele basisbewerkingen die komen kijken bij het werken met vectoren en punten in de ruimte.

Het modelleren van de som, het verschil en het product van tripels bespreken we kort. Deze zijn namelijk directe toepassingen van de directe uitbreiding, zodat we de som van twee tripels definiëren als  $\hat{+}$ , het verschil als  $\hat{-}$  en het product als  $\hat{\cdot}$ . Ook de vermenigvuldiging van een tripels met een reëel getal  $a$  kunnen we, met behulp van de half-directe uitbreiding, eenvoudig definiëren als  $(\vec{\cdot} a)$ .

Het scalair product  $\text{---}\bullet\text{---}$  van twee tripels kunnen we, m.b.v. de elastische uitbreiding van de som, als volgt definiëren:

$$\mathbf{def} \bullet : \mathbb{R}^3 \rightarrow \mathbb{R}^3 \rightarrow \mathbb{R}$$

$$\mathbf{with} v \bullet w = \sum (v \hat{+} w)$$

## Stralen en objecten

Zoals blijkt uit de pseudo-code vormen stralen het centrale begrip. We beschouwen deze stralen als halfrechten (oneindige lijnstukken in één richting vanaf een bepaald punt), waardoor we eenvoudig snijdingen kunnen bepalen met de objecten uit de 3D-wereld. Bovendien laat deze manier van werken toe om ook met hoeken tussen twee stralen te werken, zoals nodig zal blijken bij de berekening van de kleurbijdrage van een bepaalde lichtbron.

Een straal bestaat uit een startpunt  $s$  en een 3-dimensionale richting  $\mathbf{d}$ , voorgesteld door een vector. Een vector  $\mathbf{v}$  kunnen we specificeren a.d.h.v. het tupel van coördinaten  $v$ . Uit die coördinaten kunnen we de vector bepalen m.b.v. volgende uitdrukking:



$$\mathbf{v} = \sum (v \hat{\cdot} (\mathbf{i}, \mathbf{j}, \mathbf{k}))$$

waarbij  $\mathbf{i}, \mathbf{j}$  en  $\mathbf{k}$  de eenheidsvectoren zijn van het assenstelsel, we een tupel van getallen componentsgewijs vermenigvuldigen met een tupel van vectoren en die vectoren dan optellen. We geven een voorbeeld:

$$\begin{aligned} \sum ((3, 2, 1) \hat{\cdot} (\mathbf{i}, \mathbf{j}, \mathbf{k})) &= \sum (1 \cdot \mathbf{i}, 2 \cdot \mathbf{j}, 3 \cdot \mathbf{k}) \\ &= 1 \cdot \mathbf{i} + 2 \cdot \mathbf{j} + 3 \cdot \mathbf{k} \end{aligned}$$

Een straal  $r$  kunnen we beschrijven als een functie afhankelijk van een reëel getal. Met elk positief reëel getal  $k$  zal dan een punt op de straal overeenkomen, dat we kunnen bepalen met de volgende definitie voor een willekeurige straal  $r$  met startpunt  $s$  en richting  $d$ :

$$\begin{aligned} \mathbf{def} \quad r : \mathbb{R}_{\geq 0} &\rightarrow \mathbb{R}^3 \\ r \ k &= s \hat{+} (d \hat{\cdot} k) \end{aligned}$$

Deze definitie kunnen we ook expliciet met de x-, y- en z-coördinaten uitschrijven:

$$\begin{aligned} (r \ k) \ 0 &= (s \ 0) + (d \ 0) \cdot k \\ (r \ k) \ 1 &= (s \ 1) + (d \ 1) \cdot k \\ (r \ k) \ 2 &= (s \ 2) + (d \ 2) \cdot k \end{aligned}$$

Om de snijdingsberekeningen eenvoudiger te maken, zorgen we ervoor dat de richtingsvector van de straal steeds genormaliseerd is. Dit betekent dat de lengte van de vector, die we kunnen bepalen a.d.h.v. de coördinaten, steeds gelijk is aan 1. Voor een vector  $\mathbf{v}$  met coördinaten  $(a, b, c)$  wordt de lengte  $|\mathbf{v}|$  bepaald door  $\sqrt{a^2 + b^2 + c^2}$ .

Om het model (en de implementatie) nietodeloos ingewikkeld te maken, beschouwen we slechts twee soorten objecten: bollen en vlakken. Zoals later zal blijken zijn deze objecten reeds voldoende om indrukwekkende afbeeldingen te genereren.

Een bol kunnen we specificeren met behulp van het centrum  $m$  en de radius  $R: \mathbb{R}$ , zodat de coördinaten van een punt  $p$  op het oppervlak moeten voldoen aan de uitdrukking:

$$\sum (.^2 \circ (p \hat{-} m)) = R^2$$

Een vlak kunnen we beschrijven met behulp van de normaal op het vlak in de oorsprong, aangeduid door een genormaliseerde vector met tupel coördinaten  $n$ , en de loodrechte afstand  $t$  van het vlak tot de oorsprong. De coördinaten van elk punt  $p$  van het vlak moeten dan voldoen aan de volgende uitdrukking:

$$(n \cdot p) = t$$

### Bepalen van snijdingen

De dichtste snijding van een straal met een object bepalen we door de uitdrukkingen voor de coördinaten van de straal te vervangen in de uitdrukkingen voor het specifieke object, en de bekomen vergelijking op te lossen. Indien er meerdere oplossingen zijn, bepalen we de dichtste oplossing, zijnde het snijpunt dat het dichtst bij het startpunt van de straal ligt.

Indien we dit toepassen op een straal  $r$  met startpunt  $s$  en richtingscoördinaten  $d$ , een bol met centrum  $m$  en radius  $R$  en een vlak met normaal-coördinaten  $n$  en loodrechte afstand  $t$  van oorsprong tot het vlak, bekomen we volgende uitdrukkingen:

$$\sum ((.^2 \circ (r k \hat{-} m)) = R^2 \tag{2.3}$$

$$n \cdot r k = t \tag{2.4}$$

Na het uitwerken van de vergelijking (2.3) voor de snijdingen van een bol met een straal, bekomen we volgende vierkantsvergelijking:

$$A \cdot k^2 + B \cdot k + C = 0$$

$$\begin{aligned} A &= \sum (.^2 \circ d) \\ B &= 2 \cdot (d \cdot (s \hat{=} m)) \\ C &= \sum (.^2 \circ (s \hat{=} m)) + R^2 \end{aligned}$$

Omdat de richtingsvector  $(a, b, c)$  van de straal genormaliseerd is, weten we dat  $A = 1$ . Na het oplossen van de vierkantsvergelijking bekomen we volgende uitdrukking:

$$k = \frac{-B \pm \sqrt{B^2 - 4 * C}}{2}$$

Omdat de vergelijking van een vlak een lineair is, is het bepalen van een snijpunt van een rechte en een oneindig vlak een stuk eenvoudiger. Na het uitwerken van (2.4) bekomen we rechtstreeks de gewenste uitdrukking:

$$k = -\frac{n \cdot s + t}{n \cdot d}$$

Voor we het snijpunt bepalen, moeten we nagaan of er wel een snijding bestaat van de straal met het vlak. Hiertoe bepalen we de waarde van het scalair product  $n \cdot d$ : indien deze waarde 0 is, dan ligt de rechte evenwijdig met het vlak, en is er dus geen snijding. Zonder deze controle zou het bepalen van de oplossing resulteren in een nuldeling.

Om een snijpunt van een straal met een object te bekomen, bepalen we het beeldpunt van de functie  $r$  die de straal voorstelt. Het bepalen van het beeldpunt van negatieve waarden zal wiskundig wel kloppen, maar heeft voor ons weinig zin: op die manier verkrijgen we een punt buiten de straal, omdat we een straal beschouwen als een halfrechte.

## Reflectie en transparantie

Naast het bepalen van snijdingen moeten we ook de reflectie- en brekingsrichting van een straal kunnen bepalen, indien het materiaal waaruit het object bestaat dergelijke effecten toelaat. Het bepalen van de coördinaten van de richting van reflectie is vrij eenvoudig: het volstaat om te beschikken over de coördinaten van de richting  $i$  van de binnenkomende straal en de coördinaten van de normaalrichting  $n$  in het snijpunt, om die richting als volgt te bepalen:

$$i \hat{=} 2 \cdot (n \cdot i) \hat{=} n$$

Eens deze richting bepaald is, kunnen we samen met het gevonden snijpunt de gereflecteerde straal beschrijven. De gereflecteerde straal zal dezelfde hoek  $\alpha$  met de normaal

in het snijpunt vormen als de inkomende straal.

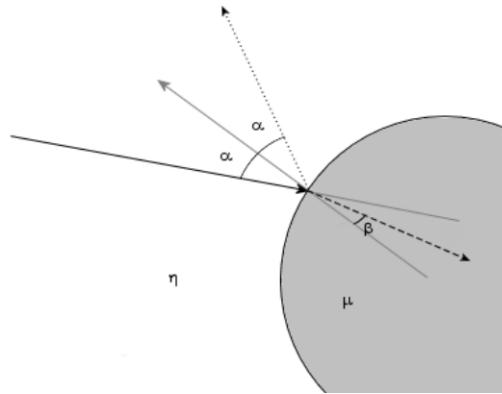
Om de brekingsrichting te bepalen, moeten we de hoek  $\alpha$  tussen de inkomende straal en de normaal in het snijpunt transformeren in een hoek  $\beta$  tussen de normaal en de gebroken straal. De verhouding tussen beide hoeken wordt vastgelegd door de refractie-index  $\eta$  van het materiaal waarin de inkomende straal zich bevindt, en de refractie-index  $\mu$  van het materiaal waarin de straal binnendringt. Deze verhouding wordt gegeven door de wet van Snell [10]:

$$\frac{\sin \alpha}{\sin \beta} = \frac{\mu}{\eta}$$

De brekingsrichting kunnen we bepalen a.d.h.v.  $i$ ,  $n$  en de refractie-indices  $\eta$  en  $\mu$ :

$$\frac{\eta}{\mu} \vec{i} \hat{+} n \vec{n} \hat{=} \left( \frac{\eta}{\mu} \cdot \cos \alpha - \sqrt{1 + \frac{\eta^2}{\mu^2} \cdot ((\cos \alpha)^2 - 1)} \right)$$

Samen met het snijpunt kunnen we de gebroken straal beschrijven. In figuur 2.2 stellen we de gereflecteerde straal en gebroken straal grafisch voor.



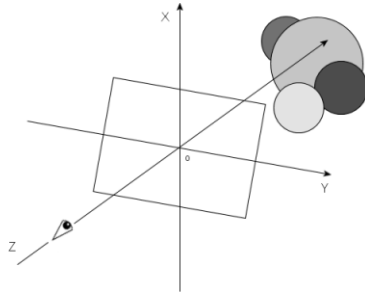
Figuur 2.2: Reflectie en transparantie van een straal bij een boloppervlak

### 2.2.3 Enkele vereenvoudigingen

Voor we het eigenlijke model presenteren, vermelden we enkele vereenvoudigingen die we gehanteerd hebben bij het opstellen van het model. Deze vereenvoudigingen zorgen er wel dat de essentie van het model niet verloren gaan in de complexiteit ervan.

Bij het construeren van een 3D-wereld waarvan we een afbeelding wensen te genereren, moeten we naast de objecten in de 3D-wereld ook de plaats van het oogpunt en de dimensies van het kijkvenster bepalen. Om de berekeningen eenvoudiger en korter te maken, leggen we de plaats van het oogpunt vast op de  $Z$ -as van het cartesische assenstelsel dat gehanteerd wordt. Op die manier zullen de  $x$ - en  $y$ -coördinaat van het oogpunt steeds 0

zijn, wat resulteert in een vereenvoudiging bij het bepalen van de straal door elke pixel van het kijkvenster. Bovendien leggen we het kijkvenster eveneens vast, namelijk in het XY-vlak, zodat de z-coördinaat van een kijkvenster-pixel steeds 0 bedraagt. Het middelpunt van de afbeelding zal zich steeds bevinden in de oorsprong van het assenstelsel.



Figuur 2.3: Het oogpunt en het kijkvenster in het assenstelsel

Op het eerste zicht lijkt dit misschien een beperking: wat als we het oogpunt horizontaal of verticaal willen verplaatsen, om de 3D-wereld vanuit een ander perspectief te bekijken bijvoorbeeld? Of wat als we besluiten dat de afbeelding er beter zou uitzien na een draaiing van 90 graden? Alle mogelijke transformaties van het oogpunt en het kijkvlak kunnen we vertalen in een inverse transformatie van de objecten in de 3D-wereld. Het verplaatsen van het oogpunt naar *links* zal resulteren in een afbeelding die identiek is wanneer we de objecten over een zelfde afstand naar *rechts* verplaatsen.

Zoals reeds vermeld, zullen we slechts ondersteuning bieden voor 2 soorten objecten: bollen en vlakken. Dit is natuurlijk een beperking die niet kan ontweken worden zonder die extra ondersteuning effectief te gaan modelleren en implementeren, maar er werd wel voor gezorgd dat die extra ondersteuning eenvoudig kan toegevoegd worden.

## 2.3 Stapgewijze opbouw van het model

We bespreken stap voor stap het model voor de raytracer applicatie [16, 17, 18]. We definiëren de verschillende begrippen formeel, beschrijven enkele hulpfuncties, formaliseren de wiskundige functionaliteit en modelleren de kern van het raytracing algoritme. We bewijzen ook enkele eigenschappen van het model m.b.v. een calculationele stijl.

### 2.3.1 Begrippen

Een eerste stap in het opbouwen van het model bestaat erin om de verschillende begrippen (straal, object, afbeelding, ...) die gehanteerd worden formeel vast te leggen. Dit is belangrijk omdat het volledige model op deze begrippen zal steunen.

We beschouwen meestal abstracte verzamelingen voor de nodige begrippen, waarvan we meestal de elementen niet verder specificeren. Bij de implementatie van het model zullen we deze verzamelingen natuurlijk wel concreet moeten specificeren.

Omdat we werken in een 3-dimensionale wereld, moeten we een punt in de ruimte kunnen voorstellen. Een punt kunnen we dus beschrijven door middel van de x-, y- en z-coördinaten van dat punt, zodat we de verzameling `Point3D` van alle punten in de ruimte als volgt kunnen voorstellen:

```
def Point3D :=  $\mathbb{R}^3$            def Vector :=  $\mathbb{R}^3$ 
```

Naast punten moeten we ook kunnen beschikken over vectoren in de ruimte, onder andere om de richting van een straal aan te geven. Net als in de bespreking van de wiskundige basis, zullen we werken met tupels van coördinaten ( $\mathbb{R}^3$ ) die de richting aangeven vanuit de oorsprong. Daarom identificeren we `Vector` met  $\mathbb{R}^3$ , hoewel het om verschillende verzamelingen gaat. Op die manier kunnen we verdere begrippen en functies elegant definiëren.

Een begrip dat centraal staat in het model is een straal. We voorzien twee functies die a.d.h.v. gegeven parameters een element van de verzameling `Ray` aanduiden:

```
def mkRay : Point3D → Point3D → Ray  
def mkRay' : Point3D → Vector → Ray
```

Een belangrijk begrip in een grafisch algoritme is kleur. Een kleur is een element van de verzameling `Color`, en wordt gedefinieerd als een tupel van drie reële getallen tussen 0 en 1:

```
def Color :=  $[0, 1]^3$ 
```

Naast stralen zijn ook de objecten uit de 3D-wereld een belangrijk begrip. Elk object heeft een bepaald type en bestaat uit een bepaald materiaal. De verschillende types objecten die ondersteund worden door het model, vormen elk hun eigen verzameling. Omdat we enkel bollen en vlakken zullen ondersteunen, hebben we dus twee verzamelingen: `Sphere` en `Plane`. Een willekeurig object is dan een element van de verzameling `Object`, die we als volgt kunnen definiëren:

```
def Object := Sphere ∪ Plane
```

Om een instantie van een bepaalde verzameling objecten te bekomen, beschikken we over

de functies `mkSphere` en `mkPlane`:

```
def mkSphere : Point3D → ℝ → Sphere
def mkPlane : ℝ4 → Plane
```

Samen met de verzameling `Texture` van alle materialen, kunnen we dan een object uit de 3D-wereld gaan voorstellen. Dit kunnen we doen m.b.v. een tupel dat het wiskundig object en het materiaal voor het object bevat:

```
def TexturedObject := Object × Texture
```

Als een snijding gevonden wordt van een straal met een object, moeten we verschillende zaken kunnen bepalen van die snijding. Daarom voorzien we een aparte verzameling `Intersection`. Uit elk element van die verzameling kunnen we m.b.v. enkele hulpfuncties de nodige informatie van de snijding bepalen. We kunnen een element van deze verzameling aanmaken met behulp van de functie `mkInt`:

```
def mkInt : ℝ → Ray → TexturedObject → Intersection
```

Deze functie bepaalt de snijding niet zelf, maar stelt wel de nodige parameters van de snijding in. Op die manier laat die toe dat de hulpfuncties voor snijdingen hun resultaat kunnen bepalen a.d.h.v. deze parameters. De waarde  $\varepsilon$  beschouwen we als een element van de verzameling `Intersection`.

Om de objecten in de 3D-wereld zichtbaar te maken, moeten we beschikken over belichting. Een lichtbron is een element van de verzameling `Light`, die we verder kunnen definiëren omdat we, net als bij objecten, twee types ondersteunen:

```
def Intensity := [0, 1]
def Light := AmbientLight ∪ PointLight
```

Hierbij zijn `AmbientLight` en `PointLight` eveneens verzamelingen, waarvan de elementen specifieke eigenschappen hebben. Een eigenschap die elke lichtbron gemeen heeft is een eigen lichtintensiteit, waarvoor we de verzameling `Intensity` voorzien.

Om te bepalen hoeveel we van de 3D-wereld te zien krijgen en op welke afstand we die bekijken, moeten we de plaats van het oogpunt en de grootte van het kijkvenster vastleggen (zodat we kunnen bepalen in welke mate de camera ‘ingezoomd’ is op de 3D-wereld). Dit kunnen we doen met een element van de verzameling `Camera`. Om een

element van deze verzameling aan te maken, voorzien we de functie `mkCamera`:

```
def mkCamera : Point3D →  $\mathbb{N}^2$  → Camera
```

Met de reeds besproken verzamelingen `Camera`, `Color`, `TexturedObject` en `Light` kunnen we een verzameling `Scene` van tupels definiëren, die toelaat om een 3D-wereld voor te stellen:

```
def Scene := Camera × Color × TexturedObject* × Light*
```

Elk tupel van deze verzameling bevat dan een beschrijving van de camera waardoor we de 3D-wereld bekijken, een achtergrondkleur, een lijst objecten en een lijst lichtbronnen.

Als laatste beschrijven we een verzameling voor het resultaat van het ray tracing algoritme: een afbeelding van een bepaalde resolutie. We beschouwen een afbeelding als een matrix van kleuren: met elke combinatie van rij en kolom stemt een kleur overeen. De verzameling `Image` bestaat dus uit de unie van alle afbeeldingen van een bepaalde resolutie, die we op zich kunnen voorstellen als een functie met twee argumenten (rij en kolom) en een kleur als resultaat.

```
def Image :=  $\bigcup (r : \mathbb{N}^2 . (\square (r\ 0) \times \square (r\ 1)) \rightarrow \text{Color})$ 
```

Hierbij merken we op dat de definitie vrij analoog is met die voor de verzameling  $A^*$  van tupels van willekeurige lengte, die gedefinieerd werd in paragraaf 2.1.2.

### 2.3.2 Functie definities

Naast het vastleggen van de begrippen bestaat het model voornamelijk uit functie-definities die de eigenlijke functionaliteit vastleggen. We verdelen deze functies onder in 3 groepen: hulpfuncties, wiskundige functies en functies die de kern van het ray tracing algoritme vormen. De eerste categorie zullen we kort bespreken, terwijl de twee laatste categorieën uitgebreider besproken worden, omdat we daarbij de functies formeel zullen definiëren, en er enkele eigenschappen van zullen aantonen.

Voor we de verschillende functies modelleren, bespreken we eerst de manier waarop we ze zullen definiëren. We hebben reeds vermeld dat de notatie `def binding` een identifier met een globaal bereik invoert. Daarnaast hebben we ook de notatie `with p`, die analoog is aan de optionele filter  $\wedge p$  in een binding. Beide notaties laten toe om het resultaat van een functie te specificeren. Het sleutelwoord `where` laat toe om identifiers met een lokaal bereik in te voeren. Op die manier kunnen we uitdrukkingen die te lang zijn of

meermaals voorkomen, vervangen door een lokale constante. Hierdoor wordt de functie-definitie duidelijker en eenvoudiger.

Bij wijze van voorbeeld definiëren we een eenvoudige functie  $f$  die de som maakt van de helft en het dubbele van een reëel getal  $a$  :

```
def f : ℝ → ℝ
with f a = half + double
where half =  $\frac{a}{2}$ 
         double = a · 2
```

## Hulpfuncties

In paragraaf 2.3.1 hebben we voor de verschillende begrippen verzamelingen ingevoerd, waarvoor we enkele eenvoudige functies voorzien. We bespreken enkel de types van deze functies, om geen restricties op te leggen aan de implementatie van de verzamelingen.

### Ray

Van een straal moeten we het startpunt en de richting van de straal kunnen bepalen. Dit kunnen we doen m.b.v. de functies `start` en `dir`, die respectievelijk een punt uit de ruimte en een vector als resultaat hebben:

```
def start : Ray → Point3D
def dir : Ray → Vector
```

### Sphere en Plane

De functies `rad` en `cen` leveren voor een object van het type `Sphere` respectievelijk de radius en het middelpunt van de bol op. Voor een vlak hebben we voldoende aan een functie `coords` die een tupel oplevert met de vier coördinaten die nodig zijn om de vergelijking van het vlak op te stellen. Voor de duidelijkheid vermelden we ook de types:

```
def rad : Sphere → ℝ
def cen : Sphere → Point3D
def coords : Plane → ℝ4
```

We merken op dat parameters van een element  $t$  van de verzameling `TexturedObject` toegankelijk zijn via tupeldenotatie:  $(t0)$  geeft een element van de verzameling `Object`, terwijl  $(t1)$  een element van de verzameling `Texture` aanduidt.

## Texture

Ieder materiaal bezit een reflectie- en transparantie-coëfficiënt, die bestaat uit een reëel getal tussen 0 en 1 en aangeeft in welke mate het object reflecteert of doorzichtig is. Twee andere parameters geven aan hoe groot de weerspiegeling van een lichtbron is op het object (*specularity*) en wat de brekingsindex is van het (transparante) object is. Voor elk van deze parameters voorzien we een aparte functie, die we hieronder vermelden:

```
def reflCoef : Texture → [0, 1]
def refrCoef : Texture → [0, 1]
def specCoef : Texture → ℕ
def refrIndex : Texture → ℝ
```

## Intersection

Bij een snijding moeten we kunnen beschikken over verschillende kenmerken. Om de kleur van het materiaal te bepalen in het snijpunt, hebben we de functie `colorAt`. De normaal in het snijpunt en de afstand van het startpunt van de straal tot het snijpunt kunnen we bepalen d.m.v. `normalAt` en `intDist`. Het snijpunt zelf en het materiaal van het object waarmee de snijding gemaakt wordt, kunnen we respectievelijk bepalen m.b.v. van de functies `intPt` en `intText`. De functie `isEmpty` bepaalt of er een snijding is of niet.

```
def colorAt : Intersection → Color
def normalAt : Intersection → Vector
def intDist : Intersection → ℝ
def intPt : Intersection → Point3D
def intText : Intersection → Texture
def isEmpty : Intersection → ℬ
```

## Light: AmbientLight en PointLight

Bij lichtbronnen onderscheiden we twee types: algemene lichtbronnen en punt-lichtbronnen. Met behulp van de functies `int` en `pos` kunnen respectievelijk we de intensiteit van een lichtbron en de positie van een punt-lichtbron bepalen.

```
def int : Light → Intensity
def pos : PointLight → Point3D
```

## Camera

Voor een element van de verzameling `Camera` voorzien we twee functies: één die het oogpunt van de camera bepaalt, en één die de dimensies van het kijkvenster (cfr. zoom van de camera) teruggeeft.

```
def eye : Camera → Point3D
def dim : Camera → ℕ2
```

Alle functies die in deze paragraaf besproken werden, laten toe om wijzigingen aan te brengen aan de specificatie van de verzamelingen, zonder dat dit gevolgen heeft voor andere functies. Het volstaat om, indien nodig, de beeldpunten van hulpfuncties samen met de specificatie te wijzigen (de types blijven wellicht behouden). Op die manier kunnen de functies die de kern van het algoritme vormen ongewijzigd blijven steunen op deze hulpfuncties.

## Wiskundige basis

De verschillende bewerkingen op vectoren zijn belangrijk bij het ray tracing algoritme, omdat vectoren erin een grote rol spelen. Het is belangrijk dat deze vectoren steeds genormaliseerd zijn, wat betekent dat het hun lengte steeds gelijk is aan 1. We voorzien enkele bewerkingen die dit toelaten.

Vooreerst definiëren we een functie `len` die de lengte van een vector bepaalt.

```
def len : Vector → ℝ
with len  $v = \sqrt{v \cdot v}$ 
```

Met deze functie kunnen we de functie `norm` definiëren, die een vector normaliseert.

```
def norm : Vector → Vector
with norm = (len  $v = 0$ ) ? (0, 0, 0) †  $v \cdot \frac{1}{\text{len } v}$ 
```

Omdat we vaak een nieuwe vector moeten aanmaken tussen twee punten in de ruimte, voorzien we een functie `mkNormVect` die meteen een genormaliseerde vector construeert tussen beide punten.

```
def mkNormVect : Point3D → Point3D → Vector
with mkNormVect  $v w = \text{norm } (w \hat{-} v)$ 
```

De afstand tussen twee punten in de ruimte kunnen we bepalen m.b.v. de functie `dist`:

```
def dist : Point3D → Point3D → ℝ
with dist p q = √((q ⋆ p) ⋅ (q ⋆ p))
```

Tijdens de berekeningen kunnen we door optelling en vermenigvuldiging van kleuren waarden bekomen die niet binnen de verzameling `Color` liggen. Daarom voorzien we een functie `clip`, die de elementen van een tupel van drie reële getallen aanpast, zodat het tupel beschouwd kan worden als een kleur.

```
def clip : ℝ³ → Color
with clip = 0 ∨̇ ∘ 1 ∧̇
```

Merk op dat we deze functie op een elegante manier puntsvrij hebben gedefinieerd. De operatoren `∨̇` en `∧̇` die gebruikt worden, komen respectievelijk overeen met de kleinste bovengrens en grootste ondergrens. We modelleren ook een functie `solveq`, die een vierkantsvergelijking oplost.

```
def solveq : ℝ³ → ℝ*²
with solveq p = d < 0 ? ε † (d > 0 ? ( (-p 1) - √d / (2 · (p 0)), (-p 1) + √d / (2 · (p 0)) ) † - p 1 / (2 · (p 0)) )
where d = (p 1) · (p 1) - 4 · (p 0) · (p 2)
```

Naast de basisbewerkingen zijn er ook een paar bewerkingen die specifiek zijn voor ray tracing en het werken met ruimtelichamen. Een eerste bewerking is het bepalen van de snijpunten van een (half)rechte en een ruimtelichaam. De functie `intRayWith` bepaalt de afstanden tussen de start van de straal en de snijpunten van de straal met een object, en geeft die terug in een *gesorteerd* lijstje. Op basis van het type van het object, bepalen we hoe we de snijpunten moeten berekenen.

```
def intRayWith : Ray → Object → ℝ*
with intRayWith r o = o ∈ Sphere ? solveq (dir r ⋅ dir r) (2 · (dir r ⋅ d)) (d ⋅ d - (rad r)²)
† - (c 3) + (c 0, c 1, c 2) ⋅ (start r) / ((c 0, c 1, c 2) ⋅ (dir r))
where d = (start r) ⋆ (cen o)
c = coords o
```

Bij effecten als reflectie en transparantie, moeten we in staat zijn om stralen te reflecteren en te breken. Dit gebeurt telkens aan de hand van de normaal in het snijpunt op het ruimtelichaam. Een normaal is een rechte die loodrecht op het oppervlak van een ruimtelichaam staat, die we kunnen bepalen m.b.v. de functie `normal`:

```

def normal : Point3D → Object → Vector
with normal p o = norm ( o ∈ Sphere ? ((cen o) ^ p) ^ 1 / rad o + (c 0, c 1, c 2) )
where c = coords o

```

Om een straal te reflecteren of te breken, moeten we aan de hand van een gegeven straal en de normaal in een punt een nieuwe straal kunnen construeren. Hiervoor voorzien we de functies `reflectDir` en `refractDir`, die respectievelijk, voor een gegeven invalrichting en normaal, de gereflecteerde en de gebroken richting bepalen.

```

def reflectDir : Vector → Vector → Vector
with reflectDir i n = i ^ ( n ^ ( 2 · ( i · n ) ) )
def refractDir : Vector → Vector → ℝ → Vector
with refractDir i n r = v < 0 ? (0, 0, 0) + norm ( i ^ r' ^ n ^ ( r' · ( abs c ) - sqrt v ) )
where c = n · ( i ^ - 1 )
          r' = ( c < 0 ? r + 1 / r )
          v = 1 + r'^2 · ( c^2 - 1 )

```

De functie `refractDir` heeft een extra parameter omdat er bij refractie rekening moet gehouden worden met de refractie-index. De functie `abs` bepaalt de absolute waarde van een getal, en kunnen we definiëren als `abs n = n ≥ 0 ? n + - n`.

De laatste functie die niet echt deel uitmaakt van de kern van het ray tracing algoritme is `mapToWin`. Deze functie zet, gegeven een resolutie en dimensies van een kijkvenster, de coördinaten van een pixel om in een punt op het kijkvenster. Hierbij is het bereik van de pixel-coördinaten beperkt tot de gegeven resolutie, m.b.v. een *dependent type*.

```

def mapToWin : ℕ² ∋ r → ℕ² → (□ r 0) × (□ r 1) → Point3D
with mapToWin r d p = ( t 0 / r 0, t 1 / r 1, 0.0 )
where t = ( p ^ ( r ^ 1 / 2 ) ) ^ d

```

## De kern van het algoritme

Na de bespreking van de begrippen, hulpfuncties en wiskundige functionaliteit waarop we kunnen steunen, bekijken we het belangrijkste onderdeel van het model. Ook hier gaan we stapsgewijs te werk: we definiëren eerst de functies die elk op zich een onderdeel van het algoritme behandelen, en presenteren dan de functies die de eerste combineren tot het eigenlijke algoritme.

Om het model flexibel te houden, hebben we het resultaat van de functie `intRayWith` gedefinieerd als een gesorteerd lijstje van reële getallen. Op die manier kunnen we later eenvoudig andere types van objecten ondersteunen die meer dan twee snijpunten kunnen hebben met een straal. Om de snijding van een object met een straal te bepalen, zijn we echter enkel geïnteresseerd in het dichtste snijpunt. Omdat de functie `intRayWith` zowel snijpunten voor als na het startpunt van de straal in rekening brengt, betekent dit dat we enkel de eerste positieve waarde van het lijstje nodig hebben. Daartoe definiëren we de recursieve functie `fstPos`, die voor een willekeurig lijstje van getallen het eerste positieve element bepaalt:

```
def fstPos : ℝ* → ℝ
with fstPos ε = 0
fstPos (x > xs) = x > 0 ? x † fstPos xs
```

Indien er geen enkel positief element aanwezig is in het gegeven lijstje, dan zal de functie 0 als beeldpunt hebben.

Om de dichtste snijding te kunnen bepalen van een straal met een verzameling van objecten, voorzien we de functie `closestInt`, die we zullen gebruiken als vergelijkingsfunctie bij het overlopen van de objecten. Deze functie bepaalt de dichtste van twee snijdingen: een gegeven snijding en een snijding tussen de gegeven straal en het gegeven object.

```
def closestInt : Ray → Intersection → TexturedObject → Intersection
with closestInt r i o = d > 0 ∧ ( isEmpty i ∨ d < (intDist i) ) ? mkInt d r o † i
where d = fstPos (intRayWith r (o 0))
```

We bepalen de dichtste positieve afstand van een snijding m.b.v. de functie `fstPos`. Indien er geen positief element in de lijst aanwezig is, dan is er ook geen gepaste snijding, en kunnen we dus de gegeven snijding teruggeven als resultaat. In het andere geval gaan we na of de gevonden snijding beter is dan de gegeven snijding, en leggen we de beste (dichtste) snijding vast als resultaat.

Met behulp van de functie `closestInt` kunnen we nu eenvoudig de functie `intersect` definiëren, die de dichtste snijding van een straal met een verzameling objecten bepaalt. We maken hierbij gebruik van de linker reductie functionaal  $\rightarrow$ , waarbij we een lege snijding  $\varepsilon$  als startwaarde voor de functie instellen. Functies die de functie `intersect` gebruiken, kunnen dan m.b.v. de hulpfunctie `isEmpty` bepalen of er effectief een snijding is tussen de straal en de verzameling objecten. Merk op dat de aanwezigheid van functies van hogere orde (en dus ook partiële functie-applicatie) ervoor zorgt dat we de functie `closestInt` al deels kunnen voorzien van de nodige parameters.

```
def intersect : Ray  $\rightarrow$  TexturedObject*  $\rightarrow$  Intersection
with intersect r o = (closestInt r)  $\rightarrow_{\varepsilon}$  o
```

Om deze functie te verduidelijken, zullen we in paragraaf 2.3.3 een eenvoudig voorbeeld uitwerken. We merken op dat we nergens in deze functie-definities een veronderstelling maken over het type van de objecten of de specificatie van een verzameling. Door middel van hulpfuncties voorzien we voldoende abstractie, zodat we slechts bij de implementatie van het model de specificatie van de verzamelingen moeten vastleggen.

De kleur van een bepaald punt van een object wordt bepaald door verschillende invloeden: de invloed van de belichting op het kleur van het object, de reflectie van de lichtbronnen/objecten en eventueel objecten die zichtbaar zijn door het punt in kwestie (bij transparantie). Voor elk van deze invloeden voorzien we een aparte functie, waarvan we de resultaten dan zullen samenstellen om de eigenlijke kleur te bekomen.

Een eerste component wordt gevormd door de diffuse kleur van een snijpunt, m.a.w. de kleur van het object onder invloed van de belichting. We definiëren de functie `diff` die een snijding en een punt-lichtbron als argumenten heeft. Om rekening te houden met de hoek waaronder het snijpunt belicht wordt door de lichtbron, bepalen we het scalair product van de richting naar de lichtbron met de normaal in het snijpunt. De waarde van dat scalair product wordt dan, m.b.v. de halve uitbreiding, vermenigvuldigd met de intensiteit van de lichtbron. Het resultaat van die bewerking wordt dan componentsgewijs vermenigvuldigd met het kleur van het object. Zo bekomen we de kleur van het snijpunt onder invloed van de gegeven punt-lichtbron.

```
def diff : Intersection  $\rightarrow$  PointLight  $\rightarrow$  Color
with diff i l = ( (int l)  $\vec{\cdot}$  ( (mkNormVect (intPt i) (pos l))  $\cdot$  normalAt i ) )  $\hat{\cdot}$  colorAt i
```

Naast de invloed van een lichtbron op de kleur van het object, wordt het licht zelf voor een deel gereflecteerd in het object, afhankelijk van het materiaal waaruit het object bestaat.

Deze component kunnen we bepalen d.m.v. de functie `spec`. Naast het snijpunt en de lichtbron, heeft deze functie ook de richting van de kijkstraal nodig, omdat de reflectie van een lichtbron in een object afhankelijk is van de hoek waaronder men het object bekijkt.

Om deze kijkhoek in rekening te brengen, berekenen we het scalair product van de normaal in het snijpunt met de bissectrice van de (omgekeerde) kijkrichting en de richting naar de lichtbron. Deze waarde wordt, d.m.v. machtsverheffing, aangepast volgens de glans van het object. Het product van de bekomen waarde met de reflectie-coëfficiënt wordt dan op zich vermenigvuldigd met de intensiteit van de lichtbron, waaruit we de kleur bekomen die wordt bepaald door de reflectie van de lichtbron in het snijpunt.

```
def spec : Intersection → Vector → PointLight → Color
with spec i d l = (int l)  $\vec{\cdot}$  ( ( reflCoef (intText i) ) · ( (normalAt i) · h )(specCoef (intText i)) )
  where h = norm ( (d  $\vec{\cdot}$  - 1)  $\hat{+}$  (mkNormVect (intPt i) (pos l)) )
```

We merken op dat we opnieuw gebruik maken van de halve uitbreiding van de vermenigvuldiging om de intensiteit van het licht te vermenigvuldigen met een factor, en dat we bij het bepalen van de bissectrice ervoor zorgen dat we een genormaliseerde vector bekomen.

Met behulp van de functies `diff` en `spec` kunnen we de functie `shadePt` definiëren die de kleur van een snijpunt bepaalt onder invloed van een lichtbron.

We controleren we of de lichtbron een algemene lichtbron (*ambient*) of een puntlichtbron is. In het eerste geval bestaat de bijdrage van de lichtbron enkel uit de intensiteit van de lichtbron zelf. Bij een puntlichtbron moeten we nagaan of het behandelde snijpunt in de schaduw ligt van een object. Hiervoor construeren we een straal vanuit het snijpunt in de richting van de lichtbron, en bepalen we de snijding met de verzameling objecten. Indien er een snijding wordt gevonden die tussen het snijpunt en de puntlichtbron ligt, dan bevindt het snijpunt zich in de schaduw van een object en wordt er geen bijdrage geleverd. In het andere geval wordt de bijdrage bepaald m.b.v. de functies `diff` en `spec`.

```
def shadePt : Intersection → Vector → TexturedObject* → Light → Color
with shadePt i d o l = clip (l ∈ AmbientLight ? int l  $\dagger$  ( s ? (0, 0, 0)
   $\dagger$  (diff i l)  $\hat{+}$  (spec i d l) ))
  where s =  $\neg$  (isEmpty i')  $\wedge$  intDist i'  $\leq$  dist (intPt i) (pos l)
        i' = intersect (mkRay (intPt i) (pos l)) o
```

Naast de rechtstreekse belichting van het object, zijn er ook indirecte invloeden die in rekening moeten worden gebracht. Zoals reeds vermeld is het vrij eenvoudig om effecten als reflectie en transparantie te bekomen met behulp van ray tracing. Het volstaat om vertrekkend vanuit een snijpunt een nieuwe straal te construeren in de gewenste richting, en recursief de kleurbijdrage van die straal te bepalen. Daardoor zijn de functies `reflectPt` en `refractPt`, die respectievelijk de reflectie van objecten in het snijpunt en de zichtbaarheid van objecten door het snijpunt behandelen, zeer eenvoudig te definiëren:

```
def reflectPt : Intersection → Vector → TexturedObject* → Light* → Color
with reflectPt i d = colorPt ( mkRay' (intPt i) (reflectDir d (normalAt i)) ) (0, 0, 0)
```

```
def refractPt : Intersection → Vector → Color → TexturedObject* → Light* → Color
with refractPt i d b = (d' = (0, 0, 0)) ? (0, 0, 0)
    † colorPt (mkRay' (intPt i) d') ( b · (refrCoef (intText i)) )
where d' = refractDir d (normalAt i) (refrIndex (intText i))
```

We definiëren de functies puntsvrij m.b.v. de functie `colorPt`. Daarbij construeren we de gereflecteerde en gebroken straal met de hulpfunctie `mkRay'`. De richtingen die we daarbij nodig hebben, bepalen we met de functies `reflectDir` en `refractDir` waarover we reeds beschikken. Hierbij moeten we wel de achtergrondkleur aanpassen, zodat de effecten die we bekomen ook kloppen met het doel van die achtergrondkleur. Bij het bepalen van de reflectie-component gaan we uit van een zwarte achtergrondkleur, zodat de gedefinieerde achtergrondkleur niet gereflecteerd wordt in de objecten. Bij refractie moeten we de gegeven achtergrondkleur aanpassen volgens de transparantie-coëfficiënt van het object, zodat die (afgezwakt) zichtbaar is door het object.

De uiteindelijke kleur van een snijpunt kunnen we bepalen met behulp van de functie `colorPt`, die per straal de kleur bepaalt a.d.h.v. de opgegeven parameters. We bepalen eerst het snijpunt van de gegeven straal met de verzameling objecten. Indien er geen snijding aanwezig is, zal de functie het achtergrondkleur als resultaat teruggeven. In het andere geval wordt het kleur bepaald door de som van de drie componenten, die berekend worden m.b.v. de functies `shadePt`, `reflectPt` en `refractPt`.

```

def colorPt : Ray → Color → TexturedObject* → Light* → Color
with colorPt r b o l = isEmpty i ? b † clip (  $\hat{\cdot}$   $\rightarrow_{(0,0,0)}$  ((shadePt i (dir r) o) o l)
                                 $\hat{\cdot}$  (reflectPt i (dir r) o l)  $\vec{\cdot}$  (reflCoef (intText i))
                                 $\hat{\cdot}$  (refractPt i (dir r) b o l)  $\vec{\cdot}$  (refrCoef (intText i))

where i = intersect r o

```

Om de kleur van een punt op het kijkvenster te bepalen voor een gegeven specificatie van een 3D-wereld, construeren we de straal vanuit het oogpunt door dat punt. Met behulp van de functie `colorPt` kunnen we dan uit die specificatie en de geconstrueerde straal de kleur bepalen van de pixel die overeenstemt met het punt op het kijkvenster.

```

def rayTracePt : Scene → Point3D → Color
with rayTracePt s p = colorPt (mkRay' p (mkNormVect (eye (s 0)) p)) (s 1) (s 2) (s 3)

```

De functie `rayTrace` vervult de rol van het eigenlijke algoritme: uit een gegeven resolutie en 3D-wereld bepalen we een afbeelding van de gewenste resolutie. Dit kan eenvoudig gedefinieerd worden als de samenstelling van de functie `mapToWin`, die pixelcoördinaten omzet naar een punt op het kijkvenster, met de functie `rayTracePt`, die voor een gegeven punt op het kijkvenster en een 3D-wereld een kleur teruggeeft.

```

def rayTrace :  $\mathbb{N}^2$  → Scene → Image
with rayTrace r sc = (rayTracePt sc) o (mapToWin r (dim (sc 0)))

```

### 2.3.3 Enkele eigenschappen

Zoals reeds eerder vermeld, kan een formeel model dienst doen als specificatie voor een uiteindelijke implementatie. Met deze specificatie kunnen we bepaalde eigenschappen van het model formeel aantonen. Op deze manier worden fouten geweerd uit de specificatie, zodat deze niet moeten opgespoord worden bij de implementatie ervan. We zorgen ervoor dat we telkens zo weinig mogelijk veronderstellingen maken over de parameters die geen deel uitmaken van de eigenschap die we aantonen. Op die manier werken we zo algemeen mogelijk, zodat er geen verborgen afhankelijkheden verantwoordelijk zijn voor het resultaat dat we bekomen.

## Beperken van het domein van rayTrace tot pixels binnen de resolutie

Als eerste eigenschap tonen we aan dat het domein van de rayTrace functie beperkt is tot de coördinaten van de pixels die binnen de opgegeven resolutie liggen. Dit is belangrijk omdat de functie voor andere pixelcoördinaten niet zinnig gedefinieerd is.

Om deze eigenschap aan te tonen, bepalen we het domein van de partiële applicatie van de rayTrace functie, namelijk voor een gegeven resolutie  $r$  en een specificatie  $s$  van een 3D-wereld. Als het domein van deze functie beperkt is volgens de vooropgestelde resolutie, dan is de eigenschap voldaan. We zullen m.a.w. het volgende aantonen:

$$\mathcal{D}(\text{rayTrace } r (c, b, \varepsilon, l)) = \square(r\ 0) \times \square(r\ 1) \quad (2.5)$$

Hierbij zullen we gebruik maken van extensionaliteit van verzamelingen, dat we kunnen uitdrukken d.m.v. een inferentie-regel:

$$\frac{(s \in S \equiv s \in T)}{S = T}$$

$$\begin{aligned} x \in \mathcal{D}(\text{rayTrace } r\ s) & \\ \equiv \langle \text{def rayTrace} \rangle \quad x \in \mathcal{D}((\text{rayTracePt } s) \circ (\text{mapToWin } r (\text{dim } s))) & \\ \equiv \langle \text{domein axioma } \circ \rangle \quad x \in \mathcal{D}(\text{mapToWin } r (\text{dim } s)) & \\ \quad \wedge (\text{mapToWin } r (\text{dim } s))\ x \in \mathcal{D}(\text{rayTracePt } s) & \\ \equiv \langle \text{type rayTracePt} \rangle \quad x \in \mathcal{D}(\text{mapToWin } r (\text{dim } s)) \wedge (\text{mapToWin } r (\text{dim } s))\ x \in \text{Point3D} & \\ \equiv \langle \text{type mapToWin} \rangle \quad x \in \mathcal{D}(\text{mapToWin } r (\text{dim } s)) & \\ \equiv \langle \text{type mapToWin} \rangle \quad x \in \square(r\ 0) \times \square(r\ 1) & \end{aligned}$$

Met behulp van de extensionaliteit van verzamelingen kunnen we besluiten dat aan de gelijkheid voldaan is, zodat de eigenschap bewezen is.

## Constante afbeelding voor lege verzameling objecten

Omdat we bij elke afbeelding een achtergrondkleur kiezen, zal een beschrijving van een 3D-wereld waarin geen enkel object aanwezig is, resulteren in een afbeelding die bestaat uit één kleur: de opgegeven achtergrondkleur. Dat betekent dat de functie die de afbeelding moet voorstellen, een constante functie zal zijn die ieder paar coördinaten van een pixel afbeeldt op een vaste kleur. Formeel kunnen we dit als volgt uitdrukken:

$$\text{rayTrace } r (c, b, \varepsilon, l) = (\square(r\ 0) \times \square(r\ 1)) \bullet b \quad (2.6)$$

Om deze gelijkheid aan te tonen, kunnen we gebruik maken van de inferentieregel voor functie-gelijkheid (zie 2.2). We moeten dus aantonen dat de domeinen van beide functies gelijk zijn en dat de beeldpunten van beide functies voor een willekeurig punt van het domein gelijk zijn.

*gelijkheid van de domeinen:*

$$\begin{aligned}
x \in \mathcal{D}(\text{rayTrace } r (c, b, \varepsilon, l)) & \\
\equiv \langle (2.5) \rangle & \quad x \in \square(r 0) \times \square(r 1) \\
\equiv \langle \mathcal{D}(X \bullet b) = X \rangle & \quad x \in \mathcal{D}(\square(r 0) \times \square(r 1) \bullet b)
\end{aligned}$$

M.b.v. de extensionaliteit van verzamelingen kunnen we besluiten dat de domeinen gelijk zijn.

*gelijkheid van de beeldpunten:*

Stel  $(x, y) \in \mathcal{D}(\text{rayTrace } r (c, b, \varepsilon, l))$ :

$$\begin{aligned}
& (\text{rayTrace } r (c, b, \varepsilon, l)) (x, y) \\
&= \langle \text{def rayTrace} \rangle \quad ( (\text{rayTracePt } (c, b, \varepsilon, l)) \circ (\text{mapToWin } r (\text{dim } c)) ) (x, y) \\
&= \langle \text{def } \circ \rangle \quad \text{rayTracePt } (c, b, \varepsilon, l) (\text{mapToWin } r (\text{dim } c) (x, y)) \\
&= \langle \text{def rayTracePt} \rangle \quad \text{colorPt } ray \ b \ \varepsilon \ l \\
&= \langle \text{def colorPt} \rangle \quad \text{isEmpty } (\text{intersect } ray \ \varepsilon) \ ? \ b \ \dagger \ color \\
&= \langle \text{def intersect} \rangle \quad \text{isEmpty } ((\text{closestInt } ray) \rightsquigarrow_{\varepsilon} \ \varepsilon) \ ? \ b \ \dagger \ color \\
&= \langle \star \rightsquigarrow_e \ \varepsilon = e \rangle \quad \text{isEmpty } \varepsilon \ ? \ b \ \dagger \ color \\
&= \langle \text{def isEmpty} \rangle \quad 1 \ ? \ b \ \dagger \ color \\
&= \langle 1 \ ? \ a \ \dagger \ b = a \rangle \quad b \\
&= \langle (X \bullet e) x = e \rangle \quad (\square(r 0) \times \square(r 1) \bullet b) (x, y)
\end{aligned}$$

Hierbij hebben we, om het bewijs te verduidelijken, de uitdrukkingen voor de straal en de kleur vervangen door *ray* en *color*. Met behulp van de inferentieregel kunnen we nu besluiten dat de gelijkheid (2.6) geldig is, zodat de eigenschap aangetoond is.

### Bepalen van dichtste snijding: voorbeeld

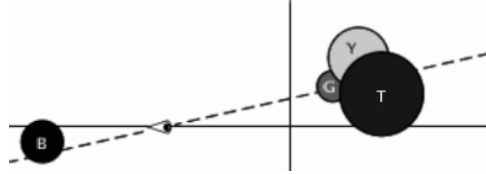
Om aan te tonen hoe de *intersect* functie werkt, bepalen we het beeldpunt voor een eenvoudig voorbeeld. We beschouwen een verzameling *o* van vier objecten:

$$\begin{aligned}
& (\text{mkSphere } (0, 20, 80) \ 40, \ t), \ (\text{mkSphere } (0, -10, -160) \ 20, \ b), \\
& (\text{mkSphere } (0, 30, 30) \ 15, \ g), \ (\text{mkSphere } (0, 60, 60) \ 30, \ y)
\end{aligned}$$

Omdat de eigenschappen van de verschillende materialen geen belang hebben, gaan we uit van enkele bestaande materialen:  $t$ ,  $b$ ,  $g$  en  $y$ . We bepalen ook de straal waarmee we de snijding moeten bepalen. We kiezen een straal  $r$  vanuit het oogpunt  $(0,0,-100)$  door het punt  $(0,20,0)$  van het kijkvenster:

**def**  $r := \text{mkRay } (0, 0, -100) (0, 20, 0)$

In figuur 2.4 stellen we de objecten en de straal grafisch voor.



Figuur 2.4: Grafische voorstelling van de objecten en de straal

We vermelden ook het (afgeronde) resultaat van de functie `intRayWith` voor de straal  $r$  en de verschillende objecten uit het lijstje.

`intRayWith r ((o 0) 0) = (143.717, 217.350)`    `intRayWith r ((o 1) 0) = (-80.747, -40.916)`  
`intRayWith r ((o 2) 0) = (118.951, 147.924)`    `intRayWith r ((o 3) 0) = (156.663, 180.856)`

We bestuderen het gedrag van de functie `intersect` door stap voor stap het resultaat uit te werken. Hierbij maken we gebruik van de shift-operator  $\sigma$ , zoals we die gedefinieerd hebben voor tupels (zie 2.1.2). We merken op dat we de functie `closestInt` niet stap voor stap uitwerken, omdat dit ons te ver zou leiden. We zetten telkens een applicatie van de functie `closestInt` in één stap om in het resultaat van die applicatie.

`intersect r o`  
 $= \langle \text{def intersect} \rangle \quad (\text{closestInt } r) \rightarrow_{\varepsilon} o$   
 $= \langle \text{def } \rightarrow \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{closestInt } r \varepsilon (o 0))} \sigma o$   
 $= \langle \text{def closestInt} \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{mkInt } r (o 0))} \sigma o$   
 $= \langle \text{def } \rightarrow \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{closestInt } r (\text{mkInt } r (o 0)) (o 1))} \sigma^2 o$   
 $= \langle \text{def closestInt} \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{mkInt } r (o 0))} \sigma^2 o$   
 $= \langle \text{def } \rightarrow \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{closestInt } r (\text{mkInt } r (o 0)) (o 2))} \sigma^3 o$   
 $= \langle \text{def closestInt} \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{mkInt } r (o 2))} \sigma^3 o$   
 $= \langle \text{def } \rightarrow \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{closestInt } r (\text{mkInt } r (o 2)) (o 3))} \varepsilon$   
 $= \langle \text{def closestInt} \rangle \quad (\text{closestInt } r) \rightarrow_{(\text{mkInt } r (o 2))} \varepsilon$   
 $= \langle \text{def } \rightarrow \rangle \quad \text{mkInt } r (o 2)$

Hieruit blijkt dat het object met materiaal  $g$  de dichtste snijding vormt met de gegeven straal, zodat het resultaat van de uitwerking klopt. We wijzen er op dat dit *geen* formeel bewijs is, maar enkel een uitwerking die aantoont dat het resultaat klopt voor het gegeven voorbeeld.

### Oneindige recursie

Als laatste punt gaan we na hoe het model omgaat met oneindige reflectie. Daartoe beschouwen we twee spiegelende vlakken evenwijdig met het kijkvenster. Wanneer het algoritme de kleur voor de straal vanuit het oogpunt door het middelpunt  $m$  van het kijkvenster zal bepalen, zal het te maken krijgen met oneindige reflectie. We werken het resultaat van de `rayTrace` functie enkele stappen uit, en gaan na hoe we dit kunnen opvangen in de implementatie.

We definiëren eerst de verzameling  $o$  van objecten, d.w.z. de twee spiegelende vlakken die we beschouwen, en de camera  $c$ :

```
def o := ( (mkPlane (0,0,-1,100)), t ), ( (mkPlane (0,0,1,100)), s )
def c := mkCamera (0,0,-50) w
```

We beschikken we hierbij over twee materialen ( $t$  en  $s$ ), een kijkvensterdimensie  $w$ , een verzameling lichtbronnen  $l$ , een achtergrondkleur  $b$  en een resolutie  $r$ . De enige vereiste is dat de verzameling van lichtbronnen niet leeg is. We vermelden opnieuw eerst enkele resultaten die we nodig zullen hebben in het algoritme: de richtingen van de stralen die voorkomen, de gereflecteerde richtingen en de snijpunten van de stralen met de vlakken:

```
dir ( mkRay (0,0,-50) (0,0,0) ) = (0,0,1) = dir ( mkRay (intPt ib) (0,0,1) )
dir ( mkRay (intPt ir) (0,0,-1) ) = (0,0,-1) = reflectDir (0,0,1) (normalAt ir)
reflectDir (0,0,-1) (normalAt ib) = (0,0,1)
```

Bij de uitwerking vermelden we de kleuren die we bekomen niet expliciet, maar vervangen we die telkens door  $clr$ .

```
rayTrace r (c, b, o, l) m = < def rayTrace > ( rayTracePt (c, b, o, l) o mapToWin r w ) m
= < def o > rayTracePt (c, b, o, l) ( mapToWin r w m )
= < def mapToWin > rayTracePt (c, b, o, l) (0,0,0)
= < def rayTracePt > colorPt ( mkRay (0,0,-50), (0,0,0) ) b o l
= < def colorPt, (1) > (*) clip ( clr  $\hat{+}$  (reflectPt ir (0,0,1) o l) )
= < def reflectPt > clip ( clr  $\hat{+}$  (colorPt (mkRay (intPt ir) (0,0,-1)) (0,0,0) o l) )
```

$$\begin{aligned}
&= \langle \mathbf{def\ colorPt}, (1) \rangle \quad \text{clip} (clr \hat{+} \text{clip} (clr' \hat{+} (\text{reflectPt } ib (0, 0, -1) o l))) \\
&= \langle \mathbf{def\ reflectPt} \rangle \quad \text{clip} (clr \hat{+} \text{clip} (clr' \hat{+} \\
&\quad \quad \quad \text{colorPt} ( \text{mkRay} (\text{intPt } ib) (0, 0, 1)) (0, 0, 0) o l ))) \\
&= \langle \mathbf{def\ colorPt}, (1) \rangle \quad \text{clip} (clr \hat{+} \text{clip} (clr' \hat{+} \text{clip} (clr'' \hat{+} (\text{reflectPt } ir (0, 0, 1) o l) )))
\end{aligned}$$

(1): omdat de vlakken enkel reflectief en niet transparant zijn, laten we de transparante term steeds vallen; hierbij maken we geen foute veronderstellingen, want de bijdrage van deze term zal steeds (0.0,0.0,0.0) zijn (en bovendien doet de kleur er niet echt toe, het gaat om de oneindige reflectie die optreedt)

We merken dat we in de laatste stap opnieuw dezelfde term bekomen als bij (\*). Bovendien is ook de dichtste snijding *ir* dezelfde, want de 3D-wereld bevat slechts twee objecten. Indien we met deze situatie geen rekening zouden houden bij de implementatie van het model, dan zou er een oneindige recursie optreden bij dit specifieke voorbeeld. Om dit te vermijden, kunnen we een parameter invoeren die de maximale recursiediepte bepaalt. Indien deze recursie-diepte bereikt wordt, dan bepalen we geen verdere termen meer, maar geven we (0,0,0) als resultaat van de functie **colorPt** terug.

# Hoofdstuk 3

## Implementatie in Haskell

Op basis van het model in Funmath, implementeren we een eenvoudige ray tracer in Haskell. We houden daarbij rekening met de tekorten en de voorzieningen van de functionele programmeertaal, in vergelijking met de concepten van het Funmath formalisme.

De verschillende verzamelingen die we besproken hebben in paragraaf 2.3.1, zullen we nu concreet gaan implementeren als (data)types. Daarom zijn de parameters van de types rechtstreeks toegankelijk, waardoor er geen nood meer is aan de hulpfuncties die gedefinieerd werden in het model.

De verschillende functies die we formeel gedefinieerd hebben in paragraaf 2.3.2, zullen we implementeren in twee aparte modules: de module `HRayMath` met de wiskundige functies en de module `HRayEngine` met de functies die de kern van het ray tracing algoritme vormen.

Omdat we materialen zullen ondersteunen waarvan de kleur varieert, zullen we enkele functies implementeren die verschillende effecten toelaten. Deze functies zullen we, samen met het algoritme dat erachter schuilt, implementeren in de module `HRayPerlin`.

Om de uitvoer van het ray tracing algoritme te kunnen visualiseren, voorzien we enkele functies die ons hierbij kunnen helpen. Die functies bevinden zich in de module `HRayOutput`, en zullen gebruikt worden door de textuele en grafische gebruikersinterface die we bespreken in hoofdstuk 4.

### 3.1 Algemeen

We bespreken eerst de syntax en de verschillende taalconstructies van Haskell, en bekijken ook de verschillen met het Funmath formalisme.

### 3.1.1 Haskell syntax

Op enkele notaties na, is de syntax voor functies in Haskell vrij gelijkaardig met de syntax die we hanteren in Funmath. Om dit te illustreren, geven we de syntax weer van de voorbeeldfunctie uit paragraaf 2.3.2.

```
f :: Double -> Double
f a = half + double
  where
    half   = a / 2
    double = a * 2
```

Net als bij Funmath, bestaat een functie-declaratie uit een typedeclaratie en een eigenlijke functie-definitie. De typedeclaratie is, in tegenstelling tot in Funmath, geen verplichting in Haskell. De Hugs-interpreter en GHC-compiler kunnen het type van een functie afleiden uit de declaratie ervan. Om type-fouten te vermijden zullen we echter steeds expliciet het type van een functie weergeven. Bovendien wordt de code hierdoor duidelijker, en worden er minder makkelijk fouten gemaakt. De sleutelwoorden **def** en **with** worden in Haskell niet meer gebruikt. Wel moet de functie-definitie op een nieuwe lijn beginnen, en moet het sleutelwoord **where** wat inspringen t.o.v. de rest van de functie-definitie. De declaraties van de lokale constanten die ingevoerd worden m.b.v. **where**, moeten allemaal gelijk gealigneerd zijn.

Om de code te documenteren bestaan er verschillende mechanismes. Een eerste is de klassieke notatie, m.b.v. `--` of `{- -}`, respectievelijk voor een lijn en meerdere lijnen commentaar. Deze stijl wordt verondersteld te worden gebruikt indien de code opgeslaan wordt in een bestand met extensie `hs`. Een tweede stijl is Literate Haskell, waarbij elke lijn code aangeduid wordt m.b.v. een `>`-teken aan het begin van de lijn en de code bewaard wordt in een bestand met `lhs` als extensie. Alle lijnen die niet beginnen met het `>`-teken worden beschouwd als commentaar. Wanneer deze stijl gehanteerd wordt, is het makkelijker om documentatie en code van elkaar te scheiden, en bovendien laat het toe om bvb. met behulp van LaTeX rechtstreeks vanuit de code documentatie te genereren.

Naast functie-declaraties zijn er natuurlijk ook type-declaraties, die toelaten om eigen (data)types te definiëren. Hiervoor bestaan de sleutelwoorden **type** en **data**, waarbij **type** toelaat om een alias toe te kennen aan een bestaand type, terwijl **data** gebruikt worden om een volledig nieuw datatype aan te maken. Samen met dat nieuwe type worden ook constructors gedefinieerd, i.e. functies die kunnen gebruikt worden om een instantie van

het type aan te maken. Deze naam van deze constructoren moeten altijd beginnen met een hoofdletter, om het onderscheid te maken tussen gewone functies en constructoren voor datatypes. Dit onderscheid is o.a. noodzakelijk omdat enkel constructoren gebruikt kunnen worden bij patroonherkenning.

Het opdelen van de code in verschillende modules laat toe om de verschillende onderdelen van de applicatie van elkaar te scheiden. Zo staat de `HRayMath` module volledig op zichzelf, en kan die ook gebruikt worden voor een andere applicatie. De functionaliteit aangeboden door deze module wordt in de `HRayEngine` module gebruikt, na het toevoegen van een `import`-declaratie. Hierbij kunnen we de volledige module importeren, of enkel specifieke functies uit die module (om bvb. naam-conflicten te vermijden of om het resulterende programma compacter te maken). Analoog kan een module alle functies die er gedefinieerd werden ter beschikking stellen, of enkel een deel ervan.

Zoals reeds aangehaald, bestaan er gelijkenissen en essentiële verschillen tussen Funmath en Haskell. De gelijkenissen omvatten o.a. de gelijklopende syntax, de ondersteuning van functies van hogere orde (en dus ook het toelaten van partiële functie-applicatie) en het links-associatief zijn van de functie-applicatie (zodat  $f \ a \ b = (f \ a) \ b$ ).

De verschillen met het Funmath formalisme dwingen ons om bepaalde definities anders te gaan noteren. Een belangrijk verschil is dat tupels in Haskell geen functies zijn zoals in Funmath. Dit kunnen we eenvoudig opvangen door gebruik te maken van patroonherkenning, dat we illustreren dit met een klein voorbeeldje. We definiëren een functie in Funmath en in Haskell, die twee tupels componentsgewijs optelt.

<pre>def f : ℝ<sup>2</sup> → ℝ<sup>2</sup> → ℝ<sup>2</sup> f v w = ( (v 0) + (w 0), (v 1) + (w 1) )</pre>	<pre>type TupleD = (Double,Double) f :: TupleD -&gt; TupleD -&gt; TupleD f (a,b) (c,d) = (a+c,b+d)</pre>
---	--

Bovendien bestaat er nu een expliciet onderscheid tussen tupels of sequenties (van de vorm  $(.,.,.)$ ) en lijsten (van de vorm  $[.,.,.]$ ). Dit onderscheid resulteert in een verschillende behandeling van tupels en lijsten: zo is de `foldl` operator enkel gedefinieerd voor lijsten, en niet voor tupels, in tegenstelling tot de  $\rightarrow$  functionaal in Funmath.

Naast verschillen tussen Funmath en Haskell zelf, moeten we er rekening mee houden dat berekeningen met reële getallen niet exact kunnen gebeuren in een computer. Hierdoor moeten we rekening houden met afrondingsfouten wanneer we de waarden van een functie gaan controleren, om zo een onderscheid te maken tussen verschillende gevallen.

## 3.2 De module HRayMath

De module `HRayMath` bevat alle wiskundige basisfuncties die besproken werden in paragraaf 2.3.2, samen met de nodige (data)types om die functies te ondersteunen. We beginnen de bespreking van deze module bij die (data)types.

```
type Point2D      = (Int,Int)
type Point3D      = (Double,Double,Double)
type Vector       = (Double,Double,Double)
type Resolution   = (Int,Int)
type Dimension    = (Int,Int)
data Object       = Sphere Double Point3D
                  | Plane (Double,Double,Double,Double)
data Ray          = Ray Point3D Vector
```

De types `Point3D` en `Vector` definiëren we als tupels van drie reële getallen, analoog met de `Funmath` definitie. Omdat we de functies onderverdelen in verschillende modules, en we de typering van de verschillende functies niet van elkaar willen laten afhangen, definiëren we drie extra types: `Point2D`, `Resolution` en `Dimension`. Deze types zullen we gebruiken in de functie `mapToWin`, net als in enkele functies (en types) in de `HRayEngine` module.

De datatypes `Object` en `Ray` zijn de specificaties van de `Object` en `Ray` verzamelingen uit het model. We merken hierbij op dat we geen datatypes voorzien voor de verzamelingen `Sphere` en `Plane`. Deze verzamelingen maken deel uit van de verzameling `Object`, en daarom wordt het datatype `Object` gedefinieerd d.m.v. de keuze-operator `|`. Als resultaat is elke instantie die gecreëerd wordt m.b.v. de constructors `Sphere` en `Plane` een instantie van het datatype `Object`. De twee constructoren `Sphere` en `Plane` laten een concrete implementatie toe van de verzamelingen `Sphere` en `Plane`. M.b.v. patroonherkenning zijn de parameters van de instanties toegankelijk.

Analoog levert de declaratie van het datatype `Ray` een constructor op die een concrete versie is van de functie `mkRay'` uit het model. De abstracte functie `mkRay` kunnen we implementeren m.b.v. deze constructor.

```
mkRay :: Point3D -> Point3D -> Ray
mkRay p1 p2 = Ray p1 (mkNormVect p1 p2)
```

De functie `mkNormVect` waarvan we gebruik maken wordt later gedefinieerd.

De verschillende functies die deel uitmaken van de wiskundige basis werden in het model dikwijls geïmplementeerd m.b.v. de directe of de halve uitbreiding van wiskundige

basisbewerkingen. Opdat we ook over deze uitbreidingen zouden beschikken, implementeren we handmatig de uitbreidingen van de verschillende operatoren die we nodig hebben. Merk op dat we dit puntsgewijs (m.b.v. patroonherkenning) moeten doen, omdat we tupels niet per element kunnen overlopen zoals we dit kunnen bij lijsten. We definiëren de directe uitbreiding van de optelling ( $\langle + \rangle$ ), het verschil ( $\langle - \rangle$ ) en de vermenigvuldiging ( $\langle * \rangle$ ), alsook de halve uitbreiding van de vermenigvuldiging ( $\langle * \rangle$ ).

```

( $\langle + \rangle$ ) :: (Double,Double,Double) -> (Double,Double,Double)
          -> (Double,Double,Double)

```

```

(x1,y1,z1)  $\langle + \rangle$  (x2,y2,z2) = (x1+x2, y1+y2, z1+z2)

```

```

( $\langle - \rangle$ ) :: (Double,Double,Double) -> (Double,Double,Double)
          -> (Double,Double,Double)

```

```

(x1,y1,z1)  $\langle - \rangle$  (x2,y2,z2) = (x1-x2,y1-y2,z1-z2)

```

```

( $\langle * \rangle$ ) :: (Double,Double,Double) -> (Double,Double,Double)
          -> (Double,Double,Double)

```

```

(x1,y1,z1)  $\langle * \rangle$  (x2,y2,z2) = (x1*x2,y1*y2,z1*z2)

```

```

( $\langle * \rangle$ ) :: (Double,Double,Double) -> Double -> (Double,Double,Double)

```

```

(x,y,z)  $\langle * \rangle$  f = (x*f,y*f,z*f)

```

We definiëren deze operatoren zodanig dat ze standaard als infix gebruikt moeten worden. Dit kunnen we doen door bij de type-declaratie de operator te omringen met haakjes, en bij de functie-definitie de operator als infix te noteren.

Naast de uitbreidingen van de klassieke aritmetische bewerkingen, implementeren we ook de halve uitbreiding van het maximum ( $\text{maxF}$ ) en het minimum ( $\text{minF}$ ), als implementatie van  $\vec{\vee}$  en  $\vec{\wedge}$  die gebruikt worden bij het definiëren van de clip-functie.

```

maxF :: Double -> (Double,Double,Double) -> (Double,Double,Double)

```

```

maxF f (x,y,z) = (max x f, max y f, max z f)

```

```

minF :: Double -> (Double,Double,Double) -> (Double,Double,Double)

```

```

minF f (x,y,z) = (min x f, min y f, min z f)

```

Het scalair product van twee vectoren definiëren we in het model m.b.v. de  $\rightarrow$  functionaal. In Haskell kunnen we echter geen gebruik maken van de elastische uitbreiding van de som, maar moeten we de termen gewoon optellen. We definiëren het scalair product ( $\langle * \rangle$ ) puntsgewijs als volgt:

```
(*.) :: Vector -> Vector -> Double
(x1,y1,z1) *. (x2,y2,z2) = x1*x2 + y1*y2 + z1*z2
```

Bij de implementatie van de overige wiskundige functies zullen we, waar mogelijk, net als in het model gebruik maken van deze uitbreidingen. We merken op dat zonder de tussentap van het formele model, we deze operatoren waarschijnlijk niet zouden definiëren, en telkens de correspondere uitdrukking zouden uitschrijven in elke functie-definitie.

De drie bewerkingen `len`, `norm` en `mkNormVect` die toelaten om steeds met genormaliseerde vectoren te werken, kunnen we als volgt implementeren:

```
len :: Vector -> Double
len v = sqrt (v *. v)

norm :: Vector -> Vector
norm v
  | len v < 10**(-9) = (0.0,0.0,0.0)
  | otherwise = v *> (1/(len v))

mkNormVect :: Point3D -> Point3D -> Vector
mkNormVect v w = norm (w <-> v)
```

De implementatie van deze functies verloopt compleet analoog met de definities in het model, behalve voor de functie `norm`. Omdat we hierbij de gegeven vector vermenigvuldigen met een quotiënt, moeten we ervoor zorgen dat we nooit delen door nul. De enige vector die als lengte nul heeft is de nulvector zelf, en dus geven we de nulvector terug als resultaat van de functie, indien de lengte voldoende dicht bij 0 ligt. Op deze manier houden we rekening met de afrondingsfouten die mogelijks optreden bij het werken met reële getallen. Merk dat we kunnen beschikken of een functie `sqrt` die de vierkantswortel bepaalt van een reëel getal, en over een machtsfunctie die we noteren als `**` (of alternatief als `^`).

De implementatie van de functies `dist` en `clip` verloopt analoog met het model:

```
dist :: Point3D -> Point3D -> Double
dist p0 p1 = sqrt ((p1 <-> p0) *. (p1 <-> p0))

clip :: (Double,Double,Double) -> (Double,Double,Double)
clip = (maxF 0.0) . (minF 1.0)
```

We merken op dat we bij de functie-definitie van `clip` net als in het Funmath formalisme gebruik kunnen maken van de functie-compositie  $f \circ g$ , die we in Haskell noteren als `f . g`.

De functie `solveq` die de oplossing(en) bepaalt voor een vierkantsvergelijking, kunnen we als volgt implementeren:

```

solveq :: (Double,Double,Double) ->[Double]
solveq (a,b,c)
  | (d < 0) = []
  | (d > 0) = [(- b - sqrt d)/(2*a), (- b + sqrt d)/(2*a)]
  | otherwise = [-b/(2*a)]
  where
    d = b*b - 4*a*c

```

Hierbij maken we gebruik van wachters (*guards*), die de voorwaarden in de if-then-else constructies uit het model vervangen. Deze wachters worden in volgorde van voorkomen geëvalueerd, en wanneer een evaluatie `True` als resultaat heeft zal de corresponderende expressie het resultaat van de functie bepalen. De laatste guard `otherwise` zal altijd tot `True` evalueren, en doet dienst als vangnet.

Tot nu toe hebben de functies maar in beperkte mate gebruikt gemaakt van de patroonherkenning van Haskell. De implementatie van de functie `intRayWith` maakt hier wel dankbaar gebruik van:

```

intRayWith :: Ray -> Object -> [Double]
intRayWith (Ray start dir) (Sphere rad cen) =
    solveq (dir *. dir, 2*(dir *. d), (d *. d) - rad^2)
  where
    d = start <-> cen
intRayWith (Ray start dir) (Plane (a,b,c,d)) =
    if (abs(part) < 10**(-9))
      then []
      else [- (d + ((a,b,c) *. start) ) / part]
  where
    part = (a,b,c) *. dir

```

De functie-definitie wordt in twee delen opgesplitst: een deel voor de elementen van de `Sphere` verzameling, en een deel voor de elementen van de `Plane` verzameling. Op basis

van patroonherkenning wordt beslist welke functie-definitie geëvalueerd moet worden voor een gegeven instantie van het `Object` datatype. Hierbij worden de constructoren `Sphere` en `Plane` gebruikt, samen met de `Ray` constructor, die enerzijds het bepalen van de juiste functie-definitie toelaten, en anderzijds toegang bieden tot de parameters van de instanties.

Ook de implementatie van de `normal` functie maakt handig gebruik van patroonherkenning:

```
normal :: Point3D -> Object -> Vector
normal p (Sphere rad cen) = norm ((p <-> cen) *> (1/rad))
normal _ (Plane (a,b,c,d)) = norm (a,b,c)
```

We merken op dat bij de functie-definitie voor elementen van de verzameling `Plane` geen enkele restrictie wordt opgelegd aan het eerste argument van de functie (er wordt zelfs geen naam gegeven aan het argument). Dit gebeurt m.b.v. het underscore-teken, dat duidelijk opvalt tussen de andere argumenten. Het gebruik van deze notatie vormt opnieuw (net als de expliciete type-declaratie) een soort documentatie van de code in de code zelf.

De functies `reflectDir` en `refractDir` hebben we als volgt geïmplementeerd in Haskell:

```
reflectDir :: Vector -> Vector -> Vector
reflectDir i n = i <-> (n *> (2*(n *. i)))

refractDir :: Vector -> Vector -> Double -> Vector
refractDir i n r = if (v < 0) then (0.0, 0.0, 0.0)
                  else norm $ (i *> r_c)
                              <+> (n *> (r_c*(abs c) - sqrt v))
  where
    c = n *. (i *> (-1))
    r_c = if (c < 0) then r else 1/r
    v = 1 + (r_c^2) * (c^2 - 1)
```

Hierbij moeten we rekening houden met de restricties van de vierkantswortel-bewerking, die enkel gedefinieerd is voor positieve waarden. Indien de lokale variabele `v` negatief is, dan vormt de nulvector het resultaat van de `refractDir` functie. We merken ook het gebruik van het `$`-teken op, dat toelaat om het aantal haakjes te beperken.

Als laatste functie bespreken we de implementatie van `mapToWin`:

```
mapToWin :: Resolution -> Dimension -> Point2D -> Point3D
mapToWin (rx,ry) (w,h) (px,py) = (x/rxD,y/ryD,0.0)
  where
    (rxD,ryD) = (fromIntegral rx, fromIntegral ry)
    (pxD,pyD) = (fromIntegral px, fromIntegral py)
    (wD,hD) = (fromIntegral w, fromIntegral h)
    (x,y) = ( (pxD-rxD/2)*wD, (pyD-ryD/2)*hD )
```

Hierbij valt het op dat we de tupels die de argumenten van de functie vormen moeten herwerken. Dit komt door het typesysteem van Haskell, dat hier iets té restrictief is. De argumenten zijn telkens tupels van natuurlijke getallen (`Int`). Omdat het resultaat echter een tupel van reële getallen (`Double`) is, wordt er vereist dat het resultaat van de bewerkingen ook telkens een reëel getal is. In het formele model is dit geen probleem, omdat  $\mathbb{N} \subset \mathbb{R}$ . Een instantie van het type `Int` is echter geen instantie van het type `Double`, waardoor een expliciete conversie tussen beide types nodig is. Deze conversie kunnen we doorvoeren m.b.v. `fromIntegral`.

### 3.3 De module `HRayEngine`

Steunend op de functies uit de module `HRayMath`, bespreken we de implementatie van de functies die de kern van het ray tracing algoritme vormen. Om aan te geven dat we zullen gebruiken maken van de module `HRayMath`, importeren we deze module in de module `HRayEngine`. Bovendien hebben we ook enkele functies uit de Haskell module `Maybe` nodig, zodat we ook die moeten importeren.

```
import Maybe
import HRayMath
```

We merken op dat we niet alle functies die we definiëren beschikbaar zullen stellen voor andere modules, in tegenstelling in `HRayMath`. Aan de hand van volgende notatie geven we aan dat enkel de vermelde types, constructoren en functies beschikbaar zullen zijn:

```
module HRayEngine (Resolution, Color, Diff(Solid,Perlin),
                  Texture(Texture), TexturedObject, Light(AmbientLight,
                  PointLight), Scene(Scene), Camera(Camera), rayTrace)
```

We stellen dus enkel de functie `rayTrace` ter beschikbaar, samen met enkele datatypes en constructoren die we nodig zullen hebben om vanuit een tekstbestand een interne beschrijving van de 3D-wereld aan te maken (zie hoofdstuk 4).

Opnieuw bekijken we eerst de (data)types die in deze module gedefinieerd worden.

```
type Color = (Double,Double,Double)
data Diff = Solid Color | Perlin (Point3D -> Color)
data Texture = Texture Diff Double Int Double Double
type TexturedObject = (Object,Texture)
type Intensity = (Double,Double,Double)
data Light = PointLight Point3D Intensity | AmbientLight Intensity
data Camera = Camera Point3D Dimension
data Scene = Scene Camera Color [TexturedObject] [Light]
data Intersection = Intersection Double Ray TexturedObject
type Image = Point2D -> Color
```

De types `Color` en `Intensity` worden gedefinieerd als tupels van drie reële getallen. Het is echter niet mogelijk om, d.m.v. een type-definitie, de waarden van het type `Color` te beperken tot waarden die in het interval  $[0, 1]$  liggen. De datatypes `Diff` en `Light` worden op analoge manier geïmplementeerd als bij het datatype `Object`. We merken op dat we twee soorten kleuren onderscheiden: vaste kleuren en kleuren die afhankelijk zijn van een punt in de ruimte (op basis van Perlin noise [19, 20, 21]). Deze laatste laat effecten als marmer, hout en vuur toe, en maakt gebruik van functies die we in een aparte module `HRayPerlin` definiëren.

Net als bij de vorige datatypes, vormen de definities van de datatypes `Texture`, `Camera` en `Scene` concrete implementaties van de verzamelingen uit het model, zodat de parameters rechtstreeks toegankelijk zijn d.m.v. patroonherkenning. Het type `TexturedObject` bestaat, net als in het model, uit een tupel dat een object bevat samen met een materiaal voor dat object, terwijl het type `Image` eveneens overeenkomt met de verzameling `Image` van alle mogelijke afbeeldingen. Ook hier kunnen we geen restrictie opleggen op de argumenten van de functie die een afbeelding vormt.

De implementatie van de `Intersection` verzameling gebeurt m.b.v. de `Maybe` monade, omdat in het model ook  $\varepsilon$  als een element van de verzameling wordt beschouwd. We definiëren het datatype `Intersection` analoog met de andere datatypes, maar zullen het bijna altijd gebruiken in combinatie met de `Maybe` monade. Deze laat toe dat zowel de waarde `Nothing` als de waarde `(Just i)` een instantie is van het datatype `(Maybe Intersection)`, waarbij `i` op zich een instantie is van het datatype `Intersection`.

Omdat we bij het datatype `Intersection` steeds zullen werken met de `Maybe` monade, moeten we hierbij wel de abstracte hulpfuncties uit het model implementeren. De functie `isEmpty` uit het model hoeven we niet zelf te gaan implementeren, omdat die reeds bestaat, zij onder een andere naam: de functie `isNothing` bepaalt of een instantie van het type `(Maybe a)` leeg is of niet. De overige functies kunnen we als volgt implementeren:

```
intDist :: (Maybe Intersection) -> Double
intDist Nothing = 0.0
intDist (Just (Intersection d _ _)) = d

intText :: (Maybe Intersection) -> Texture
intText Nothing = Texture (Solid (0.0,0.0,0.0)) 0.0 0 0.0 0.0
intText (Just (Intersection _ _ (_,t))) = t

colorAt :: (Maybe Intersection) -> Color
colorAt Nothing = (0.0,0.0,0.0)
colorAt (Just (Intersection _ _ (_,Texture (Solid c) _ _ _ _))) = c
colorAt i@(Just (Intersection _ _ (_,Texture (Perlin f) _ _ _ _)))
    = f (intPt i)

normalAt :: (Maybe Intersection) -> Vector
normalAt Nothing = (0.0,0.0,0.0)
normalAt i@(Just (Intersection _ _ (o,_))) = normal (intPt i) o

intPt :: (Maybe Intersection) -> Point3D
intPt Nothing = (0.0,0.0,0.0)
intPt (Just (Intersection d (Ray start dir) _)) = start <+> (dir *> d)
```

Om problemen te vermijden, definiëren we ook steeds een resultaat indien we te maken krijgen met een lege snijding. Deze waarden kunnen in principe nooit voorkomen, omdat de overige functies steeds controleren of een snijding leeg is. We kunnen echter beter een gepaste waarde voorzien dan de applicatie te laten stoppen bij een dergelijk geval.

We merken ook op dat we bij de functie `colorAt` m.b.v. patroonherkenning een onderscheid maken tussen vaste kleuren en kleuren die afhankelijk zijn van een punt: in het eerste geval kunnen we het kleur direct bepalen, terwijl we in het andere geval het kleur moeten bepalen aan de hand van de functie die vervat zit in de instantie van het `Texture` datatype. Ook zullen de functies `normalAt` en `intPt` de resultaten nog moeten bepalen,

gebruik makend van de functies die we gedefinieerd hebben in de module `HRayMath`.

```
fstPos :: [Double] -> Double
fstPos [] = 0.0
fstPos (l:ls) = if l > 10**(-6) then l else fstPos ls
```

De implementatie van de `fstPos` functie uit het model verloopt volgens de definitie in het model, buiten het feit dat we opnieuw rekening moeten houden met afrondingsfouten.

Ook de implementatie van de functies `closestInt` en `intersect` verlopen analoog met de definitie in het model, naast opnieuw de aangepaste voorwaarde om afrondingsfouten op te vangen en het gebruik maken van patroonherkenning ter vervanging van tuple-functies.

```
closestInt :: Ray -> (Maybe Intersection) -> TexturedObject
              -> (Maybe Intersection)
closestInt r i (o,m)
  = if d > 10**(-6) && ((isNothing i) || d < (intDist i))
    then Just (Intersection d r (o,m))
    else i
  where
    d = fstPos (intRayWith r o)

intersect :: Ray -> [TexturedObject] -> (Maybe Intersection)
intersect r o = foldl (closestInt r) Nothing o
```

De functie `closestInt` maakt gebruik van de booleaanse operatoren `&&` en `||`, die de tegenhangers vormen voor de Funmath operatoren  $\wedge$  en  $\vee$ .

De functies `diff`, `spec` en `shadePt` definiëren we, net als in het model, apart. Omdat in `diff` en `spec` enkele expressies voorkomen die compleet identiek zijn, zou men kunnen geneigd zijn om deze samen te plaatsen in één functie, om herberekening van deze termen te vermijden. Dit is in Haskell echter niet nodig, omdat een krachtige compiler als GHC ervoor zal zorgen dat de expressies niet verschillende keren geëvalueerd worden. Bij de implementatie van de functies volgen we het formele model, en concentreren we ons op het produceren van elegante code.

```

diff :: (Maybe Intersection) -> Light -> Color
diff _ (AmbientLight _) = (0.0,0.0,0.0)
diff i (PointLight pos int) = (int *> ((mkNormVect (intPt i) pos)
                                         *. (normalAt i))) <*> (colorAt i)

spec :: (Maybe Intersection) -> Vector -> Light -> Color
spec _ _ (AmbientLight _) = (0.0,0.0,0.0)
spec i d (PointLight pos int) = int *> (reflCoef * ( ((normalAt i)
                                                       *. h)**(fromIntegral specCoef) ))

where
    h = norm ((d *> (-1)) <+> (mkNormVect (intPt i) pos))
    (Texture _ reflCoef specCoef _ _) = intText i

```

Merk op dat we opnieuw m.b.v. patroonherkenning het onderscheid maken tussen een algemene lichtbron en een punt-lichtbron. We kunnen niet, zoals in het opgestelde model, de functies enkel definiëren voor punt-lichtbronnen, omdat `PointLight` enkel een constructor is, en geen datatype. Omdat de functiedefinitie voor een algemene lichtbron zeer eenvoudig zijn, vormt dit echter geen probleem.

```

shadePt :: Intersection -> Vector -> [TexturedObject] -> Light -> Color
shadePt i d o (AmbientLight int) = int
shadePt i d o l@(PointLight pos int)
    | s = (0.0,0.0,0.0)
    | otherwise = (diff (Just i) l) <+> (spec (Just i) d l)
where
    s = not (isNothing i_s)
        && (intDist i_s) <= dist (intPt (Just i)) pos
    i_s = intersect (mkRay (intPt (Just i)) pos) o

```

Bij de implementatie van de `shadePt` functie uit het model zullen we weer gebruik maken van patroonherkenning, deze keer in combinatie met guards. Zo kunnen we eenvoudig de waarden voor de parameters van de lichtbron bekomen. Bovendien kunnen we bepalen welke expressie moet geëvalueerd worden op basis van een lokale variabele `s`, die aangeeft of het beschouwde snijpunt zich in de schaduw bevindt van een bepaald object of niet.

De functies `reflectPt` en `refractPt` kunnen we eveneens volgens het model implementeren. Bij de `refractPt` functie moeten we wel rekening houden met het resultaat van de

refractDir functie. Zoals aangegeven kan het resultaat van deze functie de nulvector zijn, wat duidt op totale interne reflectie. Indien dit optreedt, zullen we als resultaat een zwarte kleur teruggeven, omdat deze geen wijziging aanbrengt in het resultaat.

```

reflectPt :: Int -> Intersection -> Vector
           -> [TexturedObject] -> [Light] -> Color
reflectPt depth i d = colorPt depth (Ray (intPt (Just i))
                                         (reflectDir d (normalAt (Just i)))) (0.0,0.0,0.0)

refractPt :: Int -> Intersection -> Vector -> Color
           -> [TexturedObject] -> [Light] -> Color
refractPt depth i d b = if refractedDir == (0.0,0.0,0.0)
                        then (\ x y -> (0.0,0.0,0.0))
                        else colorPt depth (Ray (intPt (Just i))
                                                refractedDir) (b *> refrCoef)

where
    refractedDir = refractDir d (normalAt (Just i)) refrIndex
    (Texture _ _ _ refrCoef refrIndex) = intText (Just i)

```

We wijzen erop dat de parameters van het materiaal van het object in het gegeven snijpunt bekomen worden m.b.v. patroonherkenning op een lokale variabele. Op deze manier hebben we geen nood aan de hulpfuncties die we wel nodig hadden bij het samenstellen van het model.

```

colorPt :: Int -> Ray -> Color -> [TexturedObject] -> [Light] -> Color
colorPt (-1) _ _ _ = (0.0, 0.0, 0.0)
colorPt d r@(Ray _ dir) b o l = if (isNothing i)
                                then b
                                else clip $ shadeColor <+> reflectColor
                                       <+> refractColor

where
    shadeColor = foldl (<+>) (0.0,0.0,0.0)
                (map (shadePt (fromJust i) dir o) l)
    reflectColor = if (reflCoef == 0.0) then (0.0, 0.0, 0.0)
                  else (reflectPt (d-1) (fromJust i) dir o l)
                    *> reflCoef

```

```

refractColor = if (refrCoef == 0.0) then (0.0, 0.0, 0.0)
               else (refractPt (d-1) (fromJust i) dir b o l)
                 *> refrCoef

i = intersect r o
(Texture _ reflCoef _ refrCoef _) = intText i

```

De implementatie van de functie `colorPt` functie ziet er een stuk bombastischer uit dan de definitie in het model. Door de lengte van de expressies is de implementatie duidelijker als we lokale variabelen invoeren voor de verschillende kleurbijdragen. Bovendien zullen we het algoritme enkel recursief gaan toepassen indien dit ook echt nodig is: enkel voor objecten waarbij we een reflectie- of refractiecoëfficiënt hebben die verschillend is van nul, zullen we de corresponderende functie aanroepen om bij te dragen tot het resultaat. Deze optimalisatie kan niet zomaar doorgevoerd worden door de compiler. Die zal de expressies die voorkomen in de definitie van de recursieve functies evalueren, om die dan uiteindelijk met een nulvector te gaan vermenigvuldigen. Bij het bepalen van de term `shadeColor` gebruiken we de functie `map`, die een functie toepast op elk element van een lijst.

```

rayTracePt :: Int -> Scene -> Point3D -> Color
rayTracePt d (Scene (Camera eye _) b o l) p
            = colorPt d (Ray p (mkNormVect eye p)) b o l

rayTrace :: Int -> Resolution -> Scene -> Image
rayTrace d r s@(Scene (Camera _ dim) _ _ _)
          = (rayTracePt d s) . (mapToWin r dim)

```

De laatste functies, die de kern van het algoritme vervolledigen, kunnen we zonder restricties implementeren volgens het model. We maken opnieuw gebruik van patroonherkenning en de Haskell-functie `'.'`, die toelaat om twee functies samen te stellen.

We merken nog op dat de functies `rayTrace`, `rayTracePt`, `closestInt`, `reflectPt` en `refractPt` een extra parameter hebben. Deze parameter geeft de resterende recursiediepte aan. Indien de `closestInt` de waarde `-1` herkent, zal de functie stoppen met de recursie en gewoon een zwarte kleur teruggeven als resultaat. Op die manier vermijden we oneindige recursie. Die kan optreden bij een eenvoudige 3D-wereld die bestaat uit 2 spiegelende vlakken, die evenwijdig met, en aan weerszijden van het kijkvenster opgesteld zijn (zie paragraaf 2.3.3).

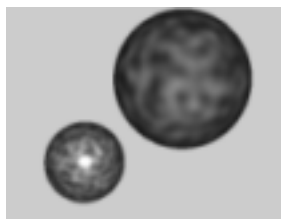
### 3.4 De module `HRayPerlin`

Zoals reeds vermeld ondersteunen we twee types kleuren voor materialen van objecten: een vast kleur dat over het volledige oppervlak van het object hetzelfde is, en een kleur die afhankelijk is van een punt uit de ruimte. We maken hierbij gebruik van de Perlin noise techniek, die toelaat om verschillende effecten, zoals wolken, marmer en hout, te bekomen. De techniek maakt gebruik van een lijst van de getallen 0 tot 255, in een willekeurige volgorde, en van een even lange lijst van vectoren, die elk een zogenaamde *gradiënt* aanduiden. Om de kleur van een punt uit de ruimte te bepalen, wordt de ruimte onderverdeeld in een rooster (waarvan de onderverdeling kan bepaald worden m.b.v. parameters). Voor elk van de roosterpunten die het dichtst liggen bij het beschouwde punt (in een 3D-wereld zijn dit  $2^3 = 8$  roosterpunten), wordt a.d.h.v. de twee lijsten een gradiënt vector bepaald. De gewogen som van deze gradiënt vectoren wordt dan vermenigvuldigd met een kleur om de kleur van het beschouwde punt uit de ruimte te bekomen. Op deze manier zal een naburig punt een kleur hebben die lichtjes afwijkt, waardoor we de vernoemde effecten kunnen bekomen.

In de module `HRayPerlin` hebben we een functie `noise` geïmplementeerd, die voor een gegeven grootte van het rooster, een punt uit de ruimte en de twee lijsten, een factor uit het interval  $[-1, 1]$  zal bepalen waarmee we een gegeven kleur componentsgewijs kunnen vermenigvuldigen. We geven hier enkel het type van deze functie weer, voor de implementatie verwijzen we naar de eigenlijke code. We voorzien ook twee hulpfuncties `semiTurbulence` en `turbulence` die zullen helpen met het implementeren van de verschillende effecten. Naast de afmetingen van het rooster, hebben deze functies een argument dat de frequentie voorstelt waarmee we het effect bekomen.

```
noise :: Double -> Point3D -> [Int] -> [Vector] -> Double
```

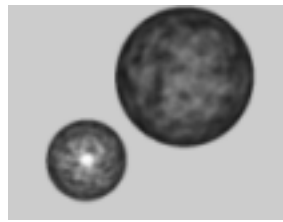
We vermelden de functies die de verschillende effecten toelaten. Hierbij gaan we niet dieper in op de nodige argumenten, hiervoor verwijzen we naar [19, 20].



Figuur 3.1: Storingseffect

Een eerste effect wordt bekomen door de factor die we berekenen met de `noise` functie,

te vermenigvuldigen met een gegeven basiskleur. Op deze manier lijkt het alsof het object vuil of versleten is. De functie `perlinSolid` implementeert dit effect.



Figuur 3.2: Semi-turbulentie effect

Een effect dat verderbouwt op het eerste is een wolk-effect, dat we kunnen bekomen m.b.v. de functie `perlinSemiTurbulence`. Door de bijdragen die bepaald worden met verschillende frequenties samen te tellen, bekomen we een fijner effect dan het vorige.



Figuur 3.3: Turbulentie en likkend vuur effect

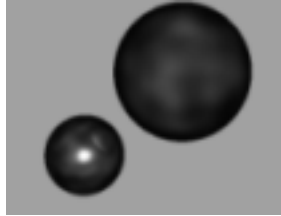
Indien we bij het optellen van de bijdragen de absolute waarde van de `noise` functie nemen i.p.v. de gewone waarden, bekomen we een effect waarbij het materiaal scherpe overgangen vertoont. Dit effect bekomen we met de functie `perlinTurbulence`. Als we bij het dit effect een basiskleur bijtellen, kunnen we met de gepaste kleuren een likkend vuur effect bekomen. Hiervoor beschikken we over de functie `perlinFire`.



Figuur 3.4: Marmertextuur

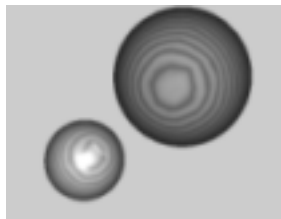
Een erg indrukwekkend effect dat we eveneens kunnen bekomen m.b.v. Perlin noise, is een marmer-textuur. Marmer wordt gekenmerkt door het voorkomen van gekleurde nerven in het materiaal. Een eerste functie `perlinMarble` past dit effect toe op één kleur, waardoor we echter zwarte nerven op een gekleurd materiaal verkrijgen, wat niet steeds

gewenst is. Wanneer we met de functie `perlinMarlbeBase` een basiskleur hierbij optellen, kunnen we ook materialen met gekleurde nerven bekomen. Indien we het verschil maken tussen een witte kleur en het bekomen kleur, kunnen we het effect verder naar wens aanpassen.



Figuur 3.5: Plasma effect

Als we met behulp van het eerste effect een kleur bekomen voor elk van de drie basiskleuren rood, groen en blauw en deze kleuren gaan samentellen, bekomen we een plasma effect. Dit kunnen we bekomen m.b.v. de functie `perlinPlasma`. Het effect van deze functie is niet echt duidelijk in een afbeelding met enkel grijswaarden. Voor een versie in kleur verwijzen we naar paragraaf 5.4.



Figuur 3.6: Houttextuur

Een laatste krachtige toepassing is het aanmaken van houttexturen, die gekenmerkt worden door concentrische grillige ringen. Indien we dit combineren met de gepaste kleuren (verschillende tinten bruin) kunnen we een erg realistische houttextuur bekomen. Dit effect kunnen we bekomen m.b.v. de `perlinWood` functie.

We vermelden nog dat we bij de implementatie van deze effecten gebruik hebben gemaakt van vaste permutaties van lijsten i.p.v. willekeurige permutaties. Dit werd gedaan om efficiëntieredenen. Het enige gevolg hiervan is dat een object steeds dezelfde patronen zal vertonen wanneer de afbeelding opnieuw berekend wordt, indien er niets gewijzigd wordt aan de parameters van het object.

## 3.5 De module HRayOutput

Om de resultaten van het ray tracing algoritme te kunnen visualiseren, voorzien we enkele hulpfuncties. We bespreken deze kort en maken er gebruik van bij het implementeren van een gebruikersinterface, die we bespreken in hoofdstuk 4.

Als eerste bespreken we de functie `getImage`, die voor een gegeven beschrijving van een 3D-wereld een lijst van kleuren bepaalt, die de afbeelding voorstelt. Aan de hand van deze lijst kunnen we dan bvb. een bestand uitschrijven dat kan gelezen worden door grafische programma's als Photoshop en Gimp.

De implementatie van deze functie gebeurt d.m.v. van een lijstcomprehensie. We overlopen de pixelcoördinaten die binnen de opgegeven resolutie liggen, en bepalen zo rij per rij en kolom per kolom de gewenste afbeelding, die we kunnen bepalen m.b.v. de `rayTrace` functie.

```
getImage :: Int -> Resolution -> Scene -> [Color]
getImage d r@(rx,ry) s = [image (fromIntegral x,fromIntegral (-y))
                          | y <- [-(ry-1)..0], x <- [0..(rx-1)]]

where
    image = rayTrace d r s
```

Om aan de hand van deze functie een bestand te kunnen uitschrijven, moeten we de lijst van kleuren kunnen omzetten naar een bestandsformaat om een afbeelding voor te stellen. Voor de eenvoud maken we hierbij gebruik het Portable Pixmap (PPM) formaat, om dit een heel eenvoudige syntax heeft, en geen compressie gebruikt. Met behulp van de functie `createPPM`, die als argumenten een resolutie en een lijst van kleuren aanneemt, construeren we een string die de inhoud van het bestand voorstelt die de afbeelding bevat. We maken eerst een korte header aan die de resolutie en het gebruikte kleurenmodel aangeeft. Daarna zetten we d.m.v. de `stringify` functie de kleuren een voor een om naar de drie componenten gescheiden door een spatie. Merk op dat we hierbij waarden tussen 0 en 255 uitschrijven, en dat we daarbij de reële waarden van de componenten afronden. Dit is noodzakelijk omdat een component in het RGB-model dat we hanteren slechts 256 mogelijke schakeringen heeft.

```

createPPM :: Resolution -> [Color] -> String
createPPM (w,h) c =
    ('P3\n'++) . shows w. (' '). shows h. (''\n255\n'++)
    . stringify c $ ' '
    where
        stringify = flip $ foldr showC
        showC (r,g,b) = shows (round (r*255)). (' ')
            . shows (round (g*255)). (' ')
            . shows (round (b*255)) . (' ')

```

Als laatste hebben we de functie `getRenderTime`, die aan de hand van het aantal verstreken picoseconden de tijdspanne bepaalt die nodig was om de afbeelding te genereren. Deze functie gebruiken we om beperkte feedback te kunnen geven aan de gebruiker van de tekstuele en grafische interface die we bespreken in hoofdstuk 4.

```

getRenderTime :: Integer -> Integer -> String
getRenderTime before after = (show min) ++ 'm' ++ secStr ++ 's'
    where
        total = div (after - before) (1012)
        sec = mod total 60
        secStr = if (sec < 10) then "0"++ (show sec) else show sec
        min = div total 60

```

# Hoofdstuk 4

## Inputspecificatie en gebruikersinterface

We bespreken de nodige voorzieningen om de ray tracer die geïmplementeerd werd bruikbaar te maken. We voorzien een inputspecificatie die toelaat om d.m.v. tekstbestanden een beschrijving van de 3D-wereld te geven waarvoor we een afbeelding willen genereren. Om die afbeelding te kunnen visualiseren voorzien we een eenvoudige tekstuele interface om een afbeelding te genereren en die uit te schrijven naar een Portable Pixmap (PPM) bestand. We bespreken ook een eenvoudige grafische gebruikersinterface (GUI), die toelaat om een tekstbestand met een beschrijving van een 3D-wereld in te laden en een afbeelding te genereren in een venster.

### 4.1 Inputspecificatie

Om een beschrijving van een 3D-wereld te kunnen inladen, moeten de syntax van een dergelijke beschrijving vastliggen. Bovendien moeten we deze beschrijving omzetten in een Haskell datatype, zodat het Haskell programma ermee kan omgaan. Dit alles kunnen we bereiken door gebruik te maken van Happy [22], een parser generator die gelijkaardig is met de yacc tool voor C. Door de syntax van de beschrijving weer te geven in een BNF-achtige notatie, kan Happy Haskell code genereren die nagaat of de inhoud van een tekstbestand voldoet aan de syntax, en die inhoud omzetten naar een Haskell datatype. Die code komt terecht in een module `HRayParser`, waarvan we dan kunnen gebruik maken alsof we deze zelf zouden geschreven hebben. De syntax-specificatie die we meegeven aan Happy is terug te vinden in het bestand `HRayParser.y`. De Haskell module die door Happy gegenereerd wordt, komt terecht in het bestand `HRayParser.hs`.

In deze module definiëren we een functie `readDescr`, die de inhoud van tekstbestand

controleert op de syntax, en indien mogelijk een instantie van het datatype `RenderDescr` aanmaakt. De functie en het datatype worden door de module beschikbaar gesteld. Alle andere functies die nodig zijn binnenin de module worden voor de gebruiker verborgen gehouden. We vermelden de het type van de functie en de beschrijving van het datatype:

```
data RenderDescr = RenderDescr Resolution Int Scene
  readDescr :: String -> RenderDescr
```

## 4.2 Tekstuele en grafische gebruikersinterfaces

Om de ray tracer die we ontwikkeld hebben gebruiksvriendelijk te maken, implementeren we een eenvoudige tekstuele en grafische interface. Deze laat toe om een afbeelding te genereren a.d.h.v. een tekstbestand die de beschrijving van een 3D-wereld bevat.

### 4.2.1 Tekstuele gebruikersinterface

De tekstuele gebruikersinterface kan gebruikt worden m.b.v. een console venster. Het programma neemt als argumenten de locatie van het inputbestand en de gewenste locatie van het bestand dat de afbeelding bevat (in het PPM dataformaat). Wanneer verkeerde argumenten meegegeven worden, wordt een gepaste foutboodschap teruggegeven.

De implementatie van deze interface is terug te vinden in `HRayMain.lhs`. Om het programma te compileren m.b.v. GHC, voeren we de volgende opdracht uit:

```
ghc --make -O HRayMain -o HRay.exe
```

Op deze manier bekomen we een programma `HRay.exe` dat we kunnen uitvoeren met de gepaste argumenten. De optie `--make` zorgt ervoor dat alle gebruikte modules met elkaar gelinkt worden, terwijl de optie `-O` voor extra optimalisatie zorgt. Om een beschrijving van een wereld om te zetten in een afbeelding, voeren we dit uit:

```
HRay.exe input.hry output.ppm
```

Hierbij is `input.hry` een tekstbestand met de beschrijving, en zal `output.ppm` (na uitvoering) de gewenste afbeelding bevatten.

Na uitvoering krijgt de gebruiker een boodschap waarin wat feedback informatie zit vervat. Hierbij maken we gebruik van de `getRenderTime` functie uit de `HRayOutput` module.

```
Image output written to output.ppm in PPM format
```

```
Time needed to render the image: 0m07s
```

De enige functie die we moeten implementeren is de `main` functie, die zal uitgevoerd worden als hoofdprogramma. Het type van deze functie is `IO()`, wat erop wijst dat er I/O bewerkingen zullen gebeuren. We maken gebruik van de `do`-clausule, die toelaat om o.a. de commandolijn-argumenten op te halen, de inhoud van een bestand te lezen en om een bestand uit te schrijven.

We gaan na of het aantal argumenten klopt, en geven een gepaste foutboodschap terug indien er minder of meer dan twee argumenten worden opgegeven. Ook wordt er gecontroleerd of het opgegeven bestand met de beschrijving wel bestaat. Daarop wordt de inhoud gelezen, en omgezet in een Haskell datatype m.b.v. de functie `readDescr`. Indien de inhoud van het bestand niet voldoet aan de opgelegde syntax, zal er een door Happy gegenereerde foutboodschap teruggegeven worden.

Na het bijhouden van de processortijd, genereren we de afbeelding m.b.v. `getImage`, die we meteen omzetten in een string én uitschrijven naar het opgegeven bestand. Voordat het programma stopt, wordt ook nog de vermelde feedback uitgeschreven.

```
main = do args <- getArgs
  if (length args /= 2)
    then error "Usage: HRay <input scene path> <output ppm path>\n"
    else do let input = head args
              output = head (tail args)
              exists <- doesFileExist input
              case exists of
                True -> do fileContent <- readFile input
                           let (RenderDescr res depth scene)
                               = readDescr fileContent
                           timeBefore <- getCPUTime
                           writeFile output (createPPM res
                                               (getImage depth res scene))
                           timeAfter <- getCPUTime
```

```

putStr $ "Image output written to "
        ++ output ++ " in PPM format\n"
putStr $ "Time needed to render the image: "
        ++ (getRenderTime timeBefore timeAfter) ++ "\n"
False -> error $ "File \"'\" ++ input
        ++ "\"' does not exist.\\"

```

## 4.2.2 Grafische gebruikersinterface (GUI)

Met behulp van `gtk2hs` werd ook een eenvoudige grafische gebruikersinterface geïmplementeerd. Deze laat toe om een beschrijving in te laden, een samenvatting van deze beschrijving te bekijken, een afbeelding te genereren én natuurlijk om die afbeelding te bekijken.

De implementatie van de GUI is terug te vinden in het bestand `HRayGUI.lhs`, en maakt zoals vermeld gebruik van het grafische pakket `gtk2hs`. Dit pakket steunt op de reeds bestaande grafische bibliotheek `Gtk+`, en bestaat voornamelijk uit bindingen naar de functies in deze bibliotheek. Het pakket ondersteunt een declaratieve stijl, en ondersteunt alle klassieke widgets die beschikbaar zijn in een volwassen grafische bibliotheek.

Net als de tekstuele interface, kunnen we het programma als volgt compileren en uitvoeren:

```

ghc --make -O HRayGUI -o HRayGUI.exe
      HRayGUI.exe

```

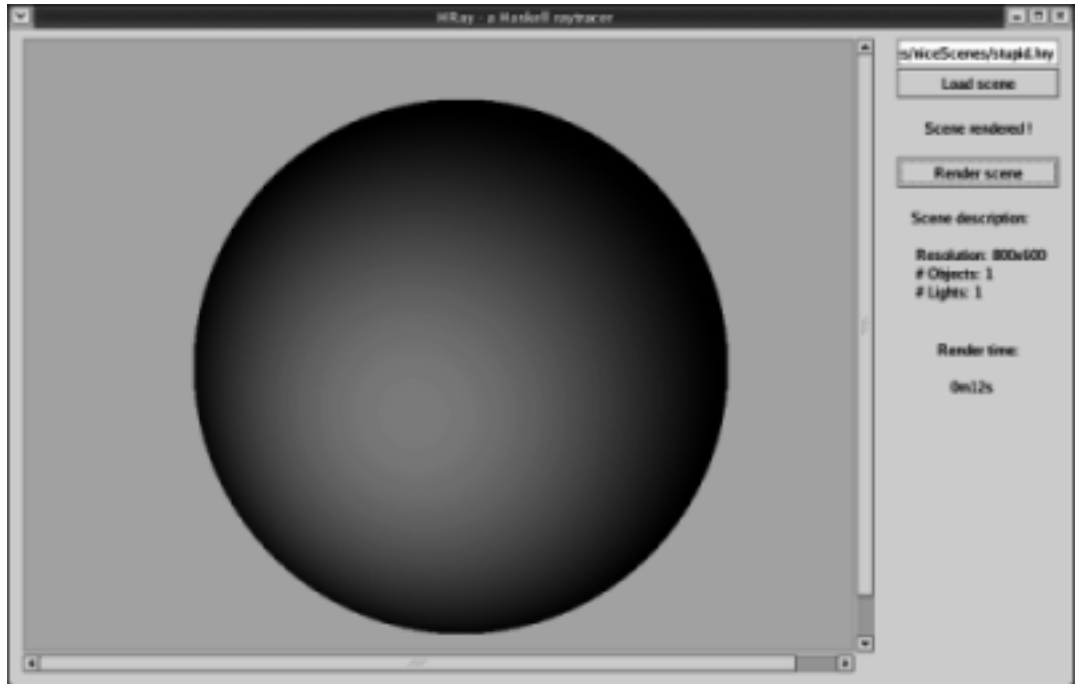
De opbouw van de GUI zullen we niet bespreken, omdat deze vrij eenvoudig is. We merken op dat er opnieuw gebruikt gemaakt wordt van het `IO()` type. Het belangrijkste deel van de code bespreken we wel: de acties die uitgevoerd worden wanneer een van de twee knoppen in het venster geactiveerd wordt.

```

onClicked loadButton $ do
    fileStr <- (entryGetText entry)
    renderDescr <- loadScene fileStr messageLabel
                    descrLabel renderTimeLabel
    writeIORef renderDescrRef renderDescr

```

Bij de opbouw van het venster, moeten we d.m.v. de `onClicked` functie per knop speci-



Figuur 4.1: De grafische gebruikersinterface m.b.v. gtk2hs

ficeren welke acties moeten uitgevoerd worden. Als de knop om een bestand in te laden ingedrukt wordt, laden we de functie in m.b.v. de functie `loadScene`. Deze functie zal nagaan of het bestand bestaat, en proberen om een instantie van het `RenderDescr` datatype aan te maken. Indien er zich een syntactische fout in het bestand bevindt, zal de applicatie echter termineren. De oorzaak daarvan is de fout die opgeworpen wordt door de parser (die gegenereerd werd m.b.v. Happy). Om dit euvel te verhelpen, zouden we de fout moeten opvangen, en die op een gepaste manier behandelen.

```
onClicked renderButton $ do
  renderDescr <- readIORef renderDescrRef
  let loop = do
    n <- eventsPending
    if n == 0 then return ()
      else mainIterationDo False >> loop
  if isNothing renderDescr then
    labelSetMarkupWithMnemonic messageLabel
      “\n<b>Please load a scene.</b>\n”
```

```

else do
    timeBefore <- getCPUtime
    labelSetMarkupWithMnemonic messageLabel
        (‘\n<span foreground=’’red’’>’’
        ++ ‘<b>Rendering scene...</b></span>\n’)
    loop
    pixBuf <- renderScene renderDescr descrLabel
    labelSetText messageLabel ‘\nScene rendered !\n’
    imageSetFromPixbuf image pixBuf
    timeAfter <- getCPUtime
    labelSetText renderTimeLabel $ ‘\nRender time:\n\n’
        ++ (getRenderTime timeBefore timeAfter) ++ ‘\n’

```

Wanneer op de knop geklikt wordt om de afbeelding te genereren, wordt deze functie uitgevoerd. We vermelden enkel de belangrijkste aspecten die hierbij naar voor komen.

De instantie van het datatype `RenderDescr` die we bekomen hebben door een bestand in te laden, halen we uit de `IORef` variabele. Dit is nodig omdat Haskell geen globale variabelen ondersteunt, en een Haskell programma in principe stateless is. Op basis van de inhoud van deze instantie, die een instantie is van het `Maybe` datatype dat we reeds eerder gebruikt hebben, voeren we een gepaste actie uit. Indien er geen beschrijving geladen is, zorgen we ervoor dat een gepaste boodschap wordt weergegeven op een daartoe bestemd label.

In het andere geval genereren we de afbeelding m.b.v. de `renderScene` functie, en vervangen we de huidige afbeelding in het venster met de nieuwe afbeelding. Met behulp van de lokale variabele `loop` zorgen we ervoor dat de wijziging aan het label `messageLabel` zichtbaar wordt. Na het genereren van de afbeelding leveren we opnieuw gepaste feedback naar de gebruiker.

Naast het `gtk2hs` pakket zelf, waren er enkele extra's nodig om de overgang van een lijst van kleur naar een afbeelding die kan gepresenteerd worden. De reden daarvoor is dat `gtk2hs` gebruikt maakt van een `Pixbuf` datatype, dat niet 100% compatibel is met de manier waarop kleuren voorgesteld worden in het model. Deze low-level code werd samengebundeld in de module `PixBufInternals`.

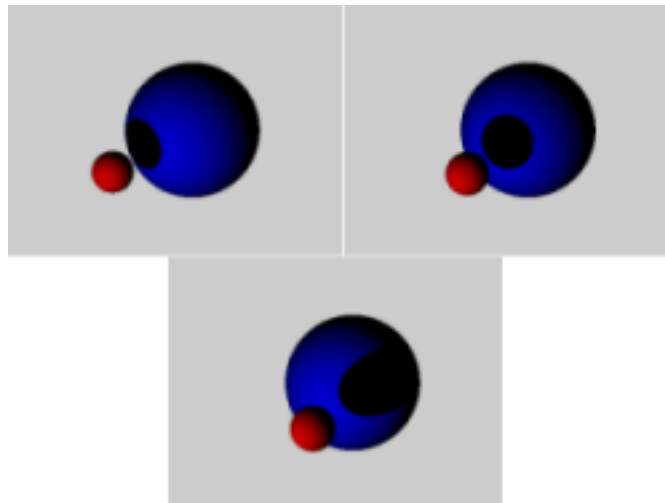
# Hoofdstuk 5

## Enkele afbeeldingen

We bekijken enkele afbeeldingen die gegenereerd werden m.b.v. de geïmplementeerde ray tracer. We geven telkens een korte beschrijving van de technieken en de plaatsing van de objecten.

### 5.1 Schaduw van een bol op andere bol

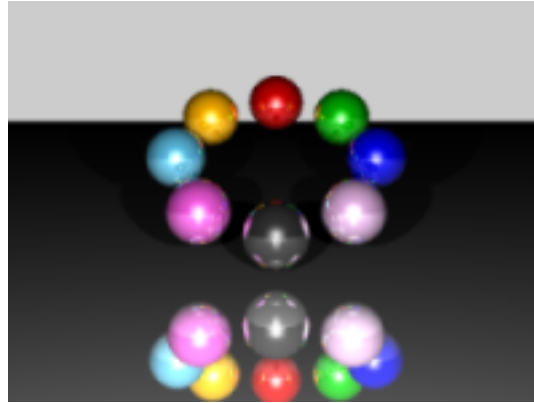
De rode bol wordt telkens een beetje naar rechts verplaatst, waardoor de schaduw die op de blauwe bol geworpen wordt wijzigt van vorm. Vooral in de onderste afbeelding is dit mooi zichtbaar.



Figuur 5.1: Schaduw onder verschillende hoeken

## 5.2 Reflectie van bollen in een vlak

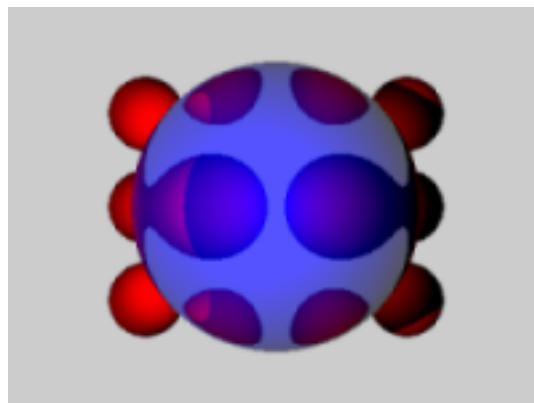
We plaatsen 8 reflecterende bollen boven een spiegeland vlak. Het vlak weerspiegelt de bollen, en die reflectie is op zich terug zichtbaar in de bollen zelf. Bovenaan de zwarte bol zien we de reflectie van de andere omringende bollen.



Figuur 5.2: Reflectie van bollen in een spiegeland vlak

## 5.3 Transparantie

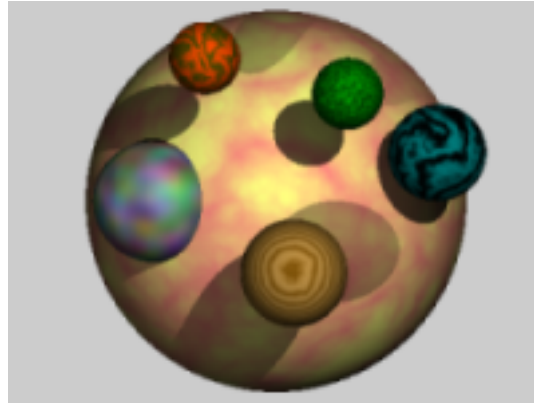
Op deze afbeelding zien we een blauwe doorzichtige bol, die zich voor een rooster van 12 rode ballen bevindt. De rode ballen zijn zichtbaar door de blauwe bal, en lijken door de breking van het licht te versmelten met de blauwe bol. Ook de schaduw van de blauwe bol op het rooster is duidelijk zichtbaar.



Figuur 5.3: Rooster van bollen bekeken door een transparante bol

## 5.4 Perlin noise materialen

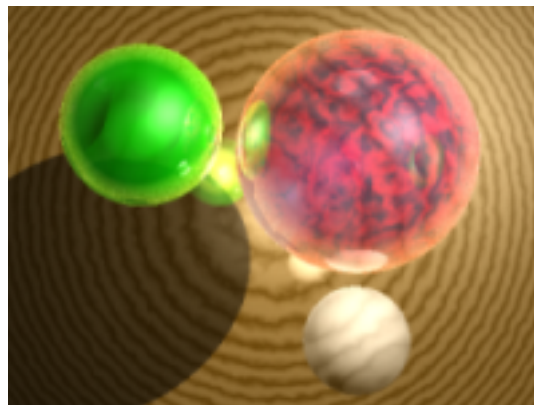
Dit is een afbeelding waarbij voor elk object Perlin noise materialen gebruikt worden. We zien duidelijk het vuur-effect in de grote bol, de houttextuur in de bol onderaan en de marmertextuur in de bol aan de rechterkant.



Figuur 5.4: Bollen met verschillende soorten perlin noise materialen

## 5.5 Combinatie van effecten

Deze combinatie van effecten illustreert de mogelijkheden van de ray tracer die gemodelleerd en geïmplementeerd werd. Alle objecten zijn reflectief, iedere bol is doorzichtig en de grote bol gebruikt, net als het vlak, een Perlin noise materiaal.



Figuur 5.5: Combinatie van de verschillende effecten

# Hoofdstuk 6

## Conclusies

In deze scriptie hebben we stap voor stap een formeel model uitgewerkt voor een ray tracer. Hierbij hebben we gebruik gemaakt van de verschillende functionaliteiten die voor handen zijn in het Funmath formalisme, waaronder generische functionalen en tupels als functies. In het model werd m.b.v. niet-verder gespecificeerde verzamelingen en hulpfuncties voldoende abstractie voorzien, zodat we slechts bij de implementatie de specificatie van de verzamelingen moeten vastleggen. We hebben enkele eenvoudige eigenschappen aangetoond, waarmee we fouten en dubbelzinnigheden in een vroeg stadium waren.

Op basis van het formele model hebben we dan een implementatie ontwikkeld voor de applicatie in de functionele programmeertaal Haskell. Hierbij moesten we rekening houden met de tekorten van Haskell t.o.v. Funmath, en die op een gepaste manier opvangen. Door gebruik te maken van het formele model bekomen we eenvoudige, korte en elegante functie-definities die makkelijk te onderhouden zijn. Het model suggereert bvb. de implementatie van “uitgebreide operatoren”, waardoor de functies definities eleganter worden. Zonder de tussenstap van het model zou de nood voor dergelijke operatoren waarschijnlijk niet naar voor komen. Hieruit blijkt duidelijk hoe het model bijdraagt tot betere code.

Het uitbreiden van de functionaliteit, zoals ondersteuning voor andere types van objecten en extra effecten, is zeer eenvoudig. Om een nieuw type object toe te voegen, volstaat het om de functies die de normaal in een snijpunt en de snijding met een straal voor dat object bepalen, uit te breiden.

Naast de implementatie van het model hebben we ook een tweede type van materialen voorzien, dat toelaat om effecten als marmer en hout te bekomen. Deze effecten kunnen we bekomen d.m.v. van de Perlin noise techniek, die toelaat om willekeurige kleuren te genereren op een gecontroleerde manier (i.e. waarbij rekening gehouden wordt met de omringende kleuren). Daarnaast werd ook een eenvoudig invoerformaat gespecificeerd, zodat de beschrijving van een wereld waarvan we een afbeelding willen aanmaken kan

doorgegeven worden met een tekstbestand.

We voorzien twee gebruikersinterfaces die de gebruiker toelaten om de functie, die een afbeelding beschrijft, om te zetten in een eigenlijke afbeelding. Een eenvoudige tekstuele gebruikersinterface laat toe om een beschrijving in te laden en de gewenste locatie voor het uitvoerbestand te specificeren. De grafische gebruikersinterface laat toe om verschillende afbeeldingen na elkaar te genereren, en om telkens een kleine samenvatting van de ingevoerde wereld te bekijken.

## 6.1 Verder onderzoek

Er zijn verschillende verbeteringen mogelijk aan de Haskell implementatie van de ray tracer. Een belangrijk aspect is het verbeteren van de prestatie. Hiervoor kunnen we gebruik maken van zogenaamde *pragma*'s die ondersteund worden door GHC. Dit zijn kleine stukjes commentaar, die de GHC compiler helpen om beslissingen te nemen omtrent de manier waarop stukken code behandeld worden. De implementatie van de Perlin noise techniek kan korter en sneller gemaakt worden, ondermeer door de verbeteringen voorgesteld in [21].

Naast de prestatie kan ook de ondersteunde functionaliteit verbeteren. Hierbij denken we aan het ondersteunen van meerdere types objecten en lichtbronnen, en het implementeren van anti-aliasing (waarbij de kleur van omringende pixels in rekening gebracht wordt). Om de ondersteuning voor bijkomende types objecten nog te verbeteren, kunnen we gebruik maken van type klassen in Haskell. Omdat we de Haskell 98 standaard volgen in de huidige implementatie, is dit wegens de beperkingen van die standaard omtrent type-klassen niet mogelijk. Omdat we het type `TexturedObject` gedefinieerd hebben als een tuple bestaande uit een instantie van het `Object` datatype en een materiaal, kunnen we niet zonder het `Object` datatype. Als we de functies `normal` en `intrayWith` in een type-klasse implementeren, dan beschikken we, zonder een uitbreiding op de Haskell 98 standaard (nl. existentiële types), niet langer over een dergelijk datatype.

De gebruiksvriendelijkheid van de voorziene gebruikersinterfaces kan verbeterd worden door gepaste foutboodschappen te voorzien wanneer een beschrijving ingeladen wordt die syntactisch niet correct is. Bovendien kan het gebruik van XML voor een dergelijke beschrijving bijkomende voordelen opleveren op het gebied van uitbreidbaarheid van de functionaliteit [16].

## 6.2 Verwant onderzoek

Wegens de opmars van multimedia toepassingen in de digitale wereld, neemt ook het belang van dergelijke applicaties toe. De declaratieve en functionele programmeertaal Haskell wordt door sommigen zelfs gezien als dé taal bij uitstek om dergelijke applicaties te ontwikkelen [4].

De voorzieningen staan nog in de kinderschoenen, maar er is steeds meer interesse naar applicaties die ontworpen werden m.b.v. Haskell. De ontwikkeling van grafische bibliotheken zoals `gtk2hs` [8], `FranTk` en `wxHaskell`, krachtige toepassingen zoals `darcs` [23] en technieken zoals `arrows` [24] dragen hiertoe bij.

# Bijlage A

## Haskell implementatie

### A.1 HRayMath

```
module HRayMath where

type Point2D = (Int,Int)
type Point3D = (Double,Double,Double)
type Vector = (Double,Double,Double)
type Resolution = (Int,Int)
type Dimension = (Int,Int)
data Object = Sphere Double Point3D
            | Plane (Double,Double,Double,Double)
data Ray = Ray Point3D Vector

mkRay :: Point3D -> Point3D -> Ray
mkRay p1 p2 = Ray p1 (mkNormVect p1 p2)

(<+>) :: (Double,Double,Double) -> (Double,Double,Double)
      -> (Double,Double,Double)
(x1,y1,z1) <+> (x2,y2,z2) = (x1+x2, y1+y2, z1+z2)

(<->) :: Point3D -> Point3D -> Vector
(x1,y1,z1) <-> (x2,y2,z2) = (x1-x2,y1-y2,z1-z2)

(<*>) :: (Double,Double,Double) -> (Double,Double,Double)
      -> (Double,Double,Double)
(x1,y1,z1) <*> (x2,y2,z2) = (x1*x2,y1*y2,z1*z2)

(*>) :: (Double,Double,Double) -> Double -> (Double,Double,Double)
(x,y,z) *> f = (x*f,y*f,z*f)

maxF :: Double -> (Double,Double,Double) -> (Double,Double,Double)
maxF f (x,y,z) = (max x f, max y f, max z f)
```

```

minF :: Double -> (Double,Double,Double) -> (Double,Double,Double)
minF f (x,y,z) = (min x f, min y f, min z f)

(*.) :: Vector -> Vector -> Double
(x1,y1,z1) *. (x2,y2,z2) = x1*x2 + y1*y2 + z1*z2

len :: Vector -> Double
len v = sqrt (v *. v)

norm :: Vector -> Vector
norm v
  | len v < 10**(-9) = (0.0,0.0,0.0)
  | otherwise       = v *> (1/(len v))

mkNormVect :: Point3D -> Point3D -> Vector
mkNormVect v w = norm (w <-> v)

dist :: Point3D -> Point3D -> Double
dist p0 p1 = sqrt ((p1 <-> p0) *. (p1 <-> p0))

clip :: (Double,Double,Double) -> (Double,Double,Double)
clip = (maxF 0.0) . (minF 1.0)

solveq :: (Double,Double,Double) -> [Double]
solveq (a,b,c)
  | (d < 0)   = []
  | (d > 0)   = [(- b - sqrt d)/(2*a), (- b + sqrt d)/(2*a)]
  | otherwise = [-b/(2*a)]
  where
    d = b*b - 4*a*c

intRayWith :: Ray -> Object -> [Double]

intRayWith (Ray start dir) (Sphere rad cen)
  = solveq (dir *. dir, 2*(dir *. d), (d *. d) - rad^2)
  where
    d = start <-> cen

intRayWith (Ray start dir) (Plane (a,b,c,d))
  = if (abs(part) < 10**(-9)) then []
    else [- (d + ((a,b,c) *. start) ) / part]
  where
    part = (a,b,c) *. dir

```

```

normal :: Point3D -> Object -> Vector

normal p (Sphere rad cen) = norm ((p <-> cen) *> (1/rad))
normal _ (Plane (a,b,c,d)) = norm (a,b,c)

reflectDir :: Vector -> Vector -> Vector
reflectDir i n = i <-> (n *> (2*(n *. i)))

refractDir :: Vector -> Vector -> Double -> Vector
refractDir i n r = if (v < 0) then (0.0, 0.0, 0.0)
                  else norm $ (i *> r_c) <+> (n *> (r_c*(abs c) - sqrt v))
  where
    c   = n *. (i *> (-1))
    r_c = if (c < 0) then r else 1/r
    v   = 1 + (r_c^2) * (c^2 - 1)

mapToWin :: Resolution -> Dimension -> Point2D -> Point3D
mapToWin (rx,ry) (w,h) (px,py) = (x/rxD,y/ryD,0.0)
  where
    (rxD,ryD) = (fromIntegral rx, fromIntegral ry)
    (pxD,pyD) = (fromIntegral px, fromIntegral py)
    (wD,hD)   = (fromIntegral w, fromIntegral h)
    (x,y)     = ( (pxD-rxD/2)*wD, (pyD-ryD/2)*hD )

```

## A.2 HRayEngine

```

module HRayEngine (Resolution, Color, Diff(Solid,Perlin),
                  Texture(Texture), TexturedObject,
                  Light(AmbientLight,PointLight), Scene(Scene),
                  Camera(Camera), rayTrace) where

import Maybe
import HRayMath

type Color = (Double,Double,Double)
data Diff = Solid Color |
          Perlin (Point3D -> Color)
data Texture = Texture Diff Double Int Double Double
type TexturedObject = (Object,Texture)
type Intensity = (Double,Double,Double)
data Light = PointLight Point3D Intensity
           | AmbientLight Intensity
data Camera = Camera Point3D Dimension
data Scene = Scene Camera Color [TexturedObject] [Light]

```

```

data Intersection = Intersection Double Ray TexturedObject
type Image = Point2D -> Color

intDist :: (Maybe Intersection) -> Double
intDist Nothing = 0.0
intDist (Just (Intersection d _ _)) = d

intText :: (Maybe Intersection) -> Texture
intText Nothing = Texture (Solid (0.0,0.0,0.0)) 0.0 0 0.0 0.0
intText (Just (Intersection _ _ (_,t))) = t

colorAt :: (Maybe Intersection) -> Color
colorAt Nothing = (0.0,0.0,0.0)
colorAt (Just (Intersection _ _ (_,Texture (Solid color) _ _ _ _)))
    = color
colorAt i@(Just (Intersection _ _ (_,Texture (Perlin f) _ _ _ _)))
    = f (intPt i)

normalAt :: (Maybe Intersection) -> Vector
normalAt Nothing = (0.0,0.0,0.0)
normalAt i@(Just (Intersection _ _ (o,_))) = normal (intPt i) o

intPt :: (Maybe Intersection) -> Point3D
intPt Nothing = (0.0,0.0,0.0)
intPt (Just (Intersection d (Ray start dir) _))
    = start <+> (dir *> d)

fstPos :: [Double] -> Double
fstPos [] = 0.0
fstPos (l:ls) = if l > 10**(-6) then l else fstPos ls

closestInt :: Ray -> (Maybe Intersection) -> TexturedObject
             -> (Maybe Intersection)
closestInt r i (o,m) = if d > 10**(-6)
                       && ((isNothing i) || d < (intDist i))
                       then Just (Intersection d r (o,m))
                       else i
    where
        d = fstPos (intRayWith r o)

intersect :: Ray -> [TexturedObject] -> (Maybe Intersection)
intersect r o = foldl (closestInt r) Nothing o

diff :: (Maybe Intersection) -> Light -> Color
diff _ (AmbientLight _) = (0.0,0.0,0.0)

```

```

diff i (PointLight pos int) = (int *> ((mkNormVect (intPt i) pos)
                                     *. (normalAt i))) <*> (colorAt i)

spec :: (Maybe Intersection) -> Vector -> Light -> Color
spec _ _ (AmbientLight _) = (0.0,0.0,0.0)
spec i d (PointLight pos int) = int *> (reflCoef * ( ((normalAt i)
                                                       *. h)**(fromIntegral specCoef) ))

  where
    h = norm ((d *> (-1)) <+> (mkNormVect (intPt i) pos))
    (Texture _ reflCoef specCoef _ _) = intText i

shadePt :: Intersection -> Vector -> [TexturedObject]
         -> Light -> Color
shadePt i d o (AmbientLight int) = int
shadePt i d o l@(PointLight pos int)
  | s = (0.0,0.0,0.0)
  | otherwise = (diff (Just i) l) <+> (spec (Just i) d l)
  where
    s = not (isNothing i_s)
        && (intDist i_s) <= dist (intPt (Just i)) pos
    i_s = intersect (mkRay (intPt (Just i)) pos) o

reflectPt :: Int -> Intersection -> Vector -> [TexturedObject]
          -> [Light] -> Color
reflectPt depth i d = colorPt depth (Ray (intPt (Just i))
                                         (reflectDir d (normalAt (Just i))))
                          (0.0,0.0,0.0)

refractPt :: Int -> Intersection -> Vector -> Color
          -> [TexturedObject] -> [Light] -> Color
refractPt depth i d b = if refractedDir == (0.0,0.0,0.0)
                        then (\x y -> (0.0,0.0,0.0))
                        else colorPt depth (Ray (intPt (Just i))
                                                refractedDir) (b *> refrCoef)
  where
    refractedDir = refractDir d (normalAt (Just i)) refrIndex
    (Texture _ _ _ refrCoef refrIndex) = intText (Just i)

colorPt :: Int -> Ray -> Color -> [TexturedObject]
        -> [Light] -> Color
colorPt (-1) _ _ _ = (0.0, 0.0, 0.0)
colorPt d r@(Ray _ dir) b o l = if (isNothing i) then b
    else clip $ shadeColor <+> reflectColor <+> refractColor
  where
    shadeColor = foldl (<+>) (0.0,0.0,0.0)
                (map (shadePt (fromJust i) dir o) l)

```

```

reflectColor = if (reflCoef == 0.0) then (0.0, 0.0, 0.0)
               else (reflectPt (d-1) (fromJust i) dir o l) *> reflCoef
refractColor = if (refrCoef == 0.0) then (0.0, 0.0, 0.0)
               else (refractPt (d-1) (fromJust i) dir b o l) *> refrCoef
i = intersect r o
(Texture _ reflCoef _ refrCoef _) = intText i

rayTracePt :: Int -> Scene -> Point3D -> Color
rayTracePt d (Scene (Camera eye _) b o l) p
  = colorPt d (Ray p (mkNormVect eye p)) b o l

rayTrace :: Int -> Resolution -> Scene -> Image
rayTrace d r s@(Scene (Camera _ dim) _ _ _)
  = (rayTracePt d s) . (mapToWin r dim)

```

### A.3 HRayPerlin

```

module HRayPerlin where

import HRayMath (Point3D, Vector, (<+>), (*>), clip)
import HRayEngine (Color)

noise :: Double -> Point3D -> [Int] -> [Color] -> Double
noise gridSize (xt,yt,zt) pList gList = res
  where
    (x,y,z) = (xt/gridSize,yt/gridSize,zt/gridSize)
    s_curve t = t*t*(3-2*t)
    lerp t a b = a + t*(b-a)
    (tx,ty,tz) = (x+256,y+256,z+256)
    (bx0,bx1,rx0,rx1) = (mod (floor tx) 256, mod (bx0+1) 256,
                        x - fromIntegral (floor x), rx0 - 1)
    (by0,by1,ry0,ry1) = (mod (floor ty) 256, mod (by0+1) 256,
                        y - fromIntegral (floor y), ry0 - 1)
    (bz0,bz1,rz0,rz1) = (mod (floor tz) 256, mod (bz0+1) 256,
                        z - fromIntegral (floor z), rz0 - 1)
    (i,j) = (pList !! bx0, pList !! bx1)
    (b00,b10,b01,b11) = (pList !! mod (i+by0) 256,
                        pList !! mod (j+by0) 256,
                        pList !! mod (i+by1) 256,
                        pList !! mod (j+by1) 256)
    (t,sy,sz) = (s_curve rx0, s_curve ry0, s_curve rz0)
    at3 (qx,qy,qz) (rx,ry,rz) = rx*qx + ry*qy + rz*qz
    u1 = at3 (gList !! mod (b00+bz0) 256) (rx0,ry0,rz0)
    v1 = at3 (gList !! mod (b10+bz0) 256) (rx1,ry0,rz0)

```

```

a1 = lerp t u1 v1
u2 = at3 (gList !! mod (b01+bz0) 256) (rx0,ry1,rz0)
v2 = at3 (gList !! mod (b11+bz0) 256) (rx1,ry1,rz0)
b1 = lerp t u2 v2
c = lerp sy a1 b1
u3 = at3 (gList !! mod (b00+bz1) 256) (rx0,ry0,rz1)
v3 = at3 (gList !! mod (b10+bz1) 256) (rx1,ry0,rz1)
a2 = lerp t u3 v3
u4 = at3 (gList !! mod (b01+bz1) 256) (rx0,ry1,rz1)
v4 = at3 (gList !! mod (b11+bz1) 256) (rx1,ry1,rz1)
b2 = lerp t u4 v4
d = lerp sy a2 b2
res = lerp sz c d

```

```

semiTurbulence :: Int -> Double -> Point3D -> Double
semiTurbulence 0 gridS intPt = noise gridS intPt getPList getGList
semiTurbulence freq gridS intPt
  = coef + (semiTurbulence (freq-1) gridS intPt)
  where
    coef = (1/(2^freq))*(noise gridS (intPt
      *> (2^freq)) getPList getGList)

```

```

turbulence :: Int -> Double -> Point3D -> Double
turbulence 0 gridS intPt = abs (noise gridS intPt getPList getGList)
turbulence freq gridS intPt
  = coef + (turbulence (freq-1) gridS intPt)
  where
    coef = (1/(2^freq))*(abs (noise gridS (intPt
      *> (2^freq)) getPList getGList))

```

```

perlinSolid :: Color -> Double -> Point3D -> Color
perlinSolid base gridS intPt
  = base *> (((noise gridS intPt getPList getGList)+1)/2)

```

```

perlinSemiTurbulence :: Color -> Int -> Double -> Point3D -> Color
perlinSemiTurbulence base freq gridS intPt
  = base *> (((semiTurbulence freq gridS intPt)+1)/2)

```

```

perlinTurbulence :: Color -> Int -> Double -> Point3D -> Color
perlinTurbulence base freq gridS intPt
  = base *> (turbulence freq gridS intPt)

```

```

perlinFire :: Color -> Color -> Int -> Double -> Point3D -> Color
perlinFire addColor base freq gridS intPt
  = clip (addColor <+> (perlinTurbulence base freq gridS intPt))

```

```

perlinPlasma :: Double -> Point3D -> Color
perlinPlasma gridS intPt = (noise1,noise2,noise3)
  where
    noise1 = (((noise gridS intPt getPList getGList) +1)/2)
    noise2 = (((noise gridS intPt getPList2 getGList) +1)/2)
    noise3 = (((noise gridS intPt getPList3 getGList) +1)/2)

perlinMarble :: Color -> Int -> Double -> (Double,Double,Double)
              -> Double -> Point3D -> Color
perlinMarble base freq gridS (xf,yf,zf) pow intPt@(x,y,z)
  = base *> (((sin (x*xf+y*yf+z*zf + pow*turb))+1)/2)
  where
    turb = turbulence freq gridS intPt

perlinMarbleBase :: Color -> Color -> Int -> Double
                  -> (Double,Double,Double) -> Double
                  -> Point3D -> Color
perlinMarbleBase base addColor freq gridS (xf,yf,zf) pow intPt
  = clip (addColor <+> (perlinMarble base freq
                       gridS (xf,yf,zf) pow intPt))

perlinWood :: Color -> Int -> Double -> Double -> Double
            -> Point3D -> Color
perlinWood base@(r,g,b) freq gridS xyFact pow intPt@(x,y,z)
  | (sinVal < 0.5) = clip (r+sinVal,g+sinVal,b+sinVal)
  | otherwise      = (r,g,b)
  where
    sinVal = 0.25 * (abs (sin (xyFact*(sqrt ((x*x+y*y+z*z))
                                         + pow*turb))))
    turb = turbulence freq gridS intPt

getPList :: [Int]
getPList = [156,75,107,...]

getPList2 :: [Int]
getPList2 = [68,7,60,...]

getPList3 :: [Int]
getPList3 = [222,172,15,...]

getGList :: [Vector]
getGList = [(-0.463400866,0.628613349,0.624583776),...]

}

```

## A.4 HRayOutput

```
module HRayOutput (getRenderTime, createPPM, getImage) where

import HRayEngine (Resolution, Color, Scene, rayTrace)

getRenderTime :: Integer -> Integer -> String
getRenderTime before after = (show min) ++ "m" ++ secStr ++ "s"
  where
    total = div (after - before) (1012)
    sec   = mod total 60
    secStr = if (sec < 10) then "0" ++ (show sec) else show sec
    min   = div total 60

getImage :: Int -> Resolution -> Scene -> [Color]
getImage d r@(rx,ry) s = [image (fromIntegral x,fromIntegral (-y))
                          | y <- [-(ry-1)..0], x <- [0..(rx-1)]]
  where
    image = rayTrace d r s

createPPM :: Resolution -> [Color] -> String
createPPM (w,h) colors = ("P3\n"++) . shows w . ( ' ' : ) . shows h
                      . ("\n255\n"++) . stringify colors $ ""
  where stringify = flip $ foldr showC
        showC (r,g,b) = shows (round (r*255)) . ( ' ' : )
                      . shows (round (g*255)) . ( ' ' : )
                      . shows (round (b*255)) . ( ' ' : )
```

## A.5 HRayParser

```
{
module HRayParser (RenderDescr(RenderDescr), readDescr) where

import HRayEngine (Scene(Scene), Texture(Texture), Diff(Solid,Perlin),
                  TexturedObject, Light(AmbientLight, PointLight),
                  Camera(Camera))
import HRayPerlin (perlinSolid, perlinSemiTurbulence,perlinTurbulence,
                  perlinFire,perlinPlasma,perlinMarble,
                  perlinMarbleBase,perlinWood)
import HRayMath (Dimension, Resolution, Object(Sphere,Plane))
import Char (isSpace, isAlpha, isDigit)

data RenderDescr = RenderDescr Resolution Int Scene
```

```

readDescr :: String -> RenderDescr
readDescr = parseScene.lexer
}

%name parseScene
%tokentype { Token }

%token
    int           { TokenInt $$ }
    double        { TokenDouble $$ }
    camera        { TokenCamera }
    background    { TokenBackground }
    diff          { TokenDiff }
    solid         { TokenSolid }
    perlinSolid   { TokenPerlinSolid }
    perlinSemiTurb { TokenPerlinSemiTurb }
    perlinTurb    { TokenPerlinTurb }
    perlinFire    { TokenPerlinFire }
    perlinPlasma  { TokenPerlinPlasma }
    perlinMarble  { TokenPerlinMarble }
    perlinMarbleBase { TokenPerlinMarbleBase }
    perlinWood    { TokenPerlinWood }
    perlin        { TokenPerlin }
    texture       { TokenTexture }
    sphere        { TokenSphere }
    plane         { TokenPlane }
    object        { TokenTexturedObject }
    objects       { TokenObjects }
    pointLight    { TokenPointLight }
    ambientLight  { TokenAmbientLight }
    lights        { TokenLights }
    scene         { TokenScene }
    resolution    { TokenResolution }
    renderDescr   { TokenRenderDescr }
    '{'          { TokenOpenAcc }
    '}'          { TokenCloseAcc }
    '('          { TokenOpenBrack }
    ')'          { TokenCloseBrack }
    '['          { TokenOpenHook }
    ']'          { TokenCloseHook }
    ','          { TokenComma }

%%

```

```

RenderDescr : renderDescr Resolution int '[' Scene ']'
              { RenderDescr $2 $3 $5}
Resolution  : '(' resolution '(' int ',' int ')')'
              { ($4,$6) }
Scene       : scene '{' Camera '}' '{' Background '}'
              '{' objects Objects '}' '{' lights Lights '}'
              { Scene $3 $6 $10 $14 }
Camera      : camera '(' double ',' double ',' double ')')
              '(' int ',' int ')')
              { Camera ($3,$5,$7) ($10,$12) }
Background  : background '(' double ',' double ',' double ')')
              { ($3,$5,$7) }
Object      : sphere double '(' double ',' double ',' double ')')
              { Sphere $2 ($4,$6,$8) }
            | plane '(' double ',' double ',' double ',' double ')')
              { Plane ($3,$5,$7,$9) }
NoiseF      : perlinSolid '(' double ',' double ',' double ')') double
              { perlinSolid ($3,$5,$7) $9 }
            | perlinSemiTurb '(' double ',' double ',' double ')')
              int double { perlinSemiTurbulence ($3,$5,$7) $9 $10 }
            | perlinTurb '(' double ',' double ',' double ')')
              int double { perlinTurbulence ($3,$5,$7) $9 $10 }
            | perlinFire '(' double ',' double ',' double ')')
              '(' double ',' double ',' double ')') int double
              { perlinFire ($3,$5,$7) ($10,$12,$14) $16 $17 }

            | perlinPlasma double { perlinPlasma $2 }
            | perlinMarble '(' double ',' double ',' double ')')
              int double '(' double ',' double ',' double ')') double
              { perlinMarble ($3,$5,$7) $9 $10 ($12,$14,$16) $18 }

            | perlinMarbleBase '(' double ',' double ',' double ')')
              '(' double ',' double ',' double ')') int double '('
              double ',' double ',' double ')') double
              { perlinMarbleBase ($3,$5,$7) ($10,$12,$14)
                  $16 $17 ($19,$21,$23) $25 }
            | perlinWood '(' double ',' double ',' double ')')
              int double double double
              { perlinWood ($3,$5,$7) $9 $10 $11 $12 }

Diff        : solid '(' double ',' double ',' double ')')
              { Solid ($3,$5,$7) }
            | perlin '(' NoiseF ')') { Perlin $3 }

```

```

Texture      : '(' texture '(' diff Diff ')' double int double
              double')'          { Texture $5 $7 $8 $9 $10 }

TexturedObject : '(' object '(' Object ')' Texture ')'
                { ($4,$6) }

Objects      : {- empty -}          { [] }
              | Objects TexturedObject { $2 : $1 }

Light        : '(' pointLight '(' double ',' double ',' double ')'
              '(' double ',' double ',' double ')' ')'
              { PointLight ($4,$6,$8) ($11,$13,$15) }
              | '(' ambientLight '(' double ',' double ','
              double ')' ')' { AmbientLight ($4,$6,$8) }

Lights       : {- empty -}          { [] }
              | Lights Light { $2 : $1 }

{

```

```

happyError :: [Token] -> a
happyError _ = error "Parse error ! ! !"

```

```

data Token
  = TokenInt Int
  | TokenDouble Double
  | TokenCamera
  | TokenBackground
  | TokenDiff
  | TokenSolid
  | TokenPerlinSolid
  | TokenPerlinSemiTurb
  | TokenPerlinTurb
  | TokenPerlinFire
  | TokenPerlinPlasma
  | TokenPerlinMarble
  | TokenPerlinMarbleBase
  | TokenPerlinWood
  | TokenPerlin
  | TokenTexture
  | TokenSphere
  | TokenPlane
  | TokenObjType
  | TokenTexturedObject
  | TokenObjects

```

```

    | TokenPointLight
    | TokenAmbientLight
    | TokenLight
    | TokenLights
    | TokenScene
    | TokenResolution
    | TokenRenderDescr
    | TokenOpenAcc
    | TokenCloseAcc
    | TokenOpenBrack
    | TokenCloseBrack
    | TokenOpenHook
    | TokenCloseHook
    | TokenComma

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
  | isSpace c = lexer cs
  | isAlpha c = lexVar (c:cs)
  | isDigit c = lexNum (c:cs) 1
lexer ('{':cs) = TokenOpenAcc : lexer cs
lexer ('}':cs) = TokenCloseAcc : lexer cs
lexer ('(':cs) = TokenOpenBrack : lexer cs
lexer (')':cs) = TokenCloseBrack : lexer cs
lexer ('[':cs) = TokenOpenHook : lexer cs
lexer (']':cs) = TokenCloseHook : lexer cs
lexer (',':cs) = TokenComma : lexer cs
lexer ('-':cs) = lexNum cs (-1)

lexNum cs mul
  | (r == '.') = TokenDouble (mul * (read (num++[r]++num2)
    :: Double)) : lexer rest2
  | otherwise = TokenInt (round (mul * (read num)))
    : lexer (r:rest)
  where (num,(r:rest)) = span isDigit cs
        (num2,rest2) = span isDigit rest

lexVar cs =
  case span isAlpha cs of
    ("camera",rest)      -> TokenCamera : lexer rest
    ("background",rest)  -> TokenBackground : lexer rest
    ("diff",rest)        -> TokenDiff : lexer rest
    ("solid",rest)       -> TokenSolid : lexer rest
    ("perlinSolid",rest) -> TokenPerlinSolid : lexer rest

```

```

("perlinSemiTurb",rest)    -> TokenPerlinSemiTurb : lexer rest
("perlinTurb",rest)       -> TokenPerlinTurb : lexer rest
("perlinFire",rest)       -> TokenPerlinFire : lexer rest
("perlinPlasma",rest)     -> TokenPerlinPlasma : lexer rest
("perlinMarble",rest)     -> TokenPerlinMarble : lexer rest
("perlinMarbleBase",rest) -> TokenPerlinMarbleBase : lexer rest
("perlinWood",rest)       -> TokenPerlinWood : lexer rest
("perlin",rest)           -> TokenPerlin : lexer rest
("texture",rest)          -> TokenTexture : lexer rest
("sphere",rest)           -> TokenSphere : lexer rest
("plane",rest)            -> TokenPlane : lexer rest
("object",rest)           -> TokenTexturedObject : lexer rest
("objects",rest)          -> TokenObjects : lexer rest
("pointLight",rest)       -> TokenPointLight : lexer rest
("ambientLight",rest)     -> TokenAmbientLight : lexer rest
("light",rest)            -> TokenLight : lexer rest
("lights",rest)           -> TokenLights : lexer rest
("scene",rest)            -> TokenScene : lexer rest
("resolution",rest)       -> TokenResolution : lexer rest
("renderDescr",rest)      -> TokenRenderDescr : lexer rest
}

```

## A.6 HRayMain

```

import HRayOutput (getRenderTime, createPPM, getImage)
import HRayParser (RenderDescr(RenderDescr), readDescr)
import Directory (doesFileExist)
import System (getArgs)
import System.CPUTime (getCPUTime)

main :: IO()
main = do args <- getArgs
  if (length args /= 2)
  then error "Usage: HRay <input scene path> <output ppm path>\n"
  else do
    let input = head args
        output = head (tail args)
        exists <- doesFileExist input
    case exists of
      True -> do
        fileContent <- readFile input
        let (RenderDescr res depth scene)
            = readDescr fileContent
            timeBefore <- getCPUTime

```

```

writeFile output (createPPM res
                  (getImage depth res scene))
timeAfter <- getCPUtime
putStr $ "Image output written to "
        ++ output ++ " in PPM format\n"
putStr $ "Time needed to render the image: "
        ++ (getRenderTime timeBefore timeAfter) ++ "\n"
False -> error $ "File \"\" ++ input ++ "\" does not exist.\n"

```

## A.7 HRayGUI

```

import Graphics.UI.Gtk hiding (Graphics.UI.Gtk.Gdk.Enums.Solid,
                               Graphics.UI.Gtk.General.Structs.Color)
import Data.IOREf (newIORef,readIORef,writeIORef)
import Data.Array (Array, (!), listArray)
import Maybe (isNothing)
import PixBufInternals (pixbufSetPixelsRGB8, imageSetFromPixbuf)
import HRayEngine (Scene(Scene))
import HRayOutput (getRenderTime, getImage)
import HRayParser (RenderDescr(RenderDescr), readDescr)
import Directory (doesFileExist)
import System.CPUTime (getCPUtime)

loadThisScene :: String -> IO (Maybe RenderDescr)
loadThisScene file = do
    exists <- doesFileExist file
    case exists of
        True -> do
            fileContent <- readFile file
            return (Just (readDescr fileContent))
        False -> return Nothing

loadScene :: String -> Label -> Label -> Label
            -> IO (Maybe RenderDescr)
loadScene "" messLabel descrLabel renderTimeLabel = do
    labelSetText messLabel "\nNo scene loaded.\n"
    labelSetText descrLabel "\nScene description:\n\n Resolution:
                             <none>\n Objects: <none>\n Lights: <none>\n"
    labelSetText renderTimeLabel "\nRender time:\n\n <none>\n"
    return Nothing
loadScene file messLabel descrLabel renderTimeLabel = do
    renderDescr <- loadThisScene file
    labelSetText renderTimeLabel "\nRender time:\n\n <none>\n"
    case renderDescr of

```

```

Nothing -> do
  labelSetText messLabel "\nNo such file.\n"
  labelSetText descrLabel "\nScene description:\n\n
    Resolution: <none>\n Objects: <none>\n Lights: <none>\n"
  (Just (RenderDescr (rx,ry) depth (Scene _ _ objects lights)) )
-> do
  labelSetText messLabel "\nScene loaded.\n"
  labelSetText descrLabel
    ("\nScene description:\n\n Resolution: "++(show rx)++"x"++
    (show ry)++"\n # Objects: "++(show (length objects))++
    "\n # Lights: "++ (show (length lights))++"\n")
return renderDescr

pixbufSetPixelsRGB8FromArray :: Int -> Pixbuf
                                -> Array Int (Int,Int,Int) -> IO()
pixbufSetPixelsRGB8FromArray w pixbuf arr
  = pixbufSetPixelsRGB8 pixbuf (\x y -> (\ (r,g,b) ->
    (fromIntegral r, fromIntegral g, fromIntegral b)) $ arr!(y*w+x) )

rendering a scene

renderScene :: (Maybe RenderDescr) -> Label -> IO Pixbuf
renderScene (Just (RenderDescr res@(rx,ry) depth scene))
  descrLabel = do
    let colorList = map (\ (r,g,b) -> (round (r*255),
      round (g*255),round (b*255))) $ getImage depth res scene
        colorArray = listArray (0,rx*ry-1) colorList
    pixBuf <- pixbufNew ColorspaceRgb False 8 rx ry
    pixbufSetPixelsRGB8FromArray rx pixBuf colorArray
    return pixBuf

deleteEvent :: Event -> IO Bool
deleteEvent _ = do return False

destroyEvent :: IO()
destroyEvent = do mainQuit

main :: IO()
main = do
  initGUI
  window <- windowNew
  onDelete window deleteEvent
  onDestroy window destroyEvent
  windowSetTitle window "HRay - a Haskell raytracer"

```

```

scrollWindow <- scrolledWindowNew Nothing Nothing
containerSetBorderWidth scrollWindow 10
image <- imageNewFromFile "default.png"
scrolledWindowAddWithViewport scrollWindow image
widgetSetSizeRequest scrollWindow 800 600
controlPanel <- vBoxNew False 2
containerSetBorderWidth controlPanel 10
entry <- entryNew
entrySetWidthChars entry 20
loadButton <- buttonNewWithLabel "Load scene"
messageLabel <- labelNew (Just "\nNo scene loaded.\n")
renderButton <- buttonNewWithLabel "Render scene"
descrLabel <- labelNew (Just "\nScene description:\n\n
    Resolution: <none>\n Objects: <none>\n Lights: <none>\n")
renderTimeLabel <- labelNew (Just "\nRender time:\n\n <none>\n")
renderDescrRef <- newIORef Nothing
onClicked loadButton $ do
    fileStr <- (entryGetText entry)
    renderDescr <- loadScene fileStr messageLabel
        descrLabel renderTimeLabel
    writeIORef renderDescrRef renderDescr
onClicked renderButton $ do
    renderDescr <- readIORef renderDescrRef
    let loop = do
            n <- eventsPending
            if n == 0 then return ()
                else mainIterationDo False >> loop
        if isNothing renderDescr
            then
                labelSetMarkupWithMnemonic messageLabel
                    "\n<b>Please load a scene.</b>\n"
            else do
                timeBefore <- getCPUTime
                labelSetMarkupWithMnemonic messageLabel
                    ("\n<span foreground=\"red\">"
                    ++ "<b>Rendering scene...</b></span>\n")
                loop
                pixBuf <- renderScene renderDescr descrLabel
                labelSetText messageLabel "\nScene rendered !\n"
                imageSetFromPixbuf image pixBuf
                timeAfter <- getCPUTime
                labelSetText renderTimeLabel $ "\nRender time:\n\n " ++
                    (getRenderTime timeBefore timeAfter) ++ "\n"
    boxPackStart controlPanel entry PackNatural 0
    boxPackStart controlPanel loadButton PackNatural 0

```

```

boxPackStart controlPanel messageLabel PackNatural 0
boxPackStart controlPanel renderButton PackNatural 0
boxPackStart controlPanel descrLabel PackNatural 0
boxPackStart controlPanel renderTimeLabel PackNatural 0
hBox <- hBoxNew False 2
boxPackStart hBox scrollWindow PackNatural 0
boxPackStart hBox controlPanel PackNatural 0
containerAdd window hBox
widgetShowAll window
mainGUI

```

## A.8 PixbufInternals

```

{-# OPTIONS -fffi -O #-}

module PixBufInternals (pixbufSetPixelsRGB8,imageSetFromPixbuf)
  where

import Graphics.UI.Gtk
import Graphics.UI.Gtk.Types (Pixbuf(Pixbuf), Image(Image))
import Foreign
import Foreign.C (CUChar)
import Data.Word (Word8)
import Monad (liftM)
import Array

{-# INLINE pixbufSetPixelsRGB8 #-}

pixbufSetPixelsRGB8 :: Pixbuf -> (Int -> Int
                                -> (Word8, Word8, Word8)) -> IO ()
pixbufSetPixelsRGB8 pixbuf setPixel = do
  rowStride <- pixbufGetRowstride pixbuf
  width <- pixbufGetWidth pixbuf
  height <- pixbufGetHeight pixbuf
  let loop ptr y | y == height = return ()
      | otherwise = do
          let rowLoop ptr x | x == width = return ()
              | otherwise =
                  case setPixel x y of
                    (red, green, blue) -> do
                      pokeByteOff ptr 0 red
                      pokeByteOff ptr 1 green
                      pokeByteOff ptr 2 blue
                      rowLoop (ptr `plusPtr` 3) (x+1)

```

```

        rowLoop ptr 0
        loop (ptr 'plusPtr' rowStride) (y+1)
pixelsPtr <- pixbufGetPixels pixbuf
loop pixelsPtr 0

pixbufGetPixels :: Pixbuf -> IO (Ptr ())
pixbufGetPixels pb = liftM castPtr $
  (\(Pixbuf arg1) -> withForeignPtr arg1 $ \argPtr1
    ->gdk_pixbuf_get_pixels argPtr1) pb

foreign import ccall unsafe " gdk_pixbuf_get_pixels"
  gdk_pixbuf_get_pixels :: ((Ptr Pixbuf) -> (IO (Ptr CUChar)))

imageSetFromPixbuf :: Image -> Pixbuf -> IO ()
imageSetFromPixbuf img pbuf =  (\(Image arg1) (Pixbuf arg2)
  -> withForeignPtr arg1 $ \argPtr1 ->withForeignPtr arg2
  $ \argPtr2 ->gtk_image_set_from_pixbuf argPtr1 argPtr2) img pbuf

foreign import ccall safe " gtk_image_set_from_pixbuf"
  gtk_image_set_from_pixbuf
  :: ((Ptr Image) -> ((Ptr Pixbuf) -> (IO ())))

```

# Bibliografie

- [1] R.T. Boute. “Formal logic for practical use: a calculational approach”. cursusnota’s Formele Semantiek en Formele Systemmodellen, Universiteit Gent, 2002
- [2] R.T. Boute. “Concrete Generic Functionals: Principles, Design and Applications”. *Generic Programming* 89-119, Kluwer, 2003
- [3] R.T. Boute. “Companion to B. Meyer’s introduction to the theory of programming languages”. cursusnota’s Formele Semantiek, Universiteit Gent, 2002
- [4] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000
- [5] R. Bird. *Introduction to Functional Programming using Haskell: second edition*. Prentice Hall, 1998
- [6] P. Hudak, J. Peterson en J. Fasel. “A Gentle Introduction to Haskell”, 2000 (<http://www.haskell.org/tutorial/>)
- [7] S.P. Jones. “A Short Introduction to Haskell”, 2001 (<http://www.haskell.org/aboutHaskell.html>)
- [8] A. Simon, D. Coutts en J. Petersen. “Gtk2Hs: A Haskell Graphical User Interface Library”. (<http://www.haskell.org/gtk2hs>)
- [9] M. Sage. “FranTk - A Declarative GUI System for Haskell”. *Proceedings ACM SIG-PLAN International Conference on Functional Programming*, 2000
- [10] M.Slater, A.Steed en Y.Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real-Time*. Pearson Education Limited, 2002
- [11] A. Glassner (ed). *An introduction to ray tracing*. Academic Press, 1989
- [12] N. Wilt. *Object-orientded ray tracing in C++*. John Wiley & Sons, Inc., 1994
- [13] “POVRay: The Persistence of Vision Raytracer”. (<http://www.povray.org>)
- [14] Pixar. “Renderman”. (<http://renderman.pixar.com>)

- [15] Blue Sky. “CGI Studio”.  
(<http://www.blueskystudios.com/content/company.php/>)
- [16] H. Verlinde. “Systematisch ontwerp van XML-hulpmiddelen in een functionele taal”.  
afstudeerwerk, Universiteit Gent, 2003
- [17] C. Elliott, S. Finne en O. de Moor. “Compiling Embedded Languages”. *Semantics, Applications and Implementation of Program Generation (SAIG)*, 2000
- [18] C. Elliott. “Functional Image Synthesis”. *Bridges*, 2001
- [19] K. Perlin. “An Image Synthesizer”. *Computer Graphics: Proceedings of ACM SIGGRAPH* 19(3) 287-296, 1985
- [20] K. Perlin en E. Hoffert. “Hypertexture”. *Computer Graphics: Proceedings of ACM SIGGRAPH* 23(3) 253-262, 1989
- [21] K. Perlin. “Improving Noise”. *Computer Graphics: Proceedings of ACM SIGGRAPH*, 2002
- [22] S. Marlow. “Happy: The Parser Generator for Haskell”.  
(<http://www.haskell.org/happy/>)
- [23] D. Roundy. “darcs: A revision control system”.  
(<http://www.darcs.net>)
- [24] R. Paterson. “Arrows: A General Interface to Computation”.  
(<http://www.haskell.org/arrows>)