

VOORWOORD

Deze cursusnota's horen bij het opleidingsonderdeel *Computer Graphics* uit de eerste bachelor Informatica. De betrachting van dit opleidingsonderdeel is om stapsgewijs te komen tot een werkende graphics engine voor het genereren van drie dimensionale (3D) beelden. Zo evolueren we startende vanuit twee dimensionale afbeeldingen, naar eenkleurige 3D-lijntekeningen, gekleurde 3D-lijntekeningen tot werkelijke 3D-afbeeldingen op basis van Z -buffering. Deze laatste voorzien we ook van verschillende types van belichting, schaduwen en texturen. De voorgestelde methodes worden ook steeds met de nodige zorg wiskundig onderbouwd. Ray casting en tracing wordt in deze tekst niet toegelicht.

Bij het schrijven van enkele niet onbelangrijke secties uit Hoofdstuk 2 en 3, werd dankbaar gebruik gemaakt van het boek getiteld *Computer Graphics for Java Programmers* van Leen Ammeraal. Voor de overige hoofdstukken werden verscheidene online bronnen gecombineerd met de nodige eigen inbreng. Een mooi voorbeeld van het type afbeeldingen dat je kan aanmaken met de engine die we hier bespreken vinden we terug in Figuur 1.

Benny Van Houdt
December 2009

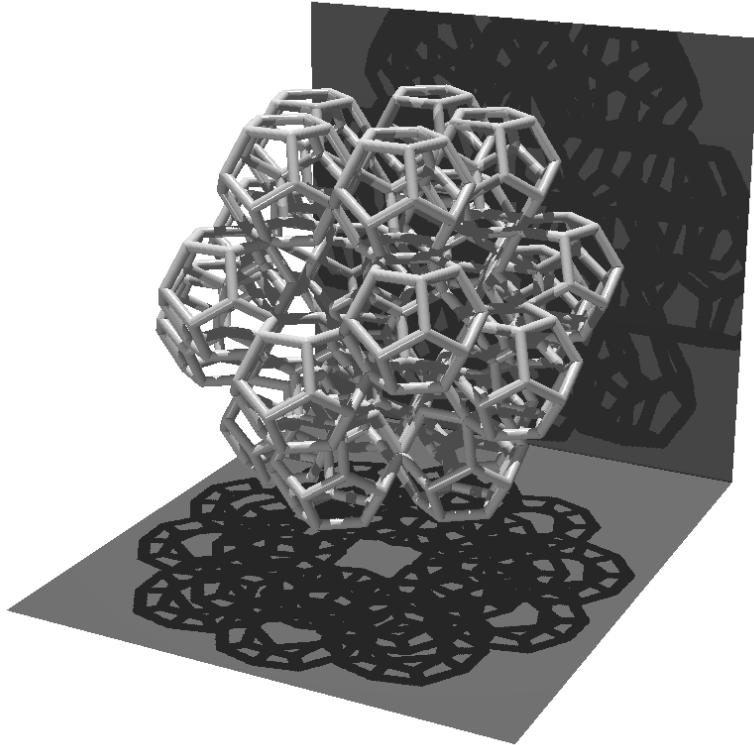


Fig. 1: 3D-graphics voorbeeld

Inhoudsopgave

2D-lijntekeningen en L-systemen	4
1 Hoe tekenen we een rechte lijn?	5
2 Specificatie van een lijntekening	7
3 Lindenmayer systemen (L-systemen)	9
4 L-systemen met brackets (haakjes)	13
3D-lijntekeningen en L-systemen	15
5 Transformaties	15
5.1 Translatie	15
5.2 Rotaties in 2D	15
5.3 Rotaties in 3D	17
5.4 Sequentie van transformaties	18
6 De perspectiefprojectie	19

6.1	Het <i>Eye</i> -coördinaatsysteem	19
6.2	De projectie zelf	22
7	De 3D-lijntekening maken	24
8	Platonische lichamen en de bol	25
9	3D L-systemen	29
Z-buffering en 3D-fractalen	34	
10	Z-buffering voor 3D-lijntekeningen met kleuren	34
11	$1/z$ -Interpolatie	36
12	Inkleuren van een driehoek	38
13	Z-buffering met driehoeken	40
14	Bepaling van dz/dx en dz/dy	43
15	3D-fractalen	45
Licht en schaduw	48	
16	Licht	48
16.1	Ambient licht	49
16.2	Diffuus licht	50
16.3	Specular licht	54
17	Schaduw	56
18	Texture mapping	62
18.1	Vlakke oppervlakken	63
18.2	Bolvormige oppervlakken	67
18.3	Willekeurige oppervlakken	68

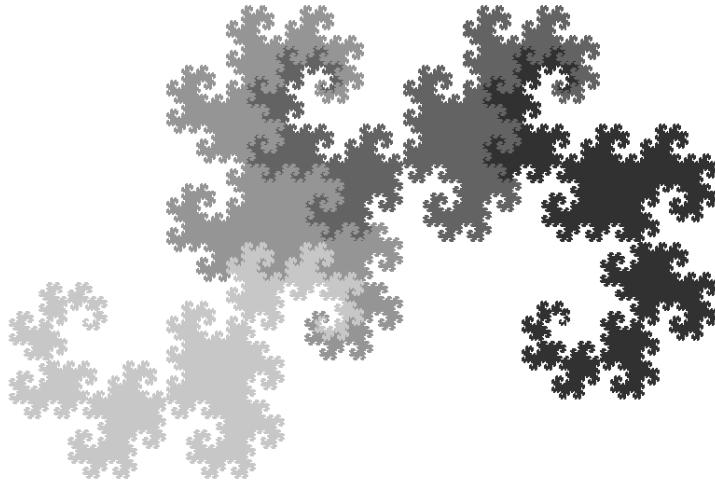


Fig. 2: Dragon Curve

2D-LIJNTEKENINGEN EN L-SYSTEMEN

In dit hoofdstuk zullen we starten met aan te geven hoe we de in te kleuren pixels bepalen wanneer we een lijn wensen te tekenen tussen twee opgegeven punten. Daarna gaan we met deze eenvoudige procedure een methode opzetten om afbeeldingen te maken die louter en alleen zijn opgebouwd uit lijnen in twee dimensies (2D). Tot slot komen we tot de hoofdbetragting van dit hoofdstuk, dewelke erin bestaat een programma te ontwikkelen dat 2D-lijntekeningen kan maken die worden gespecificeerd door een Lindenmayer systeem (L-systeem). Deze L-systemen werden in de eerste plaats ontwikkeld om grafische voorstellingen van planten te maken, maar laten eveneens tot tal van fractalen op een eenvoudige manier te specificeren en te genereren. Deze L-systemen kunnen ook verder worden uitgebreid naar stochastische L-systemen, drie dimensionale L-systemen, etc. Een voorbeeld afbeelding zien we in Figuur 2.

1 Hoe tekenen we een rechte lijn?

Stel dat we een afbeelding wensen te maken die is opgebouwd uit $Image_x$ rijen die elk $Image_y$ pixels bevatten. Kortom, de afbeelding wordt weergegeven door $Image_x Image_y$ pixels die elk een bepaalde kleur hebben. Momenteel is onze beschouwing om een lijn te tekenen van punt A met coördinaten (x_A, y_A) naar punt B met coördinaten (x_B, y_B) , deze coördinaten hoeven geen gehele getallen te zijn. We kiezen onze x - en y -as doorgaans zodat de onderkant van de afbeelding overeenstemt met de lijn $y = 0$, de bovenkant met $y = Image_y$, de linkerzijde met $x = 0$ en tot slot de rechterzijde met $x = Image_x$. Zonder verlies van algemeenheid mogen we stellen dat $x_A \leq x_B$, punt A ligt dus links van punt B (anders hernoemen we beide punten).

De startpixel wordt gegeven door $(\text{ceil}(x_A), \text{ceil}(y_A))$ en de eindpixel door $(\text{ceil}(x_B), \text{ceil}(y_B))$, waarbij de functie $\text{ceil}(x)$, x afrondt naar het kleinste geheel getal groter of gelijk aan x . Voor de eenvoud noteren we $\text{ceil}(x)$ als $x^{(r)}$. Een eerste vereiste is dat $x_A^{(r)}, x_B^{(r)} \in \{1, \dots, Image_x\}$ en dat $y_A^{(r)}, y_B^{(r)} \in \{1, \dots, Image_y\}$. Met andere woorden, elke pixel stemt dus overeen met een vierkantje met zijde één. Doorgaans zullen we het formaat van onze afbeelding automatisch laten bepalen zodat steeds aan deze voorwaarde voldaan wordt. Om de tussenliggende pixels te bepalen onderscheiden we 6 gevallen, waarvan de onderstaande twee de eenvoudigste zijn:

1. $x_A^{(r)} = x_B^{(r)}$: het betreft een verticale lijn en de in te kleuren pixels zijn $(x_A^{(r)}, y)$ met $y = [\min(y_A^{(r)}, y_B^{(r)}), \max(y_A^{(r)}, y_B^{(r)})]$.
2. $y_A^{(r)} = y_B^{(r)}$: het betreft een horizontale lijn en de in te kleuren pixels zijn $(x, y_A^{(r)})$ met $x = [x_A^{(r)}, x_B^{(r)}]$, daar $x_A \leq x_B$.

In de overige 4 gevallen definiëren we de richting van de rechte m als volgt

$$m = \frac{y_B^{(r)} - y_A^{(r)}}{x_B^{(r)} - x_A^{(r)}}.$$

Merk op dat zowel m als $1/m$ goed gedefinieerd zijn aangezien we beide bovenvermelde gevallen reeds apart afhandelde. De vier resterende gevallen worden weergegeven in Figuur 3. Wanneer de lijn het langst is in de x -richting, dan gebruiken we één pixel voor elke x -coördinaat, anders één voor elke y -coördinaat. Wanneer we een getal x afronden naar het dichtstbijgelegen gehele getal dan noteren we dit als $\text{round}(x)$.

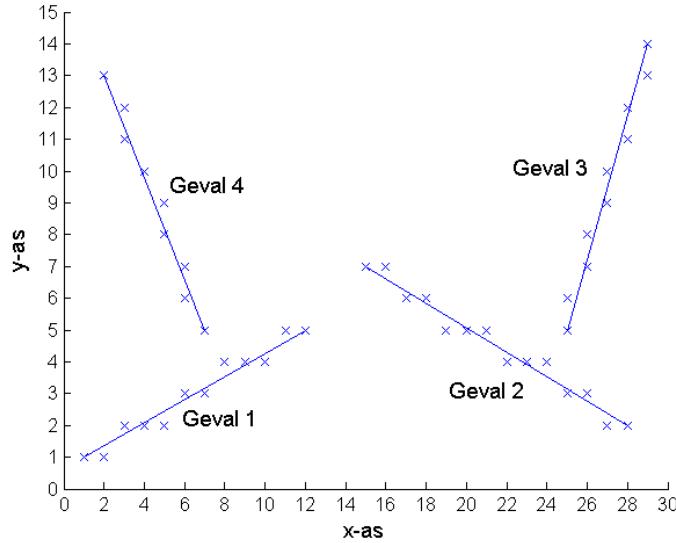


Fig. 3: 4 Gevallen bij het tekenen van een rechte lijn

1. $0 < m \leq 1$: het betreft een stijgende lijn die in de x -richting minstens even lang is als in de y -richting. De pixels worden gegeven door (x_i, y_i) met $x_i = x_A^{(r)} + i$ en $y_i = \text{round}(y_A^{(r)} + mi)$, voor $i = 0$ tot $x_B^{(r)} - x_A^{(r)}$.
2. $-1 \leq m < 0$: het betreft een dalende lijn die in de x -richting minstens even lang is als in de y -richting. De pixels worden eveneens gegeven door (x_i, y_i) met $x_i = x_A^{(r)} + i$ en $y_i = \text{round}(y_A^{(r)} + mi)$, voor $i = 0$ tot $x_B^{(r)} - x_A^{(r)}$.
3. $m > 1$: het betreft een stijgende lijn die in de y -richting langer is dan in de x -richting. De pixels worden gegeven door (x_i, y_i) met $y_i = y_A^{(r)} + i$ en $x_i = \text{round}(x_A^{(r)} + i/m)$, voor $i = 0$ tot $y_B^{(r)} - y_A^{(r)}$.
4. $m < -1$: het betreft een dalende lijn die in de y -richting langer is dan in de x -richting. De pixels worden gegeven door (x_i, y_i) met $y_i = y_A^{(r)} - i$ en $x_i = \text{round}(x_A^{(r)} - i/m)$, voor $i = 0$ tot $y_A^{(r)} - y_B^{(r)}$.

Om dit algoritme te begrijpen volstaat het vast te stellen dat in geval 1 en 2, de x -coördinaat met één verhogen, resulteert in een wijziging van de y -coördinaat met m . In geval 3 geeft een verhoging van y met één aanleiding

tot een verhoging van x met $1/m$, terwijl in het laatste geval x met $1/m$ daalt (of nog, met $1/|m|$ toeneemt), indien y met één daalt.

Het is ook mogelijk om de in te kleuren pixels te bepalen met behulp van een algoritme dat enkel gebruik maakt van gehele variabelen (Bresenham's algoritme), dit was enkele decennia geleden voornamelijk erg nuttig daar sommige chips daterende uit die periode nog geen floating point aritmetiek ondersteunde. Vandaag de dag biedt Bresenham's algoritme nog weinig voordeelen ten opzichte van bovenstaande floating point oplossing.

2 Specificatie van een lijntekening

Wanneer we een 2D-tekening wensen te maken die volledig is opgebouwd uit rechte lijnen, dan zullen we volgend input variabelen hanteren:

- *Points*: dit is een nP bij 2 array die voor elk eindpunt de reële x en y coördinaat bevat, vaak is het nuttig dat sommige punten meermaals voorkomen in deze lijst om zo de specificatie van de te tekenen lijnen te vergemakkelijken.
- *Lines*: dit is een nL bij 2 array die voor elk van de m lijnen waaruit de afbeelding bestaat, de gehele index van de twee eindpunten weergeeft.
- *Colors*: dit is een array die de kleur van elk van de m lijnen bevat (bv. RGB specificatie), het exacte formaat is momenteel onbelangrijk.

Het is ook nuttig er voor te zorgen dat de grootte van de afbeelding automatisch wordt ingesteld op basis van deze input gegevens, zodat we *size* pixels gebruiken voor de richting die met meeste pixels vereist. Dit kunnen we als volgt realiseren. Eerst berekenen we de maximale en minimale x en y coördinaat van alle punten en noteren die als x_{min} , x_{max} , y_{min} en y_{max} . Vervolgens berekenen we de ranges $x_{range} = x_{max} - x_{min}$ en $y_{range} = y_{max} - y_{min}$. We stellen $Image_x$ en $Image_y$ dan gelijk aan

$$Image_x = size \frac{x_{range}}{\max(x_{range}, y_{range})}, \quad Image_y = size \frac{y_{range}}{\max(x_{range}, y_{range})}.$$

Vervolgens stellen we $d = 0.95 \frac{Image_x}{x_{range}}$ en vermenigvuldigen alle getallen in de array *Points* met d (we creëren een kleine speelruimte door de 0.95 te hanteren). Hierdoor krijgt de input data de juiste grootte, we moeten

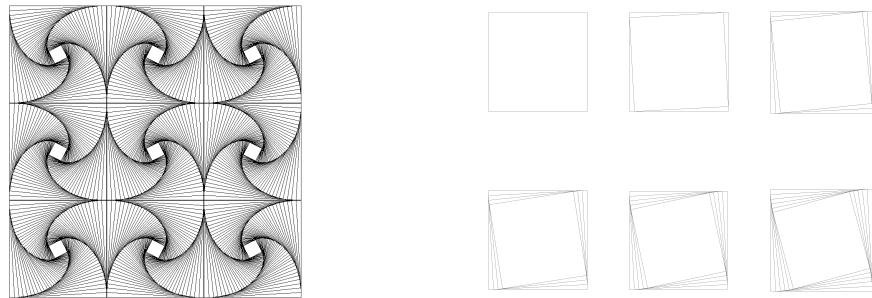


Fig. 4: Constructie van een spiraalvormige afbeelding

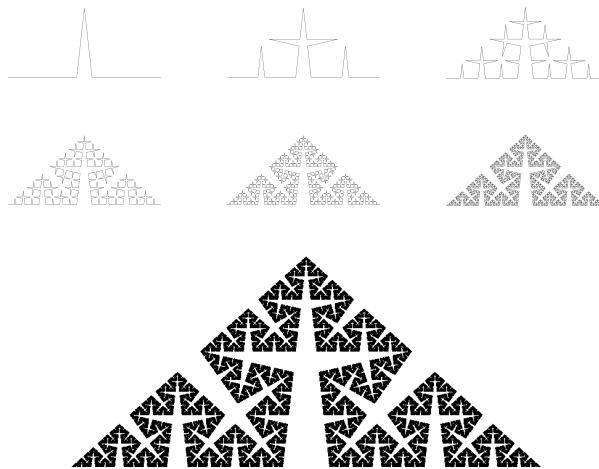


Fig. 5: Constructie van een Koch curve

enkel de data nog verschuiven zodat het centrum van de data DC , met coördinaten $DC_x = d(x_{min} + x_{max})/2$ en $DC_y = d(y_{min} + y_{max})/2$ samenvalt met het middelpunt van de afbeelding. Dit kunnen we eenvoudig door bij alle x -coördinaten $Image_x/2 - DC_x$ op te tellen en bij alle y -coördinaten $Image_y/2 - DC_y$, waardoor de hele afbeelding verschuift.

Een voorbeeld vinden we terug in Figuur 4, dewelke is opgebouwd uit negen spiraalvormige afbeeldingen. Elk van deze afbeeldingen werd opgebouwd door 40 vierkanten te construeren (zoals rechts is aangegeven). Kortom, deze tekeningen bestaat uit 9 maal 160 lijnen. Figuur 5 geeft een Koch curve weer die is opgebouwd uit 4^{10} lijnen, de eerste 6 stappen van de constructie zijn eveneens weergegeven. Deze Koch curve is een voorbeeld van een fractaal die wordt gegenereerd door telkenmale elke lijn te vervangen door een aan-

tal andere lijnen die een bepaald patroon vormen (in ons voorbeeld wordt het patroon gevormd door de vier lijnen die links bovenaan worden weergegeven). Het genereren van de verzameling *Points* en *Lines* voor tal van fractale afbeeldingen brengt dan ook heel wat werk met zich mee. Om dit te vermijden zullen we L-systemen invoeren. Deze systemen zullen toelaten om afbeeldingen van dit type heel compact te beschrijven zodat de bijhorende set van *Points* en *Lines* automatisch gegenereerd kan worden met behulp van deze compacte specificatie. Naast het snel genereren van de meer traditionele fractalen kunnen we hiermee eveneens 2D-afbeeldingen creëren van planten.

3 Lindenmayer systemen (L-systemen)

Een L-systeem bestaat uit een vijftal componenten:

1. Alphabet \mathcal{A} : dit is een lijst van symbolen gebruikt door het L-systeem. Bijvoorbeeld,

$$\mathcal{A} = \{F, f\}.$$

Het plus en min teken behoort niet tot \mathcal{A} .

2. Functie *Draw* die elk symbool in \mathcal{A} afbeeldt op 0 of 1. We noteren de set van symbolen die afgebeeld worden op 0 als \mathcal{A}_0 en de overige als \mathcal{A}_1 . Bijvoorbeeld, indien $Draw(F) = 1$ en $Draw(f) = 0$, dan is $\mathcal{A}_1 = \{F\}$ en $\mathcal{A}_0 = \{f\}$.
3. Initiator I : dit is een string die bestaat uit symbolen die tot $\mathcal{A} \cup \{+, -\}$ behoren. Bijvoorbeeld,

$$I = F + F + F + F.$$

4. Angles δ en α_0 : dit zijn twee hoeken tussen 0 en 2π (radialen) waarvan de betekenis later duidelijk zal worden.
5. Een set \mathcal{R} van replacement rules: voor elk symbool is er één replacement rule die het symbool afbeeldt op een string die bestaat uit symbolen die behoren tot $\mathcal{A} \cup \{+, -\}$. Bijvoorbeeld,

$$F \rightarrow F + f - FF + F + FF + Ff + FF - f + FF - F - FF - Ff - FFF$$

en

$$f \rightarrow fffff.$$

Een symbool X met als replacement rule $X \rightarrow X$ wordt een constante genoemd.

Om aan te geven hoe we een L-systeem kunnen hanteren om een complexe afbeelding bijna automatisch te genereren, zullen we met elke string $S = s_1s_2\dots s_k$ met $s_i \in \mathcal{A} \cup \{+, -\}$ voor $i = 1, \dots, k$ en $k \geq 1$, als volgt een afbeelding associëren:

1. We starten in een willekeurige positie, typisch positie $(x, y) = (0, 0)$, en in de richting $\alpha = \alpha_0$ radialen.
2. We overlopen de string $S = s_1s_2\dots s_k$ van links naar rechts en voeren afhankelijk van het huidige symbool s_i volgende actie uit:
 - $s_i \in \mathcal{A}_1$: we tekenen een lijn van lengte 1 in de huidige richting α waardoor onze huidige positie (x, y) wijzigt in $(x + \cos \alpha, y + \sin \alpha)$. De huidige richting α blijft identiek.
 - $s_i \in \mathcal{A}_0$: we verplaatsen onze pen over een lengte van 1 in de huidige richting, zoals in het voorgaande geval, maar dit keer zonder een lijn te tekenen. De huidige richting α blijft wederom identiek.
 - $s_i = +$: we verhogen onze huidige richting α met δ (modulo 2π). Kortom we draaien naar *links* over een hoek van δ radialen.
 - $s_i = -$: we verlagen onze huidige richting α met δ (modulo 2π). Kortom we draaien naar *rechts* over een hoek van δ radialen.

Bijvoorbeeld, de Initiator $I = F + F + F + F$ geeft een vierkant weer met de linkerbenedenhoek in $(0, 0)$ en de rechterbovenhoek in $(1, 1)$. Deze manier van het grafisch interpreteren van strings wordt vaak aangeduid met de benaming *turtle graphics* waarbij de huidige positie overeenstemt met de positie van een schildpad die voortbeweegt volgens de instructies van de string S .

De manier waarop een L-systeem een complexe afbeelding genereert, bestaat er dus gewoon in om de overeenkomstige string S op te stellen. Deze kan dan snel worden vertaald in een array van *Points* en *Lines*, die we dan met onze lijntekening procedure kunnen omzetten in een afbeelding. Het creëren van de string S zal gebeuren door te starten met de Initiator string

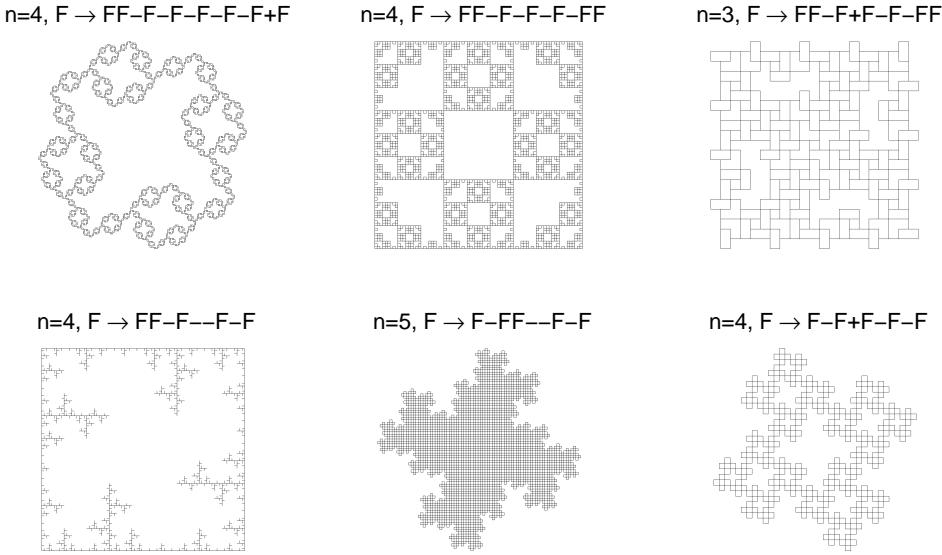


Fig. 6: Enkele fractalen die met behulp van een L-systeem met één symbool F kunnen worden aangemaakt, $I = F - F - F - F$, $\delta = \pi/2$ en $\alpha_0 = 0$

I. Vervolgens gaan telkens alle symbolen s_i in I vervangen door de string waarop s_i wordt afgebeeld door zijn overeenkomstige replacement rule. De + en - symbolen blijven gewoon staan¹. Hierdoor bekomen we de string S_1 . Vervolgens vervangen we analoog alle symbolen in S_1 om S_2 te bekomen en dit herhalen we n maal, zodat $S = S_n$. Laten we dit eens in meer detail bekijken voor de Koch curve uit Figuur 5. Voor deze figuur kunnen we volgend L-systeem hanteren:

- $\mathcal{A} = \mathcal{A}_1 = \{F\}$,
- $\delta = 2\pi(85/360)$, d.i., 85 graden of 1.4835 radialen, en $\alpha_0 = 0$,
- $I = F$ en $F \rightarrow F + F --F + F$.

We starten dus met een lijn van lengte 1, deze lijn F wordt tijdens de eerste iteratie vervangen door $F + F --F + F$, wat overeenstemt met de afbeelding links bovenaan in Figuur 5, die is opgebouwd uit 4 lijnen. Tijdens de tweede iteratie wordt elk van deze 4 lijnen op zijn beurt vervangen door de string

¹ Je zou ook kunnen stellen dat zij de overeenkomstige regels $+ \rightarrow +$ en $- \rightarrow -$ hebben.

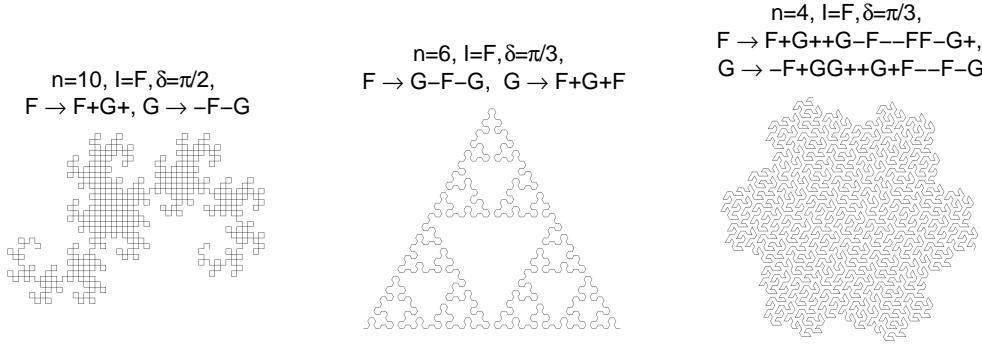


Fig. 7: Enkele fractalen die met behulp van een L-systeem met 2 symbolen kunnen worden aangemaakt met $\mathcal{A}_1 = \{F, G\}$ en $\alpha_0 = 0$

$F + F -- F + F$ wat resulteert in

$$\overbrace{F + F -- F + F}^F + \overbrace{F + F -- F + F}^F -- \overbrace{F + F -- F + F}^F + \overbrace{F + F -- F + F}^F.$$

Deze string S_2 stemt overeen met de tweede afbeelding in Figuur 5. Indien we $n = 10$, dan bekomen we de Koch curve die onderaan is weergegeven. In principe stemt de Koch curve overeen met het geval $n = +\infty$, maar doordat we met een beperkt aantal pixels werken (in dit geval 800 in de breedte), volstaat een eindige n om een goed beeld te krijgen van de werkelijke fractaal met $n = +\infty$.

Uit bovenstaand voorbeeld zou men de indruk kunnen krijgen dat een L-systeem steeds start met een aantal lijnen en vervolgens elke lijn vervangt door een patroon waarvan het begin- en eindpunt samenvalt met het begin- en eindpunt van de lijn die wordt vervangen. L-systeem zijn echter een stuk algemener. Dit blijkt bijvoorbeeld uit voorbeeld 1, 3, 5 en 6 in Figuur 6, die elk door een L-systeem met een enkel symbool worden gegenereerd. Bij de twee meest rechtse afbeeldingen (3 en 6) is zelfs de richting op het einde van het patroon tegengesteld aan de richting in het begin van het patroon (want $-$, $+$, $-$, $-$ resulteert in een draai van π radialen). Verder kunnen twee lijnen samenvallen, die dan elk door hun eigen patroon worden vervangen, wat bijvoorbeeld het geval is in voorbeeld 2 en 4. Bij voorbeeld 4 is de richting van de samenvallende lijnen niet hetzelfde waardoor de twee patronen niet samenvallen, dit in tegenstelling tot voorbeeld 2.

In Figuur 7 zien we enkele voorbeelden van L-systeem met twee symbolen. Wanneer we in de meest linkse figuur n verder verhogen tot 18 en voor

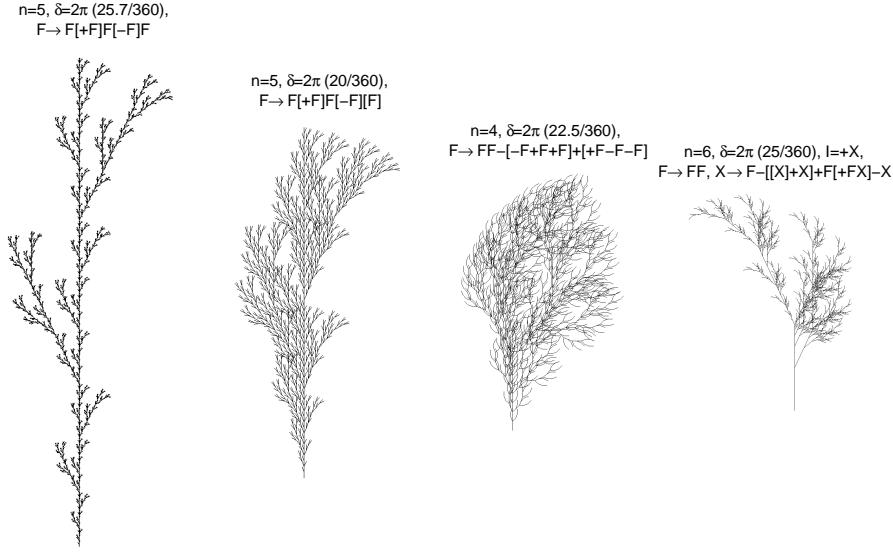


Fig. 8: Enkele 2D planten gegenereerd door middel van L-systemen met brackets, met $I = F$ of $I = X$, $\mathcal{A}_1 = \{F\}$ en $\alpha_0 = \pi/2$

de 2^{18} lijnen vier verschillende kleuren hanteren (kleur 1 voor lijn 1 tot 2^{16} , enz.), dan verkrijgen we de afbeelding in Figuur 2.

4 L-systemen met brackets (haakjes)

Er bestaan tal van veralgemeningen van L-systemen. Een eerste erg belangrijke is dat we naast de bijzondere symbolen $+$ en $-$ ook de vierkante haakjes $[$ en $]$ kunnen invoeren (dewelke ook niet mogen behoren tot \mathcal{A}). Het effect van deze haakjes is als volgt: wanneer we een string overlopen en het huidige symbool $s_i = [$ dan slaan we de huidige positie (x, y) en de huidige richting α op. Wanneer we later het overeenkomstige sluitende haakje tegenkomen dan vragen we deze opgeslagen positie en richting op en gaan verder vanuit (x, y) in de richting α . Aangezien we meerdere haakjes van de vorm $[$ kunnen tegenkomen voor we het overeenkomstige sluitende haakje aantreffen, gaan we dit implementeren door gebruik te maken van een stack. Telkens we een openend haakje $[$ tegenkomen pushen we de huidige positie en richting op de stack, terwijl een sluitend haakje $]$ een pop operatie veroorzaakt.

In Figuur 8 zien we drie voorbeelden van L-systemen met haakjes (brac-

kets). Deze afbeeldingen geven aan dat L-systemen kunnen gebruikt worden om de fractale structuur van planten weer te geven. Zoals eerder vermeld werden L-systemen precies ook om deze reden ontwikkeld.

Een andere nuttige uitbreiding is om voor sommige symbolen meerdere replacement rules te voorzien en met elke regel een kans te associëren (zodat de som van de kansen gelijk is aan één per symbool). Telkens we een symbool moeten vervangen kiezen we met behulp van deze kansen dan de replacement rule. Hierdoor zal één en hetzelfde L-systeem met bijvoorbeeld $n = 5$ aanleiding kunnen geven tot vele (licht) verschillende afbeeldingen. Dit kan erg nuttig zijn wanneer we een tuin moeten weergeven die gevuld is met een honderd tal planten. We wensen namelijk te vermijden dat we honderd dezelfde planten hebben in onze tuin en willen eveneens geen honderd verschillende L-systemen opzetten. Een stochastisch L-systeem zoals hierboven beschreven biedt dan een oplossing doordat dit ons de mogelijkheid geeft vele gelijkaardige planten te genereren met behulp van één L-systeem.

3D-LIJNTEKENINGEN EN L-SYSTEMEN

In dit hoofdstuk zullen we ons opnieuw toeleggen op het maken van lijntekeningen, echter dit maal doen we dit in drie dimensies. Er is nog één belangrijke beperking die we ons voorlopig opleggen: we zullen alle lijnen in dezelfde kleur tekenen, waardoor het volstaat dat we de perspectiefprojectie van elke lijn berekenen zonder dat we moeten nagaan welke lijn de voorste is indien twee of meer lijnen elkaar kruisen (de betreffende pixel kan gewoon ingekleurd worden). Om aan te geven hoe we een perspectiefprojectie kunnen maken, moeten we eerst enkele elementaire transformaties introduceren, met behulp van deze transformaties zal blijken dat het maken van de projectie betrekkelijk eenvoudig is.

5 Transformaties

5.1 Translatie

De eenvoudigste transformatie is de zogenaamde translatie. Deze zal alle punten verschuiven over een vaste afstand in een bepaalde richting. Een translatie over (a, b, c) zal de coördinaten van een willekeurig punt (x_0, y_0, z_0) wijzigen in (x_1, y_1, z_1) zodat

$$x_1 = x_0 + a, \quad y_1 = y_0 + b, \quad z_1 = z_0 + c.$$

Hiermee verplaatsen we dus eigenlijk de oorspong van ons assenstelsel naar het punt $(-a, -b, -c)$.

5.2 Rotaties in 2D

Voor we een rotatie in drie dimensies aanschouwen starten we met eens te kijken naar het twee dimensionale geval. Stel dat we de x - en y -as roteren over een hoek ϕ . Dan moeten we ons de vraag stellen wat de nieuwe coördinaten (x_1, y_1) van een punt het punt (x_0, y_0) zijn na het roteren van de assen. In Figuur 9 kunnen we het antwoord hierop mits een kleine inspanning aflezen.

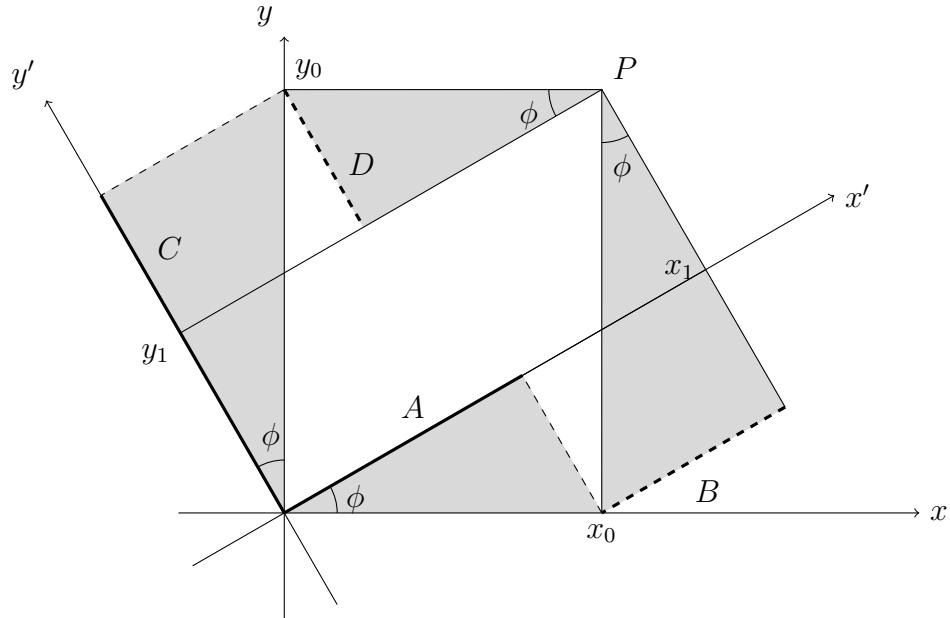


Fig. 9: Nieuwe coördinaten na rotatie over een hoek ϕ

Laten we starten met x_1 : de waarde hiervan is gelijk aan de lengte van het vetgedrukte lijnstuk A plus de lengte van het gearceerde lijnstuk B . We merken tevens volgende zaken op:

- De lengte van lijnstuk A is duidelijk $x_0 \cos \phi$ daar het de aanliggende zijde betreft van een rechthoekige driehoek waarvan de schuine zijde lengte x_0 heeft.
- De lengte van lijnstuk B is gelijk is aan $y_0 \sin \phi$ daar het de overstaande zijde betreft van een rechthoekige driehoek waarvan de schuine zijde lengte y_0 heeft.
- Bijgevolg is x_1 , de nieuwe x -coördinaat, gelijk aan $x_0 \cos \phi + y_0 \sin \phi$.

De nieuwe y -coördinaat y_1 kunnen we berekenen als de lengte van het vetgedrukte lijnstuk C min de lengte van het gearceerde lijnstuk D . Door wederom gebruik te maken van de rechthoekige driehoeken zien we dat

- De lengte van lijnstuk C is $y_0 \cos \phi$ daar het de aanliggende zijde betreft van een rechthoekige driehoek waarvan de schuine zijde lengte y_0 heeft.

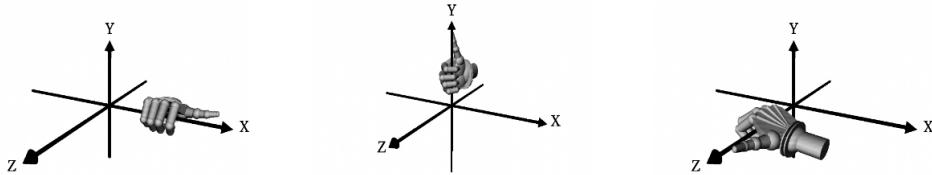


Fig. 10: Rotatierichting van rechtshandig coördinaatsysteem

- De lengte van lijnstuk D is gelijk aan $x_0 \sin \phi$ aangezien de schuine zijde nu lengte x_0 heeft en het de overstaande hoek betreft.
- Bijgevolg is y_1 , de nieuwe y -coördinaat, gelijk aan $-x_0 \sin \phi + y_0 \cos \phi$.

Samengevat kunnen we de nieuwe coördinaten van een willekeurig punt (x_0, y_0) door het roteren van de assen over ϕ radialen in matrixvorm noteren als

$$(x_1, y_1) = (x_0, y_0) \begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix}.$$

Het volstaat dus de coördinaten van het punt te vermenigvuldigen met bovenstaande 2×2 rotatiematrix.

5.3 Rotaties in 3D

Wanneer we een rotatie gaan uitvoeren in drie dimensies, dan zullen we ons (voorlopig) beperken tot een rotatie over de x -as, y -as of z -as. Wanneer we de x -as en y -as gekozen hebben, is het belangrijk even stil te staan bij de richting van de z -as, alsook de richting van de rotatie om elk van de assen. Deze zullen we kiezen volgens de rechterhandregel (zie Figuur 10). Een positieve rotatie stemt in dit geval overeen met het naar voor draaien van de vuist (wat tegengesteld is aan de beweging die we maken om *gas* te geven).

Een rotatie van ϕ radialen van de x - en y -as rond de z -as wijzigt de x en y coördinaat van een punt (x_0, y_0, z_0) dus exact op dezelfde wijze als in het twee dimensionale geval, terwijl de z -coördinaat ongewijzigd blijft. De rotatie rond de z -as kunnen we dus eveneens in matrixvorm noteren als

$$(x_1, y_1, z_1) = (x_0, y_0, z_0) \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

We noteren deze 3×3 rotatiematrix als $M_z(\phi)$. Belangrijk is op te merken dat in het twee dimensionale geval de x -as naar de oude y -as toe beweegt bij een positieve rotatie.

Wanneer we de y - en z -as rond de x -as gaan roteren dan merken we op dat y de rol speelt van de x -as in het twee dimensionale geval en z de rol van de y -as, daar de y -as naar de z -as toe beweegt. Verder blijft de x -coördinaat ongewijzigd, bijgevolg

$$(x_1, y_1, z_1) = (x_0, y_0, z_0) \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix}.$$

We noteren deze 3×3 rotatiematrix als $M_x(\phi)$.

Tot slot bekijken we de rotatie van de x - en z -as rond de y -as. Het is essentieel op te merken dat in dit geval de x -as van de z -as weg beweegt, in plaats van ernaar toe, wat impliceert

$$(x_1, y_1, z_1) = (x_0, y_0, z_0) \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix}.$$

We noteren deze 3×3 rotatiematrix als $M_y(\phi)$.

5.4 Sequentie van transformaties

In een aantal gevallen zullen we een reeks van transformaties na elkaar uitvoeren. Wanneer dit een aantal rotaties betreft, kunnen we het resultaat van deze sequentie nog steeds eenvoudig uitdrukken met een enkele 3×3 matrix. Bijvoorbeeld, indien we eerst de x - en y -as θ radialen roteren om de z -as en vervolgens de y - en z -as ϕ radialen om de x -as, dan zijn de coördinaten van het punt (x_0, y_0, z_0) na de eerste transformatie gelijk aan $(x_1, y_1, z_1) = (x_0, y_0, z_0)M_z(\theta)$ en na de tweede hebben we $(x_2, y_2, z_2) = (x_1, y_1, z_1)M_x(\phi)$. Met andere woorden, $(x_2, y_2, z_2) = (x_0, y_0, z_0)M_z(\theta)M_x(\phi)$. De 3×3 matrix

$$M_z(\theta)M_x(\phi) = \begin{bmatrix} \cos \theta & -\sin \theta \cos \phi & \sin \theta \sin \phi \\ \sin \theta & \cos \theta \cos \phi & -\cos \theta \sin \phi \\ 0 & \sin \phi & \cos \phi \end{bmatrix}.$$

Het volstaat dus het product van elk van de rotatiematrixen te nemen. Wanneer de sequentie echter ook een translatie bevat is het niet meer mogelijk

om de sequentie te karakteriseren met een 3×3 matrix. Dit kan wel door over te stappen op zogenaamde homogene coördinaten. In dit geval voegen we bij elk punt een extra coördinaat toe die gelijk is aan één: $(x_0, y_0, z_0, 1)$. De rotatiematrixen worden eveneens 4×4 matrices, bijvoorbeeld de rotatie beschreven door door de $M_z(\phi)$ matrix wordt nu gegeven door

$$(x_1, y_1, z_1, 1) = (x_0, y_0, z_0, 1) \begin{bmatrix} \cos \phi & -\sin \phi & 0 & 0 \\ \sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Nu kunnen we een translatie over (a, b, c) wel weergeven met een matrix aangezien

$$(x_1, y_1, z_1, 1) = (x_0 + a, y_0 + b, z_0 + c, 1) = (x_0, y_0, z_0, 1) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix}.$$

Indien we nu een sequentie hebben die bestaat uit een aantal translaties en rotaties, dan kunnen we deze complete transformatie dus compact voorstellen door het product te nemen van elk van de overeenkomstige 4×4 matrixen.

6 De perspectiefprojectie

In deze sectie geven we aan hoe we onze drie dimensionale input data omzetten naar een twee dimensionale projectie ervan. Onze input bestaat ditmaal uit een array *Points* met n rijen en 3 kolommen, terwijl elke rij van de array *Lines* zoals voorheen aangeeft tussen welke twee punten de lijn getrokken wordt.

De eerste stap bij het maken van de projectie bestaat erin de coördinaten van elk van de n eindpunten om te zetten in een ander coördinaatsysteem dat we zullen bekomen door drie opeenvolgende transformaties.

6.1 Het Eye-coördinaatsysteem

We starten de perspectiefprojectie met een wijziging van ons assenstelsel. Deze wijziging bestaat erin dat we een punt in de ruimte kiezen met coördinaten (x_E, y_E, z_E) dat we het *Eye* noemen en van waaruit we onze driedimensionale

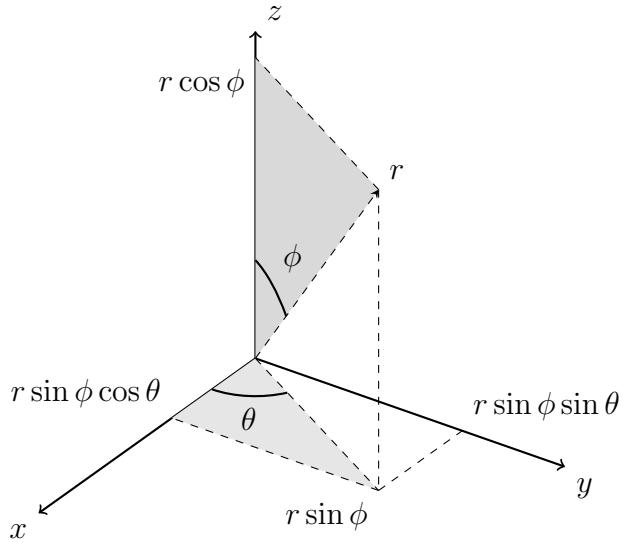


Fig. 11: Bolcoördinaten en hun relatie met cartesische coördinaten

wereld aanschouwen. De bedoeling is de oorsprong van het originele assenstelsel O te verleggen naar $O' = (x_E, y_E, z_E)$, de z -as te laten lopen van de oude oorsprong O naar O' , terwijl de x -as naar rechts wijst en de y -as naar boven.

Om dit te realiseren, gaan we de coördinaten van het *Eye* eerst omzetten in bolcoördinaten (d.i., sferische coördinaten). Deze zijn weergegeven in Figuur 11: r is de afstand van de oorsprong tot het punt, θ de hoek tussen de x -as en de projectie van het punt op het xy -vlak en ϕ de hoek tussen de z -as en het punt. Zoals eveneens is weergegeven in Figuur 11 geldt de volgende relatie tussen de (x, y, z) coördinaten en de bolcoördinaten (r, θ, ϕ) :

$$\begin{aligned} x &= (r \sin \phi) \cos \theta, \\ y &= (r \sin \phi) \sin \theta, \\ z &= r \cos \phi. \end{aligned}$$

We starten met de x - en y -as te roteren om de z -as zodat deze naar rechts wijst. Kijkend naar Figuur 11 en 12, zien we dat er een rotatie van $\theta + \pi/2$ radialen nodig is. Deze rotatie wordt dus beschreven door de matrix

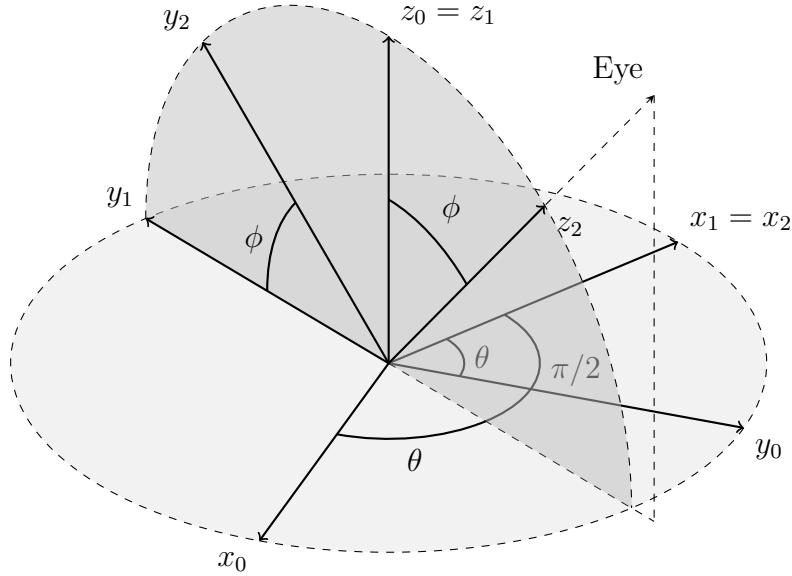


Fig. 12: Overgang naar het Eye-coördinaatsysteem: rotatie over $\theta + \pi/2$ radialen om de z -as gevuld door een rotatie over ϕ radialen om de x -as

$M_z(\theta + \pi/2)$:

$$\begin{bmatrix} \cos(\theta + \pi/2) & -\sin(\theta + \pi/2) & 0 & 0 \\ \sin(\theta + \pi/2) & \cos(\theta + \pi/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\sin \theta & -\cos \theta & 0 & 0 \\ \cos \theta & -\sin \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Nu de x -as op zijn plaats ligt moeten we enkel de y - en z -as nog roteren om de x -as. Op Figuur 12 zien we dat z precies ϕ radialen moet draaien (want de x -as wijst nu naar rechts). De tweede transformatie wordt dus beschreven door $M_x(\phi)$.

We eindigen met een translate die de oorsprong O naar O' verplaatst, aangezien de coördinaten van O' in het geroteerde assenstelsel duidelijk gelijk

zijn aan $(0, 0, r)$ wordt deze translatie beschreven door

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -r & 1 \end{bmatrix}.$$

De totale transformatie wordt dus gegeven door $V = M_z(\pi/2 + \theta)M_x(\phi)T$ dewelke gelijk is aan

$$V = \begin{bmatrix} -\sin \theta & -\cos \theta \cos \phi & \cos \theta \sin \phi & 0 \\ \cos \theta & -\sin \theta \cos \phi & \sin \theta \sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & -r & 1 \end{bmatrix}.$$

De eerste stap in het realiseren van de perspectiefprojectie bestaat er dus in alle rijen in *Points* (die we even moeten aanvullen met een kolom met enen omwille van de homogene coördinaten) te vermenigvuldigen met V (het volstaat dit te doen met de eerste drie kolommen van V , daar de vierde coördinaat steeds gelijk is aan één).

Een niet onbelangrijke keuze is deze van r , de afstand van ons oog tot de te visualiseren wereld. Een veilige keuze bestaat er vaak in enkele malen de afstand te nemen tussen het grootste verschil in de x -, y - en z -coördinaat van de opgegeven data in *Points*, tenminste als de data zich min of meer in de buurt van de oorsprong O bevindt. Anders, kan het nuttig zijn om, nog voordat we de transformatie V uitvoeren, een translatie uit te voeren die hiervoor zorgt.

6.2 De projectie zelf

Nu we onze input data hebben uitgedrukt in het *Eye*-coördinaatsysteem, kunnen we eenvoudig de overeenkomstige pixel van elk punt p bepalen door middel van een perspectiefprojectie.

Deze bestaat erin dat we een projectievlak plaatsen evenwijdig met het xy -vlak van het nieuwe assenstelsel, op (kleine) afstand d van $(0, 0, 0)$, het betreft dus het vlak met vergelijking $z_E = -d$. Vervolgens kiezen we $(0, 0, 0)$ als het centrum van de projectie (COP) en trekken een lijn vanuit het punt P met coördinaten (x_E, y_E, z_E) naar $(0, 0, 0)$ (zie Figuur 13). De geprojecteerde coördinaten (x', y') bekomen we door de x - en y -coördinaat van het snijpunt van deze lijn met het projectievlak $z = -d$ te nemen.

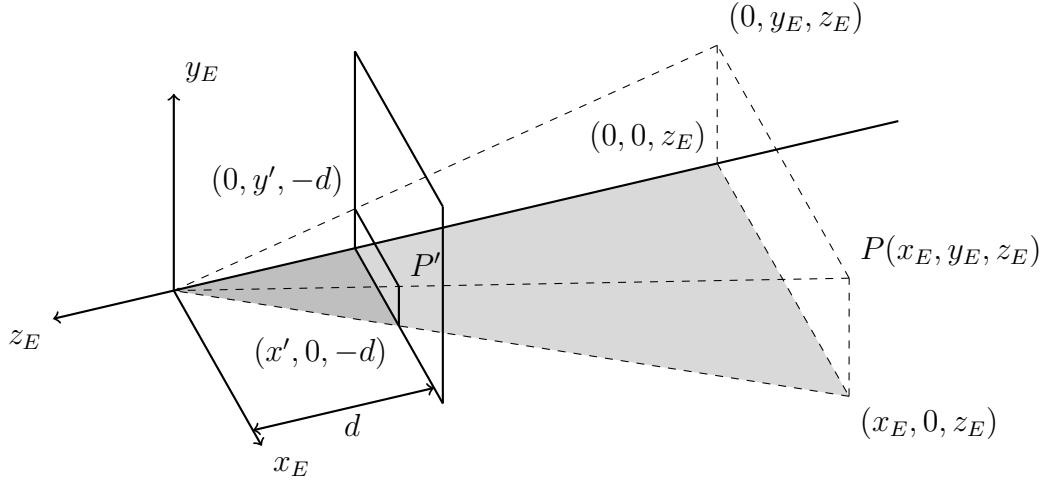


Fig. 13: Een klassieke perspectiefprojectie

In Figuur 13 geven we aan hoe we de coördinaten x' en y' van het geprojecteerde punt kunnen bepalen. We moeten hierbij opmerken dat de driehoek gevormd door de punten $(0, 0, 0)$, $(0, 0, z_E)$ en $(x_E, 0, z_E)$ dezelfde vorm heeft als de driehoek gevormd door $(0, 0, 0)$, $(0, 0, -d)$ en $(x', 0, -d)$ bijgevolg zijn volgende twee verhoudingen identiek (merk op, z_E is negatief door de richting van de z_E -as)

$$\frac{x_E}{-z_E} = \frac{x'}{d},$$

of nog $x' = \frac{dx_E}{-z_E}$. Analoog zijn de driehoeken gevormd door $(0, 0, 0)$, $(0, 0, -d)$ en $(0, y', -d)$ en $(0, 0, 0)$, $(0, 0, z_E)$ en $(0, y_E, z_E)$ gelijkvormig, waardoor

$$\frac{y_E}{-z_E} = \frac{y'}{d},$$

of nog $y' = \frac{dy_E}{-z_E}$. Met andere woorden, voor alle punten in de array *Points* moeten we, na de transformatie door de matrix V , enkel de x - en y -coördinaat delen door de z -coördinaat en het teken wijzigen. De vermenigvuldiging met d behandelen we in de volgende sectie, voorlopig stellen we $d = 1$.



Fig. 14: Wireframe voorstelling van een stoel

7 De 3D-lijntekening maken

Laten we even samenvatten hoe we onze 3D-lijntekening maken. Als input krijgen we een array *Points* met n rijen (het aantal eindpunten) en 3 kolommen, de x -, y - en z -coördinaat van onze input data. Verder hebben we een array met m rijen (het aantal lijnen) en 2 kolommen. Elke rij specificeert het nummer van beide eindpunten van de lijn.

Vervolgens kiezen we een punt van waaruit we onze afbeelding wensen te aanschouwen, dit doen we door de bolcoördinaten (r, θ, ϕ) vast te leggen, waarbij voornamelijk gelet moet worden dat r voldoende groot is.

Daarna voegen we een kolom enen aan de array *Points* toe en vermenigvuldigen deze $n \times 4$ array met de eerste drie kolommen van V . Hierdoor bekomen we een $n \times 3$ array die de coördinaten van onze punten bevat in het Eye-coördinaatsysteem. Wanneer we nu de elementen in de eerste twee kolommen van *Points* delen door het element in de derde kolom en het teken wijzigen, dan hebben we de waarde van de geprojecteerde coördinaten.

Op dat ogenblik is ons probleem gereduceerd tot het maken van een 2D-tekening en kunnen we terugvallen op onze routine voor een 2D-lijntekening. Herinner dat deze routine de afbeelding automatisch schaalde (zie Sectie 2 in het voorgaande hoofdstuk) waardoor de keuze van de parameter d uit de voorgaande sectie alsnog wordt vastgelegd.

8 Platonische lichamen en de bol

Hoewel we in dit hoofdstuk enkel kijken naar lijntekeningen, kunnen we toch al beelden creëren van driedimensionale objecten door van deze objecten enkel hun ribben voor te stellen (men spreekt van een *wireframe* tekening). Een complex voorbeeld vinden we terug in Figuur 14. We merken op dat het object volledig transparant is.

In deze sectie gaan we de wireframe modellen van enkele elementaire driedimensionale objecten bespreken, de zogenaamde platonische lichamen. In het volgende hoofdstuk zullen we hiermee tal van 3D-fractalen genereren. Verder gaan we in dit hoofdstuk ook na hoe we een wireframe model van een bol kunnen opstellen door uit te gaan van een icosaëdron.

Kubus (Cube): We starten met het eenvoudigste platonische lichaam: de kubus. Deze wordt beschreven door 8 punten weergegeven in de array *Points*, de 12 ribben stellen we voor door de vier hoekpunten van elk van de zes vlakken te specifiëren in de array *Faces*. Deze laatste is eenvoudig om te zetten in een array *Lines* zoals vereist in de voorgaande secties. De notatie T geeft aan dat we de array getransponeerd weergeven:

$$\text{Points}^T = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \end{bmatrix}$$

en

$$\text{Faces}^T = \begin{bmatrix} 1 & 5 & 2 & 6 & 7 & 1 \\ 5 & 2 & 6 & 1 & 3 & 6 \\ 3 & 8 & 4 & 7 & 8 & 2 \\ 7 & 3 & 8 & 4 & 4 & 5 \end{bmatrix}$$

Het resultaat zien we in de linkse afbeelding van Figuur 15. Merk op, het middelpunt van de kubus is gelokaliseerd in $(0, 0, 0)$.

Tetrahedron: Dit platonische lichaam is een piramide met drie zijden en bestaat dus uit 4 punten en 4 vlakken:

$$\text{Points}^T = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

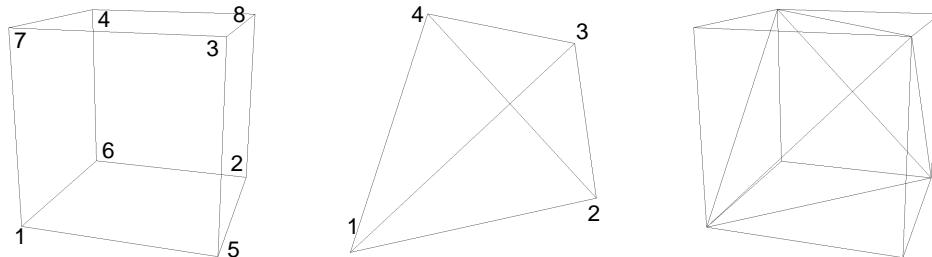


Fig. 15: Wireframe voorstelling van een kubus en tetraheïdron

en

$$Faces^T = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 2 & 4 & 4 & 3 \\ 3 & 3 & 2 & 4 \end{bmatrix}$$

Het resultaat zien we in de middelste afbeelding van Figuur 15. Deze piramide heeft eveneens zijn middelpunt in $(0, 0, 0)$ en maakt gebruik van 4 punten van de kubus, zoals rechts te zien is in Figuur 15.

Octahedron: De octahedron bestaat uit 8 driehoeken en 6 punten en zien we in de linkse afbeelding van Figuur 16:

$$Points^T = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

en

$$Faces^T = \begin{bmatrix} 1 & 2 & 3 & 4 & 2 & 3 & 4 & 1 \\ 2 & 3 & 4 & 1 & 1 & 2 & 3 & 4 \\ 6 & 6 & 6 & 6 & 5 & 5 & 5 & 5 \end{bmatrix}$$

Dit lichaam heeft eveneens zijn middelpunt in $(0, 0, 0)$.

Icosahedron: De icosaheïdron bestaat uit 20 driehoeken en 12 hoekpunten en is in het midden afgebeeld in Figuur 16. We zullen dit lichaam gebruiken om een bol voor te stellen met behulp van een wireframe. De coördinaten

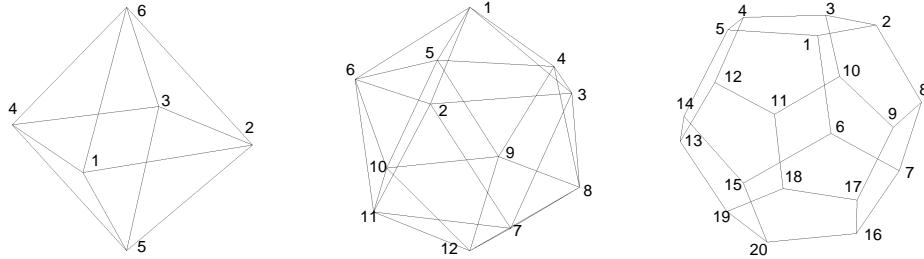


Fig. 16: Wireframe voorstelling van een Octahedron, Icosahedron en een Dodecahedron

van de 12 punten zijn wat complexer en stellen we voor in tabelvorm:

punt i	x_i	y_i	z_i
1	0	0	$\sqrt{5}/2$
2, 3, 4, 5, 6	$\cos((i-2)2\pi/5)$	$\sin((i-2)2\pi/5)$	0.5
7, 8, 9, 10, 11	$\cos(\pi/5 + (i-7)2\pi/5)$	$\sin(\pi/5 + (i-7)2\pi/5)$	-0.5
12	0	0	$-\sqrt{5}/2$

en

$$Faces^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 2 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 6 & 2 & 12 & 12 & 12 & 12 & 12 \\ 2 & 3 & 4 & 5 & 6 & 7 & 7 & 8 & 8 & 9 & 9 & 10 & 10 & 11 & 11 & 11 & 8 & 9 & 10 & 11 & 11 & 7 \\ 3 & 4 & 5 & 6 & 2 & 3 & 8 & 4 & 9 & 5 & 10 & 6 & 11 & 2 & 7 & 7 & 8 & 9 & 10 & 11 & 11 \end{bmatrix}$$

Dit lichaam heeft eveneens zijn middelpunt in $(0, 0, 0)$.

Dodecahedron: Het laatste platonische lichaam is de dodecahedron die bestaat uit 12 vijfhoeken en 20 hoekpunten, deze is rechts afgebeeld in Figuur 16. De coördinaten van de 20 punten bekomen we door het middelpunt van de 20 driehoeken van de icosaëdron te nemen. Bijvoorbeeld, de x -coördinaat van het tweede punt bekomen we door de som van de x -coördinaten van het eerste, derde en vierde punt te delen door drie (zie kolom twee van de array $Faces^T$ van de icosaëdron). De twaalf vijfhoeken worden gegeven door:

$$Faces^T = \begin{bmatrix} 1 & 1 & 2 & 3 & 4 & 5 & 20 & 20 & 19 & 18 & 17 & 16 \\ 2 & 6 & 8 & 10 & 12 & 14 & 19 & 15 & 13 & 11 & 9 & 7 \\ 3 & 7 & 9 & 11 & 13 & 15 & 18 & 14 & 12 & 10 & 8 & 6 \\ 4 & 8 & 10 & 12 & 14 & 6 & 17 & 13 & 11 & 9 & 7 & 15 \\ 5 & 2 & 3 & 4 & 5 & 1 & 16 & 19 & 18 & 17 & 16 & 20 \end{bmatrix}$$

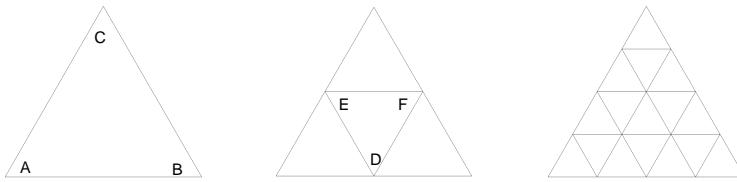


Fig. 17: Opdelen van de driehoeken van de icosaëdron

Ook dit lichaam heeft zijn middelpunt in $(0, 0, 0)$.

Bol (Sphere): We eindigen deze sectie met aan te geven hoe we een wireframe model van een bol kunnen maken. Hiervoor bestaan een aantal mogelijkheden, wij zullen opteren voor het genereren van de bol uitgaande van de icosaëdron, wat een beter ogend resultaat geeft dan wanneer we starten met een octaëdron of werken met vierhoekige vlakjes. Het idee bestaat erin dat we elk van de 20 driehoeken van de icosaëdron gaan vervangen door 4 driehoeken waarvan de zijdes maar half zo groot meer zijn, zoals aangegeven in Figuur 17). We vervangen de driehoek ABC door de driehoeken ADE , BFD , CEF en DFE (waarbij we de hoekpunten in tegenwijzerzin oplossen, het belang hiervan zal later duidelijk worden). De coördinaten van D , E en F bekomen we eenvoudig doordat deze punten halfweg gelegen zijn tussen twee punten waarvan we de coördinaten reeds kennen. Bijvoorbeeld, voor punt E nemen we de som van de coördinaten van A en C en delen deze door twee (dit doen we voor zowel de x -, y - als de z -coördinaat). Dit opdelen herhalen we dan voor elk van de vier kleinere driehoeken, etc.

Wanneer we deze procedure uitvoeren bekomen we een icosaëdron waarvan de 20 vlakken nu zijn opgedeeld in tal van kleinere driehoeken, wat natuurlijk nog geen voorstelling van een bol geeft. Echter wanneer we nu alle punten eenvoudig herschalen zodat ze allemaal op afstand één van de oorsprong gelegen zijn, dan krijgen we een goed wireframe model voor een bol (want het centrum van de icosaëdron was gelegen in $(0, 0, 0)$). Het herschalen van de punten betekent dat we voor elk punt (x_0, y_0, z_0) , zowel de x -, y - als de z -coördinaat hiervan moeten delen door $\sqrt{x_0^2 + y_0^2 + z_0^2}$, wat de afstand is van dit punt tot de oorsprong. Het resultaat zien we in Figuur 18.

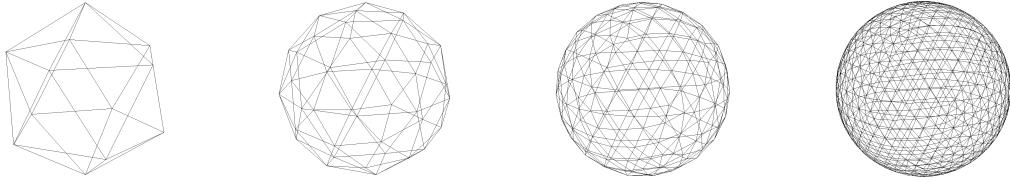


Fig. 18: Een wireframe voorstelling van een bol uitgaande van een icosaëdrone

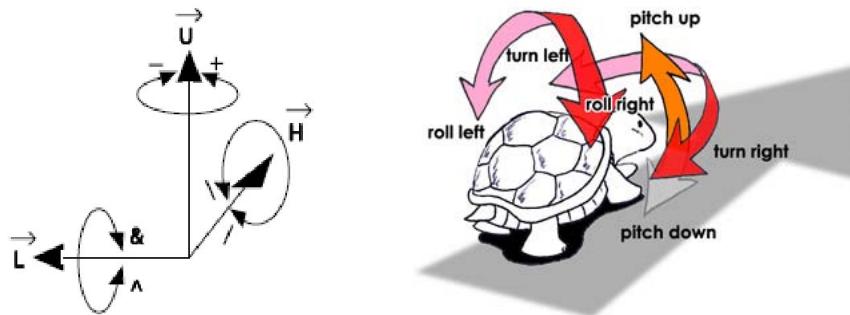


Fig. 19: Oriëntatie van de (zee)schildpad

9 3D L-systemen

In deze sectie geven we kort aan hoe we de eerder besproken twee dimensionale L-systemen kunnen veralgemenen naar drie dimensies. Naast het bijhouden van de huidige positie (in drie dimensies) moeten we ook de richting van de curve (schildpad) onthouden. In twee dimensies volstond het dat we de hoek α bijhielden. In drie dimensies zijn hiervoor minstens twee parameters nodig.

Bijvoorbeeld, wanneer we kijken naar de manier waarop bolcoördinaten kunnen gebruikt worden om een punt in de ruimte te specificeren, dan merken we dat we een richting in drie dimensies volledig kunnen vastleggen met de twee parameters θ en ϕ (zie Figuur 11). We zouden de symbolen $+$ en $-$ wederom kunnen gebruiken om de hoek θ te verhogen of verlagen (met δ) en twee nieuwe symbolen \wedge en $\&$ voor het verhogen en verlagen van ϕ (met δ). Het tekenen van een lijn met lengte 1 vanuit de huidige lokatie (x, y, z) zou dan resulteren in de nieuwe positie $(x + \sin \phi \cos \theta, y + \sin \phi \sin \theta, z + \cos \phi)$, zoals mag blijken uit Figuur 11.

De manier waarop drie dimensionale L-systemen in de literatuur voorkomen is echter iets complexer. Er wordt namelijk gewerkt met drie vectoren met lengte één: $H = (h_x, h_y, h_z)$, $L = (l_x, l_y, l_z)$ en $U = (u_x, u_y, u_z)$, zie ook Figuur 19:

- De vector H geeft de huidige richting aan van de schildpad². Deze wordt doorgaans op $(1, 0, 0)$ geïnitialiseerd (d.i., de x -as).
- De vector L zal steeds loodrecht op H staan en geeft aan welke richting we als links beschouwen. Initieel kiezen we L als $(0, 1, 0)$ (de y -as).
- De vector U geeft aan welke richting we als opwaarts beschouwen. Merk op dat deze reeds volledig vast ligt wanneer H en L bepaald zijn, initieel is U dus $(0, 0, 1)$ (d.i., de z -as).

Een 3D L-systeem zal dus niet enkel de huidige positie (x, y, z) bijhouden, maar eveneens de vectoren H , L en U . Het systeem bestaat opnieuw uit een *Alphabet*, een functie *Draw*, een *Initiator*, een hoek δ en een set van replacement rules. Er zijn nu echter 7 bijzondere symbolen in plaats enkel de + en -:

+ : We draaien naar links over een hoek δ . Met andere woorden, we roteren de H - en L -vector δ radialen om de U -vector. De opwaartse richting blijft dus behouden.

- : We draaien naar rechts over een hoek δ , of nog, we roteren de H - en L -vector $-\delta$ radialen om de U -vector.

[^] : We draaien δ radialen opwaarts, wat overeenstemt met een rotatie om de L -vector van $-\delta$ radialen.

& : In dit geval draaien we δ radialen neerwaarts.

\ : We maken een rolbeweging naar links over δ radialen, wat onze richting niet beïnvloedt. Het gaat hier dus om een rotatie van $-\delta$ radialen om de H -vector.

/ : We maken een rolbeweging naar rechts over δ radialen.

² Merk op, aangezien h lengte één heeft voldoet h aan de vergelijking $\sqrt{h_x^2 + h_y^2 + h_z^2} = 1$, waardoor we eigenlijk ook maar met twee parameters te doen hebben.

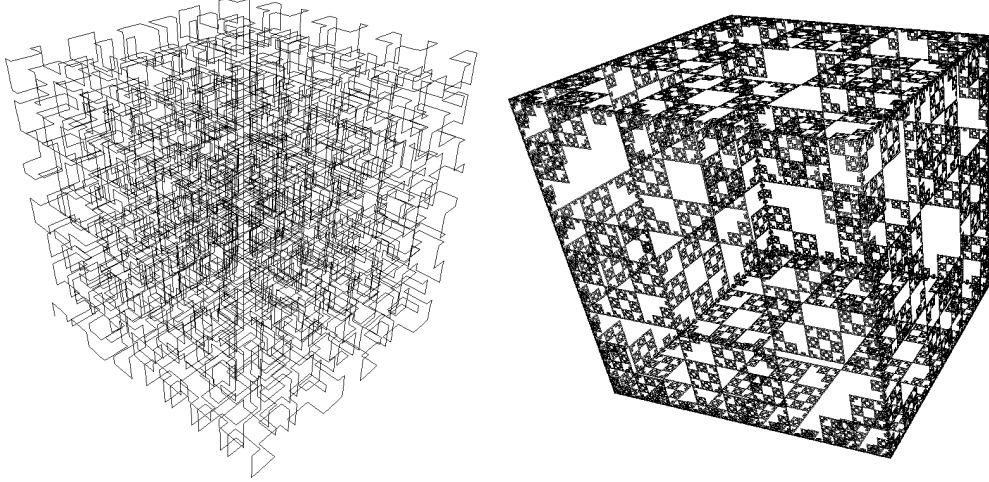


Fig. 20: Hilbert curve in 3D en een kubus met een 2D L-systeem op elke zijde

| : We kerend onze richting om, wat betekent dat we π radialen roteren om de vector U , dewelke ongewijzigd blijft.

Het updaten van de huidige positie is nu erg eenvoudig daar we enkel de vector H hierbij moeten optellen: $(x + h_x, y + h_y, z + h_z)$. Wanneer we een + tegenkomen moeten we H en L als volgt aanpassen:

$$H_{new} = (h_x, h_y, h_z) \cos \delta + (l_x, l_y, l_z) \sin \delta$$

en

$$L_{new} = -(h_x, h_y, h_z) \sin \delta + (l_x, l_y, l_z) \cos \delta,$$

aangezien H en L loodrecht op elkaar staan en lengte één hebben. Bij een - doen we hetzelfde, maar moeten we δ door $-\delta$ vervangen. Wanneer we een ^ tegenkomen moeten we nu H en U analoog aanpassen door (terwijl een & dezelfde aanpassing vereist met $-\delta$ in plaats van δ)

$$H_{new} = (h_x, h_y, h_z) \cos \delta + (u_x, u_y, u_z) \sin \delta$$

en

$$U_{new} = -(h_x, h_y, h_z) \sin \delta + (u_x, u_y, u_z) \cos \delta.$$

Verder zal een \ resulteren in een wijziging in L en U door

$$L_{new} = (l_x, l_y, l_z) \cos \delta - (u_x, u_y, u_z) \sin \delta$$

en

$$U_{new} = (l_x, l_y, l_z) \sin \delta + (u_x, u_y, u_z) \cos \delta,$$

waarbij we voor / opnieuw δ door $-\delta$ vervangen. Tot slot zal de | resulteren in $H_{new} = -H$ en $L_{new} = -L$.

Een voorbeeld van de Hilbert curve in drie dimensies vinden we in links in Figuur 20 terug. Deze werd gegenereerd door $Alphabet = \{F, X\}$, $\delta = \pi/2$, $Draw(F) = Draw(X) = 1$, $Initiator = X$, $F \rightarrow F$ en

$$X \rightarrow ^{\wedge} \backslash XF^{\wedge} \backslash XFX - F^{\wedge} // XFX & F + // XFX - F/X - /.$$

De afbeelding rechts werd eveneens met behulp van een 3D L-systeem gemaakt en stelt een kubus voor waarvan elke zijde het 2-de L-systeem uit Figuur 6 bevat. Het reproduceren van deze Figuur is een uitstekende oefening. Verder zien we ook een voorbeeld van een 3D L-Systeem met haakjes in Figuur 21.

Wanneer we enkel de + en - symbolen in onze replacement rules gebruiken, dan valt het 3D L-systeem samen met het 2D-geval, daar elke rotatie om de U -vector overeenstemt met een rotatie om de z -as. Verder kunnen we deze systemen eveneens uitbreiden met haakjes en stochastische regels.

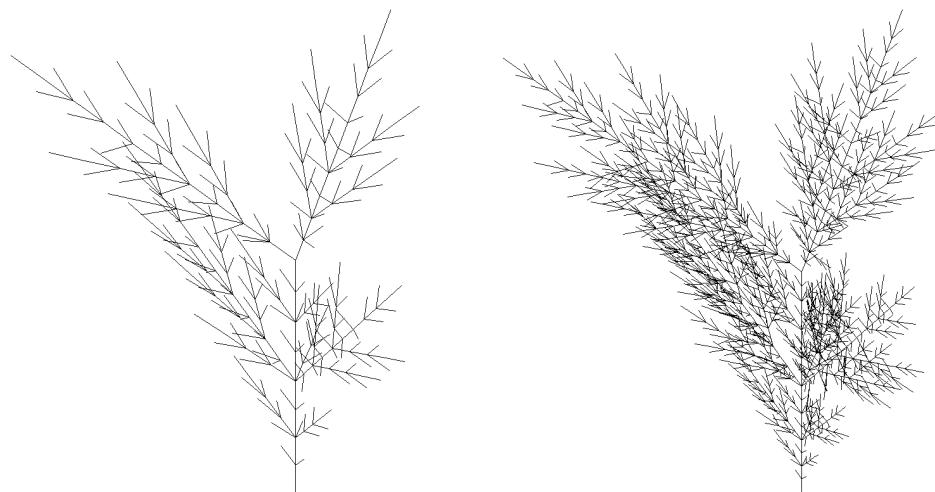


Fig. 21: Voorbeeld van 3D L-systeem voor $n = 3$ en $n = 4$ met
 $\delta = \pi/8$, $Initiator = ^{^\wedge\wedge\wedge\wedge} F$,
 $F \rightarrow F[-\&\backslash G][\backslash + +\&G]F[-- -\&/G][+\&G]$ en
 $G \rightarrow F[+G][-G]F[+G][-G]FG$

Z-BUFFERING EN 3D FRACTALEN

In dit hoofdstuk zullen we bij het maken van 3D-tekeningen rekeningen houden met de diepte, of z -waarde, van de objecten in onze 3D-ruimte wanneer we deze projecteren. We starten met 3D-lijntekeningen waarbij de lijnen verschillende kleuren kunnen hebben. Dit geeft aan dat we moeten beslissen welke kleur een pixel krijgt telkens wanneer er twee of meer lijnen op worden geprojecteerd. Dit zal gebeuren op basis van het Z -buffering algoritme. Onze input bestaat dus wederom uit een array *Points*, *Lines* en *Colors*.

Daarna kijken we naar het geval waarbij de $m \times 2$ array *Lines* wordt vervangen door een $m \times 3$ array *Trias* die op elke rij het nummer van drie punten bevat. De drie punten op rij i , voor $i = 1, \dots, m$, zullen steeds de hoekpunten vormen van een ondoorzichtige driehoek die is ingekleurd met de kleur in element i van de array *Colors*. Bij het maken van de afbeelding dienen we dan wederom voor elke pixel te beslissen welke kleur we gebruiken wanneer twee of meer driehoeken deze pixel zouden inkleuren. Hiervoor zullen we eveneens het Z -buffering algoritme hanteren.

Merk op dat een driehoek gedeeltelijk zichtbaar kan zijn en dat driehoeken elkaar ook kunnen doorboren. Verder merken we nog op dat elke veelhoek (bv. vierhoek, vijfhoek, etc.) kan worden opgedeeld in driehoeken, waardoor we bijvoorbeeld ook de platonische lichamen uit de voorgaande sectie kunnen afbeelden met ondoorzichtige zijdes. Met behulp van deze lichamen zullen we ook enkele fractalen in 3D genereren om de kracht van het Z -buffering algoritme te demonstreren.

10 Z-buffering voor 3D-lijntekeningen met kleuren

In het voorgaande hoofdstuk bespraken we hoe we een 3D-tekening konden maken die was opgebouwd uit lijnen. De input voor deze tekeningen bestond uit een array *Points* en *Lines*. Nu gaan we eveneens kleuren toevoegen door de array *Colors* opnieuw in te voeren zoals in het 2D geval. Dit zorgt voor een extra moeilijkheid daar we gebruik moeten maken van de juiste kleur op de afbeelding indien meerdere lijnen elkaar kruisen in het geprojecteerde

beeld. Met andere woorden we moeten voor elke pixel de kleur gebruiken van het punt met de grootste z -coördinaat (in het *Eye*-coördinaat systeem, waarin alle punten negatieve z -coördinaten hebben) dat op deze pixel wordt afgebeeld.

Om dit te realiseren zullen we gebruik maken van een Z -buffer, die één element bevat voor elke pixel in onze afbeelding, deze buffer zal voor elke pixel onthouden wat de z -coördinaat is van het punt dat tot nu toe het dichts gelegen was bij ons oogpunt, of nog, dat tot nu toe de grootste z -coördinaat had. We zullen dadelijk aangeven dat de buffer eigenlijk de inverse van deze z -coördinaat bijhoudt, wat betekent dat de tot nu toe kleinste inverse waarde wordt onthouden.

Stel nu dat punt P met coördinaten (x_P, y_P, z_P) in het *Eye*-coördinaat systeem wordt geprojecteerd op pixel (x'_P, y'_P) en dit punt bevindt zich op een lijn met kleur k . Dan zullen we moeten nagaan of $1/z_P$ kleiner is dan de waarde die is opgeslagen op rij x'_P en kolom y'_P van de Z -buffer. Is dit niet het geval, dan is deze lijn achter een andere lijn gelegen en herkleuren we de pixel niet. Is de $1/z_P$ -waarde wel kleiner, dan (her)kleuren we de pixel in kleur k en slaan de inverse z -coördinaat $1/z_P$ op in de Z -buffer op lokatie (x'_P, y'_P) . Merk op, het kan ook zijn dat pixel (x'_P, y'_P) helemaal nog niet was ingekleurd. Om dit op te vangen worden alle elementen van de Z -buffer op $+\infty$ geïnitialiseerd, waardoor het eerste punt dat wordt afgebeeld op een pixel steeds resulteert in het inkleuren van de pixel.

Het probleem is echter dat we van een punt dat we wensen te tekenen enkel de coördinaten (x'_P, y'_P) kennen en niet (x_P, y_P, z_P) , tenzij het om één van de eindpunten van een lijn gaat (want hiervoor staan de coördinaten (x_P, y_P, z_P) in de lijst *Points*). Om dit op te lossen zullen we in de volgende sectie aantonen dat als I een punt is gelegen op een lijn tussen punten A en B , dan zal de z -coördinaat z_I van dit punt voldoen aan de vergelijking

$$\frac{1}{z_I} = \frac{p}{z_A} + \frac{(1-p)}{z_B},$$

waarbij z_A en z_B de z -coördinaten van A en B zijn en p is de waarde tussen nul en één zodat de projectie van $I = (x'_I, y'_I)$ gelijk is aan

$$(x'_I, y'_I) = p(x'_A, y'_A) + (1-p)(x'_B, y'_B).$$

Zo een p waarde bestaat daar de projectie van I eveneens gelegen is op de lijn tussen de projectie (x'_A, y'_A) van A en de projectie (x'_B, y'_B) van B .

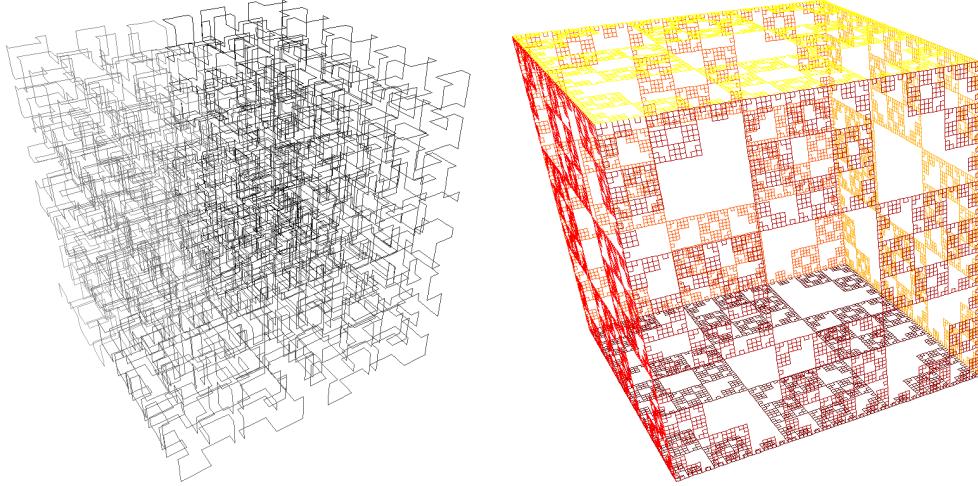


Fig. 22: Hilbert curve in 3D en een kubus met een 2D L-systeem op elke zijde in kleur

Wanneer we nu $a + 1$ pixels gebruiken om de eindpunten A en B van een lijn te verbinden, dan kunnen we hierdoor voor het bepalen van de $a + 1$ vereiste inverse z -coördinaten, gewoon devolgende $a + 1$ combinaties van $1/z_A$ en $1/z_B$ hanteren

$$\frac{i/a}{z_A} + \frac{(1 - i/a)}{z_B},$$

met $i = a, a - 1, \dots, 1, 0$. In Figuur 22 zien we een heruitgave van Figuur 20 waarbij we gebruik maken van gekleurde lijnen (en waarbij ons oogpunt is verplaatst).

11 $1/z$ -Interpolatie

In deze sectie tonen we kort aan dat als $A = (x_A, y_A, z_A)$ en $B = (x_B, y_B, z_B)$ twee punten zijn die door de perspectiefprojectie worden afgebeeld op twee verschillende punten (x'_A, y'_A) en (x'_B, y'_B) , dan zal het punt $I = (x_I, y_I, z_I)$ gelegen op de lijn tussen A en B met geprojecteerde coördinaten

$$(x'_I, y'_I) = p(x'_A, y'_A) + (1 - p)(x'_B, y'_B),$$

met $p \in (0, 1)$ als overeenkomstige $1/z_I$ -coördinaat $p/z_A + (1 - p)/z_B$ hebben. Zonder verlies van algemeenheid veronderstellen we dat $x'_A \neq x'_B$.

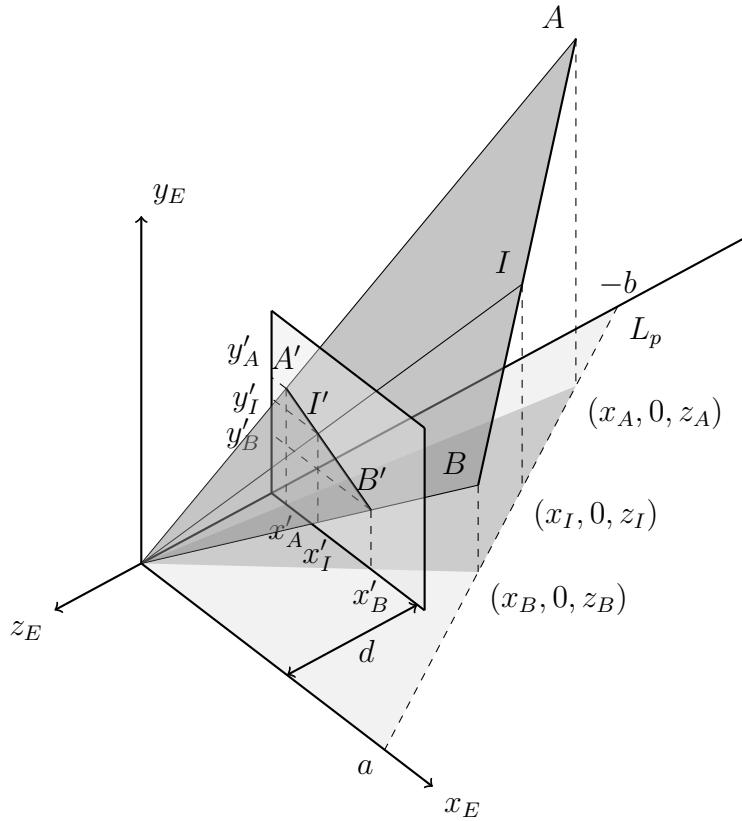


Fig. 23: 1/z-interpolatie bij de perspectiefprojectie

Om dit in te zien, volstaat het te kijken naar de projectie van de punten A , B en I op het xz -vlak (zie Figuur 23 en ook de gelijkenis met Figuur 13). We definiëren a en $-b$ als de waarden waarin de geprojecteerde lijn L_p tussen A en B de x - en z -as snijdt, respectievelijk. Merk op, indien $a = b = 0$ dan zou $x'_A = x'_B$, wat we hebben uitgesloten. Kortom, a en b zijn niet allebei gelijk aan nul. De vergelijking van deze geprojecteerde lijn is daarom gelijk aan (want $x = 0$ invullen geeft $z = -b$ en $x = a$ invullen geeft $z = 0$)

$$z = \frac{bx}{a} - b = \left(\frac{x}{a} - 1\right)b.$$

Dankzij Sectie 6.2 weten we reeds dat

$$x = \frac{-x'z}{d}.$$

Indien we deze twee vergelijken combineren dan verkrijgen we

$$z = \left(\frac{-x'z}{ad} - 1 \right) b.$$

Beide kanten delen door zb resulteert in

$$\frac{1}{z} = \frac{-x'}{ad} - \frac{1}{b},$$

wanneer we de termen herschikken. Belangrijk is op te merken dat deze vergelijking opgaat voor alle punten op de geprojecteerde lijn L_p . Ze gaat dus op voor de geprojecteerde punten $(x_A, 0, z_A)$, $(x_B, 0, z_B)$ en $(x_I, 0, z_I)$. Met andere woorden:

$$\frac{1}{z_A} = \frac{-x'_A}{ad} - \frac{1}{b}, \quad \frac{1}{z_B} = \frac{-x'_B}{ad} - \frac{1}{b}, \quad \frac{1}{z_I} = \frac{-x'_I}{ad} - \frac{1}{b}.$$

Als we dan $p/z_A + (1-p)/z_B$ uitrekenen verkrijgen we

$$p\frac{1}{z_A} + (1-p)\frac{1}{z_B} = p\left(\frac{-x'_A}{ad} - \frac{1}{b}\right) + (1-p)\left(\frac{-x'_B}{ad} - \frac{1}{b}\right) = \frac{-x'_I}{ad} - \frac{1}{b} = \frac{1}{z_I},$$

zoals te bewijzen was.

12 Inkleuren van een driehoek

In deze sectie bespreken we een algoritme om te bepalen welke pixels (x, y) we moeten inkleuren wanneer we een driehoek met geprojecteerde hoekpunten $A = (x'_A, y'_A)$, $B = (x'_B, y'_B)$ en $C = (x'_C, y'_C)$ wensen in te kleuren. Het algoritme zal voor elke y -waarde de minimale en maximale x -waarde bepalen, dewelke we noteren als x_L en x_R , respektievelijk. De set van te evalueren y -waardes $Y = \{y_{min}, y_{min} + 1, \dots, y_{max}\}$ is duidelijk gelijk aan

$$\{round(\min(y'_A, y'_B, y'_C) + 0.5), \dots, round(\max(y'_A, y'_B, y'_C) - 0.5)\}.$$

Het introduceren van de 0.5 gebeurt om te vermijden dat twee aan elkaar grenzende driehoeken dezelfde pixel zouden inkleuren. Voor elke $y_I \in Y$ stellen we nu een routine op om het snijpunt te bepalen van de lijn $y = y_I$ met het lijnstuk PQ gevormd door twee punten $P = (x_P, y_P)$ en $Q = (x_Q, y_Q)$. We gebruiken vervolgens deze routine voor de drie lijnstukken AB , AC en BC .

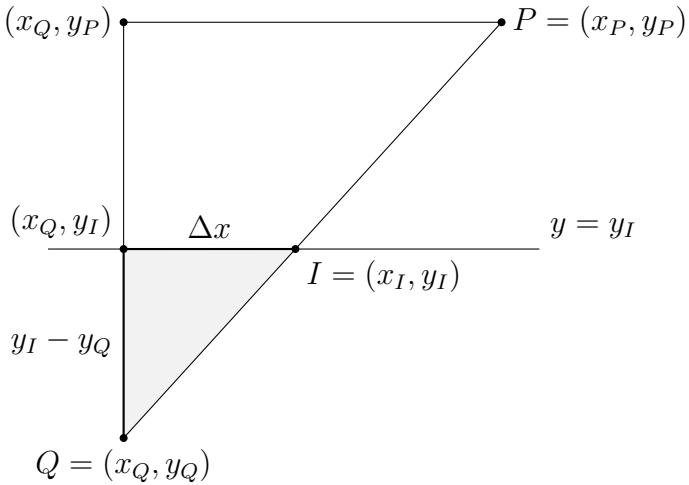


Fig. 24: Bepalen van het snijpunt van het lijnstuk \$PQ\$ met de lijn \$y = y_I\$

Ten eerste merken we op dat er geen sprake is van een snijpunt indien \$y_I < y_P\$ en \$y_I < y_Q\$, want dan is het lijnstuk boven de lijn \$y = y_I\$ gelegen, tevens is er ook geen snijpunt indien \$y_I > y_P\$ en \$y_I > y_Q\$, wat zou aangeven dat het lijnstuk onder de lijn \$y = y_I\$ gelegen was. In alle andere gevallen is er wel sprake van een snijpunt, we kunnen dit samenvatten door te stellen dat volgende voorwaarde moet gelden voor het hebben van een snijpunt:

$$(y_I - y_P)(y_I - y_Q) \leq 0.$$

Tevens moet ook \$y_P \neq y_Q\$, anders loopt het lijnstuk evenwijdig met de lijn \$y = y_I\$, waardoor er geen of oneindig veel (indien \$y_I = y_P = y_Q\$) snijpunten zijn. Wanneer we nu kijken naar Figuur 24 dan merken we op dat de driehoek gevormd door \$P\$, \$Q\$ en \$(x_Q, y_P)\$ gelijkvormig is aan de driehoek \$Q\$, \$I\$ en \$(x_Q, y_I)\$. Dit impliceert dat

$$\frac{\Delta x}{y_I - y_Q} = \frac{x_P - x_Q}{y_P - y_Q}.$$

Anderzijds is \$x_I\$, de \$x\$-waarde die we zoeken, gelijk aan \$x_Q + \Delta x\$, wat aangeeft dat

$$x_I = x_Q + (x_P - x_Q) \frac{y_I - y_Q}{y_P - y_Q}.$$

Het inkleuren van de driehoek kan nu als volgt gebeuren, we gaan voor alle y_I waardes in Y de x_L - en x_R -waarde bepalen wat aangeeft dat de pixels (x_L, y_I) tot (x_R, y_I) tot de driehoek behoren. Om x_L en x_R te bepalen voor een enkele y -waarde y_I gebruiken we 6 variabelen die we als volgt initialiseren:

$$x_L^{(AB)} = x_L^{(AC)} = x_L^{(BC)} = +\infty, \quad x_R^{(AB)} = x_R^{(AC)} = x_R^{(BC)} = -\infty.$$

We testen voor elk van de drie lijnstukken PQ gelijk aan AB , AC en BC of

$$(y_I - y_P)(y_I - y_Q) \leq 0 \quad \text{en} \quad y_P \neq y_Q.$$

Merk op dat deze test exact twee maal succesvol is³, waardoor één $x_L^{(PQ)}$ waarde nog gelijk is aan $+\infty$ en één $x_R^{(PQ)}$ waarde gelijk aan $-\infty$. Indien de test succesvol is bepalen we de waarde x_I door de eerder bekomen formule en stellen $x_L^{(PQ)}$ en $x_R^{(PQ)}$ hieraan gelijk. Tenslotte laten we

$$\begin{aligned} x_L &= \text{round}(\min(x_L^{(AB)}, x_L^{(AC)}, x_L^{(BC)}) + 0.5), \\ x_R &= \text{round}(\max(x_R^{(AB)}, x_R^{(AC)}, x_R^{(BC)}) - 0.5), \end{aligned}$$

waarbij de 0.5 er voor zorgt dat aangrenzende driehoeken geen pixels delen. Tot slot merken we op dat er geen enkele pixel gekleurd wordt indien de drie geprojecteerde punten A , B en C op één lijn gelegen zijn. Men zou in dit geval kunnen opteren om de driehoek als lijn weer te geven.

13 Z-buffering met driehoeken

In deze sectie geven we aan hoe we een 3D-afbeelding kunnen maken die is opgebouwd uit een verzameling ondoorzichtige driehoeken. De input wordt dus gevormd door de arrays *Points*, *Trias* en *Colors*. Net zoals in het voorgaande hoofdstuk zullen we eerst alle punten omzetten in het *Eye*-coördinaat systeem. Daarna gaan we per driehoek ABC elk van de pixels die behoren tot deze driehoek bekijken. We kunnen hiervoor gebruik maken van de methode uit de voorgaande sectie. Echter in plaats van elke pixel zomaar in te kleuren, moeten we nagaan of het punt op de driehoek ABC dat met deze pixel overeenstemt wel degelijk dichter ligt dan de overige tot op heden bekeken driehoeken die deze pixel eveneens wenste in te kleuren.

³ Enkel indien PQ op de lijn $y = y_I$ gelegen is, faalt de test omwille van de tweede conditie, in dit geval zullen de twee overige lijnstukken x_L en x_R bepalen.

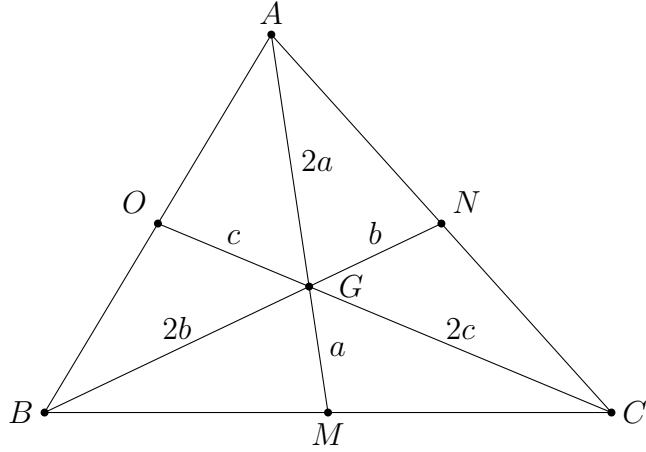


Fig. 25: $1/z$ -coördinaat van het centrum G van de geprojecteerde driehoek ABC

We doen dit net zoals in Sectie 10 door een Z -buffer bij te houden die de kleinste tot op heden tegengekomen $1/z$ -waarde voor elke pixel bijhoudt (en die geïnitialiseerd wordt op $+\infty$). Enkel wanneer de $1/z$ -waarde van de huidige driehoek ABC kleiner is krijgt de pixel (voorlopig) de kleur van de driehoek ABC . Het probleem dat resteert bestaat er dan ook in om een efficiënte manier te vinden om de $1/z$ -waarde te bepalen van een willekeurige pixel (x, y) die behoort tot de driehoek ABC .

Hiervoor gaan we eerst de $1/z$ -waarde bepalen van het centrum van de *geprojecteerde* driehoek ABC met hoekpunten $A = (x_A, y_A)$, $B = (x_B, y_B)$ en $C = (x_C, y_C)$ (d.i., het massamiddelpunt). Dit is het snijpunt van de drie rechtes weergegeven in Figuur 25. We noteren de coördinaten van G als (x_G, y_G) en merken op dat $x_G = (x_A + x_B + x_C)/3$ en $y_G = (y_A + y_B + y_C)/3$.

Figuur 25 geeft aan dat het centrum G op de lijn tussen C en O gelegen is, terwijl O zich halfweg tussen A en B bevindt (met andere woorden $x_O = (x_A + x_B)/2$ en $y_O = (y_A + y_B)/2$). Omwille van het resultaat uit Sectie 11 weten we bijgevolg dat

$$\frac{1}{z_O} = \frac{1}{2z_A} + \frac{1}{2z_B}.$$

Verder ligt G exact twee maal zo dicht bij O als bij C , want $x_G = (x_A + x_B + x_C)/3 = 2x_O/3 + x_C/3$ en $y_G = (y_A + y_B + y_C)/3 = 2y_O/3 + y_C/3$. Kortom,

$G = 2/3O + 1/3C$, waardoor we bekomen dat

$$\frac{1}{z_G} = \frac{2}{3z_O} + \frac{1}{3z_C},$$

wat in combinatie met de voorgaande vergelijking leidt tot

$$\frac{1}{z_G} = \frac{1}{3z_A} + \frac{1}{3z_B} + \frac{1}{3z_C}.$$

Deze $1/z_G$ -waarde zal als basis dienen voor het bepalen van de $1/z$ -waarde van elke andere pixel (x, y) die behoort tot de driehoek ABC .

In de volgende sectie zullen we zien dat voor alle pixels (x, y) die behoren tot de driehoek ABC het verhogen van x met één de $1/z$ -waarde met een constante hoeveelheid, die we noteren als $dzdx$, zal doen toenemen (de toename is dus onafhankelijk van de waarde van x en y). Hetzelfde geldt wanneer we de y -waarde doen toenemen met één, in dit geval noteren we de toename als $dzdy$. De $1/z$ -waarde van de pixel (x, y) wordt vervolgens berekent als

$$\frac{1}{z} = 1.0001 \frac{1}{z_G} + (x - x_G)dzdx + (y - y_G)dzdy.$$

Als we even de factor 1.0001 buiten beschouwing laten, dan nemen we dus de $1/z$ -waarde van het centrum G en corrigeren die met $(x - x_G)dzdx$ omdat onze pixel $(x - x_G)$ pixels naast de centrum pixel gelegen is in de x -richting en met $(y - y_G)dzdy$, daar de pixel (x, y) in de y -richting $(y - y_G)$ pixels verwijderd is van het centrum.

De factor 1.0001 wordt gebruikt om mogelijke conflicten tussen twee naburige driehoeken op te lossen. Bijvoorbeeld, indien we naar een huis kijken met een schuin dak en dit dak heeft een andere kleur aan de voorzijde dan aan de achterzijde, dan kunnen we ons de vraag stellen welke kleur de pixels krijgen die de bovenste rand van het dak voorstellen. Aangezien de voor-en achterzijde dezelfde maximale y -waarde hebben, bieden de 0.5-waarden die we eerder invoerden voor conflicten tussen naburige driehoeken in dit geval geen oplossing. Echter, daar het centrum van de voorzijde een kleinere $1/z$ -waarde heeft dan het centrum van de achterzijde, zorgt de 1.0001 factor ervoor dat de voorzijde een iets kleinere $1/z$ -waarde krijgt, waardoor de dakrand de correcte kleur krijgt.

14 Bepaling van dz/dx en dz/dy

Deze sectie is wellicht de meest wiskundige van de cursus. De bedoeling is aan te geven hoe we de waarde dz/dx en dz/dy moeten bepalen voor een driehoek met hoekpunten A , B en C . Deze waardes geven aan hoeveel $1/z$ wijzigt indien we x' en y' met één verhogen.

We starten met het aanschouwen van de vergelijking van het vlak dat de punten A , B en C bevat. Hiervoor zijn er twee mogelijkheden: ten eerste kan het zijn dat dit vlak het punt $(0, 0, 0)$ bevat, waardoor de driehoek zal worden geprojecteerd op een lijn en de vergelijking van het vlak van de vorm

$$ax + by + cz = 0$$

is. In dit geval hoeven we dz/dx en dz/dy niet te bepalen, daar er geen enkele pixel wordt ingekleurd indien de drie geprojecteerde hoekpunten op een lijn liggen. Het alternatief bestaat erin deze lijn te tekenen zoals we bespraken in Sectie 10.

In het tweede geval bevat het vlak het punt $(0, 0, 0)$ niet en kunnen we de vergelijking van het vlak dat A , B en C bevat schrijven als

$$ax + by + cz = 1.$$

Door gebruik te maken van de ondertussen gekende gelijkheden $x' = -dx/z$ en $y' = -dy/z$ kunnen we dit herschrijven als

$$\frac{ax'}{-d} + \frac{by'}{-d} + c = 1/z,$$

door eveneens links en rechts te delen door z . Met andere woorden, als we x' verhogen met één dan zal $1/z$ met $-a/d$ toenemen en y' verhogen geeft een toename van $-b/d$, of nog

$$dz/dx = \frac{a}{-d}, \quad dz/dy = \frac{b}{-d}.$$

We moeten dus enkel de waarde van a en b nog bepalen. Hiervoor hebben we wat basiskennis meetkunde nodig. Ten eerste zal de richting (a, b, c) loodrecht staan op het vlak met vergelijking $ax + by + cz = 1$. Dit kunnen we inzien door eerst een translatie uit te voeren zodat het vlak het punt $(0, 0, 0)$ bevat. Hierdoor zal de richting (a, b, c) niet wijzigen en verkrijgen

we de vergelijking $ax + by + cz = 0$. Nu is het scalaire product tussen twee vectoren $u = (u_1, u_2, u_3)$ en $v = (v_1, v_2, v_3)$ is gelijk aan $u_1v_1 + u_2v_2 + u_3v_3$ en kan dit eveneens gescheven worden als de lengte van u maal de lengte van v maal de cosinus van de hoek tussen u en v , wat aangeeft dat het scalair product nul is indien de hoek $\pi/2$ radianen bedraagt. Met andere woorden de vergelijking $ax + by + cz = 0$ geeft aan dat het scalaire product tussen alle vectoren in het vlak en de vector (a, b, c) gelijk is aan nul, wat enkel het geval kan zijn indien (a, b, c) loodrecht staat op het vlak.

Het vectoriële product tussen twee vectoren $u = (u_1, u_2, u_3)$ en $v = (v_1, v_2, v_3)$ is dan weer gelijk aan de vector $w = (w_1, w_2, w_3)$ met

$$\begin{aligned} w_1 &= u_2v_3 - u_3v_2, \\ w_2 &= u_3v_1 - u_1v_3, \\ w_3 &= u_1v_2 - u_2v_1. \end{aligned}$$

Tevens staat de vector w loodrecht op het vlak dat gevormd wordt door u en v . Met andere woorden indien we u gelijk nemen aan de vector tussen A en B en v gelijk aan de vector tussen A en C , wat wil zeggen

$$\begin{aligned} u &= (x_B, y_B, z_B) - (x_A, y_A, z_A), \\ v &= (x_C, y_C, z_C) - (x_A, y_A, z_A), \end{aligned}$$

dan zal w loodrecht staan op het vlak gevormd door de punten A , B en C , of nog w is gelijk aan (a, b, c) op een constante na (aangezien alle veelvouden van w eveneens loodrecht staan op dit vlak)

$$(w_1, w_2, w_3) = k(a, b, c).$$

De waarde van k kunnen we bepalen door

$$w_1x_A + w_2y_A + w_3z_A = k(ax_A + by_A + cz_A) = k,$$

aangezien A in het vlak gelegen is. Met andere woorden,

$$dz/dx = \frac{w_1}{-dk}, \quad dz/dy = \frac{w_2}{-dk}.$$

Merk op dat indien het vlak gevormd door A , B en C het punt $(0, 0, 0)$ wel bevat (en de driehoek dus op een lijn wordt geprojecteerd), dan kunnen we bovenstaande berekeningen eveneens uitvoeren. We zullen echter merken dat $w_1x_A + w_2y_A + w_3z_A$ gelijk is aan nul daar $ax_A + by_A + cz_A$ gelijk is aan nul in plaats van één. Kortom, we zullen automatisch vaststellen dat het vlak door de oorsprong $(0, 0, 0)$ gaat, waardoor we dz/dx en dz/dy niet hoeven te bepalen.

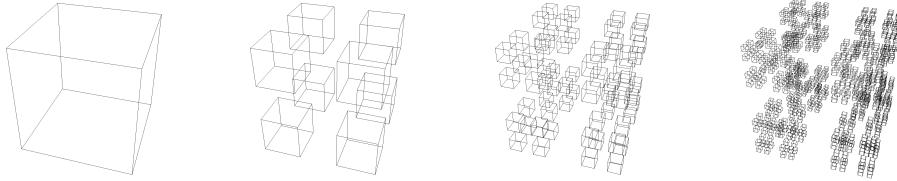


Fig. 26: 3D-fractaal op basis van een kubus, we verkleinen steeds met een factor $scale = 3$

15 3D-fractalen

In deze sectie geven we kort aan hoe we een bepaalde klasse van 3D-fractalen kunnen genereren op basis van een enkel initieel (platonisch) lichaam. Stel dat we starten met een lichaam dat bestaat uit h hoekpunten en f oppervlakken. Bijvoorbeeld voor de icosaëdron is $h = 12$ en $f = 20$. Kortom, *Points* bevat h rijen en de array *Faces* heeft precies f rijen.

Om te komen tot een fractaal gaan we n stappen uitvoeren, met de bedoeling tijdens elke stap elk lichaam te vervangen door h kleinere lichamen zodat het j -de hoekpunt van het j -de kleinere lichaam samenvalt met het j -de hoekpunt van het grotere lichaam en dit voor $j = 1, \dots, h$. In Figuur 26 zien we een voorbeeld op basis van een kubus in wireframe waarbij het resultaat van de eerste 3 stappen is weergegeven samen met het initiële lichaam. De factor waarmee we telkens het lichaam verkleinen noteren we als *scale*.

De lijst van punten en oppervlakken na i stappen noteren we als *Points_i* en *Faces_i*. Na i stappen zullen we over h^{i+1} punten beschikken en fh^i faces. Om de set van punten en faces te genereren gaan we als volgt te werk. Stel dat we reeds $i - 1$ stappen hebben uitgevoerd, waardoor we beschikken over een *Points_{i-1}* array met h^i rijen en een *Faces_{i-1}* array met fh^{i-1} rijen. De lijsten *Points_i* en *Faces_i* zijn initieel leeg. We overlopen nu de array *Points_{i-1}* rij per rij met de bedoeling tijdens stap i elk punt op deze lijst te vervangen door een lichaam dat gelijk is aan het initiële lichaam verkleind met een factor $scale^i$. Met andere woorden voor elke rij k in *Points_{i-1}* voegen we h nieuwe rijen toe aan *Points_i*. Om de coördinaten van deze h rijen te bekomen zullen we

1. De h rijen van de array *Points*, die de hoekpunten van het initiële lichaam bevat, delen door $scale^i$, wat aangeeft dat het centrum van het initiële lichaam in het centrum van ons coördinaat systeem moet

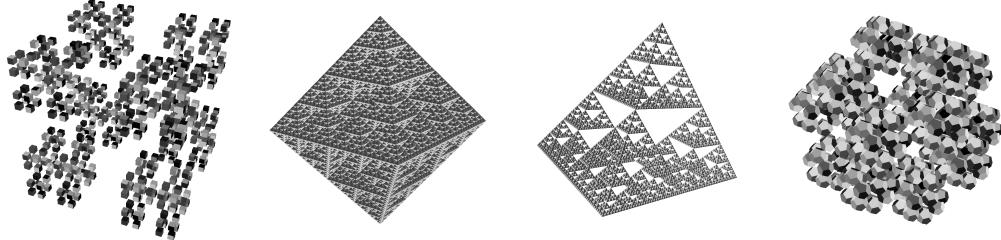


Fig. 27: Voorbeelden van 3D-fractalen op basis van platonische lichamen

liggen. We slaan het resultaat hiervan op in *addPoints*. Dit hoeven we slechts één maal te doen voor alle k -waarden.

2. We voeren nu twee translaties uit op de h punten in *addPoints*.

- (a) Eerst zorgen we ervoor dat het juiste hoekpunt j van het kleinere lichaam met punten *addPoints* in de oorsprong komt te liggen. De keuze van j wordt bepaald door k , het punt dat we door dit kleinere lichaam wensen te vervangen. Namelijk, dit punt was zelf het j -de hoekpunt van een groter lichaam (voor een bepaalde j tussen 1 en h). Om de waarde van j te bepalen maken we gebruik van het feit dat de lijst *Points* _{$i-1$} zo is opgesteld dat we de punten lichaam per lichaam opsommen, hierdoor is $j = (k - 1 \bmod h) + 1$. De eerste translatie bestaat er dus in dat we van elke rij van *addPoints* de j -de rij aftrekken (waardoor het j -de punt in de oorsprong terecht komt).
- (b) Daarna volstaat het de k -de rij van *Points* _{$i-1$} bij elke rij van *addPoints* op te tellen, waardoor het j -de hoekpunt van het kleinere lichaam met hoekpunt k samenvalt. De h rijen van *addPoints* voegen we nu toe aan *Points* _{i} .

Doordat we de punten lichaam per lichaam oplijsten, kunnen we de oppervlakken van het kleinere lichaam eenvoudig beschrijven door tijdens de k -de iteratie de oppervlakken $Faces + h * (k - 1)$ toe te voegen aan *Faces* _{i} .

In Figuur 27 zien we een voorbeeld voor 4 platonische lichamen: de kubus, octahedron, tetrahedron en dodecahedron. De *scale* factor is gelijk aan 3, 2, 2 en $2 + (1 + \sqrt{5})/2$, respectievelijk. Deze methode kunnen we echter ook gebruiken voor niet-platonische lichamen zoals aangegeven in Figuur 28 voor

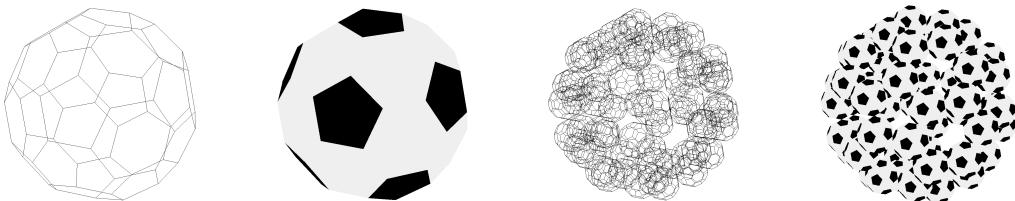


Fig. 28: 3D-fractal op basis van een Buckyball met $scale = 1 + 3(1 + \sqrt{5})/2$

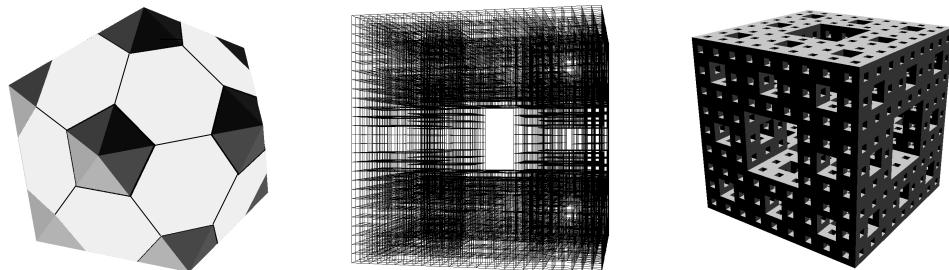


Fig. 29: Links: Constructie van de buckyball via de icosahedron. Midden en rechts: Menger spons in wireframe en met Z-buffering

de zogenaamde buckyball (of zijn meer gesofisticeerde naam buckminsterfullerene). Voor de meeste is dit lichaam nog het beste gekend als een voetbal die bestaat uit 20 zeshoeken en 12 vijfhoeken. Je kan de buckyball construeren door elk van de 20 driehoeken van de icosahedron op te delen in een gelijkzijdige zeshoek en drie driehoeken. Zoals aangegeven in Figuur 29 ontstaan er hierdoor piramides met vijf zijdes. Wanneer we van deze piramides enkel het grondvlak overhouden dan verkrijgen we een buckyball. Vandaar dat de buckyball ook gekend is onder de naam truncated icosahedron.

Er zijn ook tal van interessante 3D-fractalen die we niet zomaar op deze wijze kunnen genereren. Een voorbeeld hiervan is de Menger spons die afgebeeld is in Figuur 29, deze bestaat uit 8000 kubussen en is zowel in wireframe als met Z-buffering weergegeven.

LICHT, SCHADUW EN TEXTURE MAPPING

In dit hoofdstuk zullen we aangeven hoe we licht en schaduw kunnen toevoegen aan onze afbeeldingen, alsook hoe we een oppervlak kunnen voorzien van een bepaalde textuur. We zullen hierbij een onderscheid maken tussen drie types van belichting: *ambient*, *diffuus* en *specular* licht, waarbij we twee varianten van diffuus licht zullen bekijken. Daarna geven we aan hoe we met behulp van het Z-buffering algoritme uit het voorgaande hoofdstuk eveneens schaduw kunnen toevoegen, waardoor delen van objecten mogelijk in de schaduw komen te liggen van andere objecten. We zullen dit niet enkel doen wanneer er slechts één lichtbron is, maar ook voor meerdere lichtbronnen die tevens gepositioneerd kunnen zijn tussen de objecten die we bekijken, zoals bijvoorbeeld in een 3D-fractaal. Daarna bespreken we hoe we het realiteitsgehalte van onze afbeelding verder kunnen verbeteren door afbeeldingen te mappen op bepaalde oppervlakken in onze 3D-ruimte om bijvoorbeeld een tafelblad van een houtstructuur te voorzien.

16 Licht

We maken een onderscheid tussen drie types van licht en zullen starten met de meest eenvoudige vorm: ambient licht. Voor we hierop kunnen ingaan moeten we eerst even stilstaan bij een precieze beschrijving van licht. We maken hiervoor gebruik van het RGB model, wat staat voor *Red*, *Green*, *Blue*. In dit model, dat voortvloeit uit de manier waarop het menselijke oog licht waarneemt, bestaat elke lichtstraal uit drie componenten: een rode, groene en blauwe. De intensiteit van elke component wordt weergegeven door een cijfer tussen nul en één. In geval van rood licht zijn deze drie componenten gelijk aan $(i_R, 0, 0)$, met i_R tussen nul en één. Hoe kleiner i_R hoe donkerder het rood wordt, waarbij $i_R = 0$ zal resulteren in zwart. Voor groen licht hebben we $(0, i_G, 0)$ en voor blauw licht hebben we $(0, 0, i_B)$. Alle andere kleuren worden gevormd door combinaties van deze drie kleuren, zo worden de geel tinten bekomen door $(i_R, i_G, 0)$, met $i_R = i_G$ en is wit licht gelijk aan

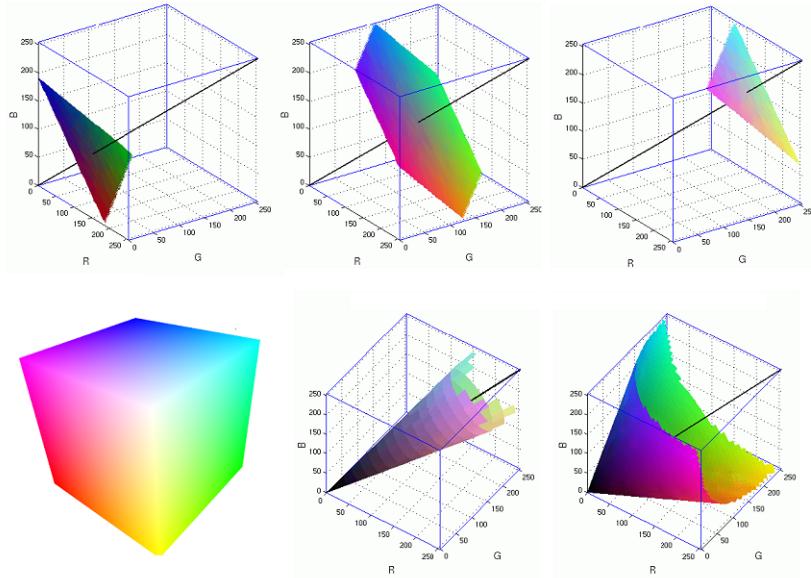


Fig. 30: RGB model voor kleuren

$(1, 1, 1)$. Alle grijstinten zijn van de vorm (i_R, i_R, i_R) en vormen inderdaad combinaties van wit en zwart licht (namelijk i_R maal wit licht plus $1 - i_R$ keer zwart licht). De helderheid van het licht wordt doorgaans aangegeven door de som van de drie componenten.

In Figuur 30 zien we een afbeelding van de RGB color cube, alsook drie vlakken met een constante helderheid (25, 50 en 75 %). De laatste twee afbeeldingen geven twee kegels weer met constante saturatie (20 en 70%). Deze laatste geeft aan hoe kleurrijk de kleur is in verhouding met zijn helderheid. De grijstinten hebben een saturatie gelijk aan nul. We merken ook op dat de waarden van de drie componenten in deze afbeelding zijn herschaald tussen 0 en 255.

16.1 Ambient licht

Ambient licht is de meest eenvoudige vorm van licht om te implementeren en is de enige vorm van licht dat invalt op een object dat volledig in de schaduw is gelegen van alle lichtbronnen. Elke lichtbron zal een component ambient

licht bevatten (die mogelijk intensiteit nul heeft) en deze component wordt weergegeven door zijn eigen triple $L_a = (i_R^a, i_G^a, i_B^a)$, bijvoorbeeld $(0.2, 0.2, 0)$. Daarnaast zal elk object ook aangeven in welke mate het ambient licht reflecteert, opnieuw op basis van een triple $O_a = (r_R^a, r_G^a, r_B^a)$, bijvoorbeeld $(0.5, 0, 0.5)$. De waardes van O_a geven aan in welke mate het ambient licht wordt weerkaatst door een object. Bijvoorbeeld, de $(0.5, 0, 0.5)$ geeft aan dat het rode en blauwe licht voor de helft wordt weerkaatst en voor de helft wordt geabsorbeerd, terwijl al het groene licht wordt geabsorbeerd. Kortom, indien er wit ambient licht zou invallen op dit object zal het weerkaatste licht een magenta tint hebben (daar magenta gevormd wordt door rood en blauw).

In het algemeen zal de gereflecteerde kleur van het ambiente licht in het RGB model worden weergegeven door $(i_R^a r_R^a, i_G^a r_G^a, i_B^a r_B^a)$ wanneer een object met triple O_a wordt belicht door een licht met triple L_a . We moeten dus enkel voor elke kleurcomponent het product van de intensiteit van het licht met de mate van reflectie nemen. Indien er meerdere lichtbronnen aanwezig zijn dan tellen we de resultaten van alle lichtbronnen eenvoudig bij elkaar op. Bijvoorbeeld indien lichtbron één een ambiente intensiteit heeft van $(0.2, 0.2, 0)$ en de tweede van $(0.1, 0, 0.4)$ en het object heeft $O_a = (0.5, 0, 0.5)$, dan resulteert dit in

$$(0.2 \ 0.5, 0.2 \ 0, 0 \ 0.5) + (0.1 \ 0.5, 0 \ 0, 0.4 \ 0.5) = (0.15, 0, 0.2).$$

Daar het resultaat steeds tussen 0 en 1 gelegen moet zijn, mag de som van de intensiteiten van alle lichtbronnen nooit groter zijn dan één. In ons voorbeeld is deze som slechts $(0.3, 0.2, 0.4)$, wat nog $(0.7, 0.8, 0.6)$ over laat voor het diffuus en specular licht.

Voor het bepalen van de ambiente kleurcomponent van een oppervlak (i.e., driehoek) hoeven we dus helemaal geen rekening te houden met de positie en richting van het oppervlak, noch met de positie of richting van de lichtbronnen, waardoor we de ambiente kleurcomponent van alle driehoeken kunnen bepalen voor het uitvoeren van het Z-buffer algoritme. Voor diffuus en specular licht is dit niet altijd meer het geval.

16.2 Diffuus licht

Diffuus licht verschilt van ambient licht doordat de positie van de lichtbron nu wel een invloed zal hebben op het gereflecteerde licht. Echter, gegeven dat de diffuse component invalt op een oppervlak, dan zal dit invallende licht in alle

richtingen gelijkmatig weerkaatst worden, waardoor de positie van waaruit we kijken nog steeds geen invloed zal hebben op de diffuse kleurcomponent. Evenals bij het ambiente licht heeft elk object zijn eigen triple dat aangeeft hoe diffuus licht wordt weerkaatst, dit noteren we als $O_d = (r_R^d, r_G^d, r_B^d)$ en zal elke lichtbron een diffuse lichtintensiteit hebben dewelke we noteren als $L_d = (i_R^d, i_G^d, i_B^d)$. We gaan een onderscheid maken tussen twee vormen van diffuus licht: bronnen op oneindig en puntbronnen die zich bevinden op een bepaalde locatie in onze 3D-ruimte.

Bronnen op oneindig: Bij de eerste vorm van diffuus licht gaan we doen alsof de lichtbron zich op oneindig bevindt en zal het diffuus licht naast het triple L_d voor de intensiteit, ook door een vector l_d worden gekarakteriseerd. Doorgaans is l_d de genormaliseerde vector die de locatie van de lichtbron met de oorsprong van ons coördinaatsysteem verbindt. Het licht komt dus als het ware uit een bepaalde richting en zal enkel weerkaatst worden door een oppervlak indien dit oppervlak zo gepositioneerd is dat het licht er op invalt. Bijvoorbeeld, indien het licht van boven komt, dan zal de onderzijde van een tafel geen diffuus licht reflecteren. Indien het oppervlak wel juist is georiënteerd, dan zal de hoek tussen het invallende licht en het oppervlak mee de intensiteit bepalen. Valt het licht loodrecht in, dan is de intensiteit het grootste, daar het aantal lichtstralen per oppervlak het grootste is, terwijl licht dat zeer schuin invalt op het oppervlak slechts een heel beperkte reflectie zal hebben, daar er erg weinig lichtstralen per oppervlakte eenheid aanwezig zijn.

Mathematisch gaan we dit als volgt uitdrukken. We berekenen de vector n met lengte één die loodrecht staat op het oppervlak. Merk op dat er zo twee vectoren bestaan en dat we diegene wensen die naar de buitenkant van het object wijst. De richting (w_1, w_2, w_3) van deze vector, in het Eye-coördinaat systeem, hebben we reeds bepaald in het voorgaande hoofdstuk tijdens de berekening van $dzdx$ en $dzdy$ in Sectie 14. Wanneer we de vector (w_1, w_2, w_3) normaliseren (d.i., delen door $(w_1^2 + w_2^2 + w_3^2)^{1/2}$), dan bekomen we één van deze twee vectoren. De vraag is nu hoe we weten of dit de juiste vector is. De vector (w_1, w_2, w_3) werd bekomen door het vectoriële product tussen de vector AB en AC . Dus, indien we de driehoek ABC van buiten het object bekijken en de volgorde van de hoekpunten van de driehoek ABC is tegen de klok in, dan zal dit product resulteren in een vector die naar buiten toe wijst (omwille van de rechterhandregel). Kortom, indien we er steeds voor zorgen

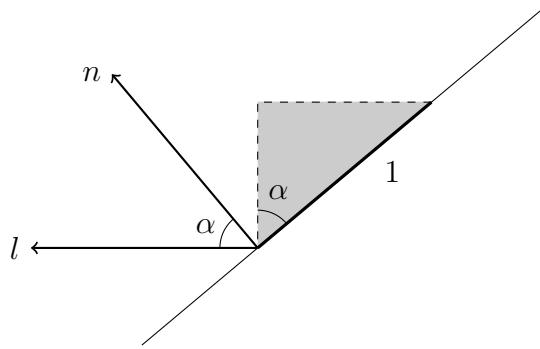


Fig. 31: Formule diffuus licht op oneindig

dat de volgorde van de hoekpunten van alle driehoeken van de buitenkant van het object bekeken tegen de klok in gaan, dan is de gezochte vector n gelijk aan (w_1, w_2, w_3) genormaliseerd.

We berekenen nu het scalaire product tussen de vector $n = (n_1, n_2, n_3)$ en de vector l gelijk aan $-l_d$, met richting (l_1, l_2, l_3) in het Eye-coördinaat systeem, hetwelke gelijk is aan

$$n_1 l_1 + n_2 l_2 + n_3 l_3.$$

Deze waarde geeft de cosinus weer van de hoek α tussen de vector n en l , daar de lengte van l en n gelijk is aan één (merk op, l_d geeft de richting aan waarin het licht schijnt, dus $l = -l_d$ geeft aan vanwaar het licht komt). Indien het licht loodrecht invalt op het oppervlak is deze cosinus gelijk aan één, hoe schuiner het licht invalt hoe kleiner de cosinus. Valt het licht in op de achterzijde van het oppervlak dan is deze cosinus negatief en is er geen diffuse bijdrage van de lichtbron aan het object. Figuur 31 geeft aan dat het aantal lichtstralen per oppervlakte eenheid inderdaad gelijk is aan $\cos(\alpha)$ indien de hoek tussen de normaalvector n en de richting waaruit het licht komt l gelijk is aan α .

Indien de cosinus positief is, dan kunnen we de diffuse kleurcomponent van een object bekomen, door eerst zoals bij het ambiente licht, elke component van O_d met L_d te vermenigvuldigen en vervolgens het resultaat hiervan met de cosinus van de hoek tussen n en l te vermenigvuldigen, of nog

$$(i_R^d r_R^d, i_G^d r_G^d, i_B^d r_B^d) \cos(\alpha),$$



Fig. 32: Diffuus licht op oneindig: $O_a = O_d = (1, 1, 1)$, $L_a^1 = (0, 0, 0)$, $L_a^2 = (0.1, 0.1, 0.1)$ en $L_d^1 = L_d^2 = (0.8, 0.8, 0.8)$.

met $\cos(\alpha) = n_1 l_1 + n_2 l_2 + n_3 l_3$. Indien er meerdere lichtbronnen zijn dan tellen we opnieuw alle bijdrages bij elkaar op. Let wel, elke bron heeft zijn eigen l_d vector en de hoek α moet dus voor elke bron apart bepaald worden.

Een erg belangrijke opmerking hierbij is tevens dat de richting van de vector l dezelfde is voor alle punten die deel uitmaken van een driehoek. Dit geeft aan dat net zoals bij het ambiente licht, de diffuse bijdrage identiek is voor alle punten van de driehoek, wat maakt dat het toevoegen van deze vorm van diffuus licht niet rekenintensief is. In Figuur 32 zien we een voorbeeld van een bol die is belicht met ambient en diffuus licht. In de meest rechtse afbeelding hanteren we twee lichtbronnen, de overige twee laten het effect van elke lichtbron apart zien.

Puntbron: Indien we niet veronderstellen dat het om een lichtbron op oneindig gaat, maar om een puntbron op positie $p = (p_1, p_2, p_3)$ die in alle richtingen schijnt, dan gaan we heel gelijkaardig te werk als in het voorgaande geval. We berekenen opnieuw de vector n als (w_1, w_2, w_3) genormaliseerd en berekenen wederom de cosinus van de hoek tussen n en *een* vector l , dewelke we dan op exact dezelfde manier gebruiken voor het bepalen van de diffuse bijdrage van het licht.

Het grote verschil zit hem nu echter in de keuze van de vector l . Bij een bron op oneindig, was dit de vector met lengte één die wees naar de lichtbron op oneindig en deze was dezelfde voor alle punten op het oppervlak. Voor een puntbron is dit echter de genormaliseerde vector die het punt van het oppervlak waarvoor we de bijdrage willen berekenen, verbindt met de positie p van de lichtbron. Kortom, deze is afhankelijk van het punt dat we kiezen op het oppervlak, daar de richting van de vector hierdoor wordt beïnvloed. We moeten dus voor elke pixel die behoort tot een oppervlak zijn coördinaten (x, y, z) bepalen in het Eye-coördinaat systeem en de vector die

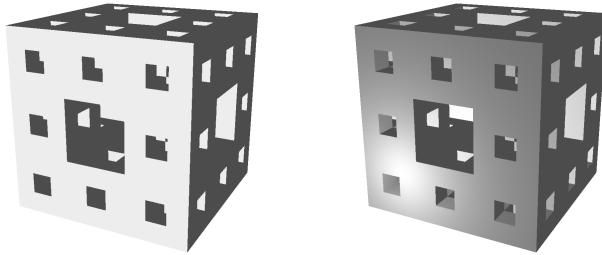


Fig. 33: Diffuus licht op oneindig versus puntbron: $O_a = O_d = (1, 1, 1)$, $L_a = (0.3, 0.3, 0.3)$, $L_d = (0.7, 0.7, 0.7)$ en $p = (1.5, -0.5, -0.5)$.

(x, y, z) verbindt met p in het Eye-coördinaat systeem normaliseren om l te bekomen. Dit kunnen we door gebruik te maken van de vergelijkingen in Sectie 6.2 voor de omzetting van de Eye-coördinaten naar de geprojecteerde coördinaten. We moeten hierbij wel opmerken dat we er voor gezorgd hebben dat het middelpunt van de afbeelding samenviel met het middelpunt van de data, wat betekent dat de pixel nummers niet gelijk hoeven te zijn aan de geprojecteerde coördinaten.

Het mag duidelijk zijn dat puntbronnen een stuk meer rekenwerk vereisen, maar we kunnen ze wel gewoon midden in de afbeelding plaatsen, zoals zal blijken bij een later voorbeeld. In Figuur 33 zien we een eenvoudiger voorbeeld dat het verschil aangeeft tussen een bron op oneindig, waardoor alle oppervlakken uniform van kleur zijn en een puntbron in de nabijheid van een object.

16.3 Specular licht

In tegenstelling tot diffuus licht, waarbij het weerkaatste licht gelijk verspreid wordt in alle richtingen, zal bij specular licht het grootste deel van het invallende licht weerkaatst worden in de reflectierichting r (zie Figuur 34). Dit betekent dat de positie van ons cameraoogpunt nu wel een rol zal spelen bij het bepalen van de spiegelende of speculaire licht component. Net zoals voorheen heeft elk object een triple dat de mate van reflectie weergeeft voor het speculaire licht $O_s = (r_R^s, r_G^s, r_B^s)$ en heeft elke lichtbron een speculaire intensiteit $L_s = (i_R^s, i_G^s, i_B^s)$. De speculaire component wordt dan berekent als

$$(r_R^s i_R^s, r_G^s i_G^s, r_B^s i_B^s) \cos(\beta)^{m_s},$$

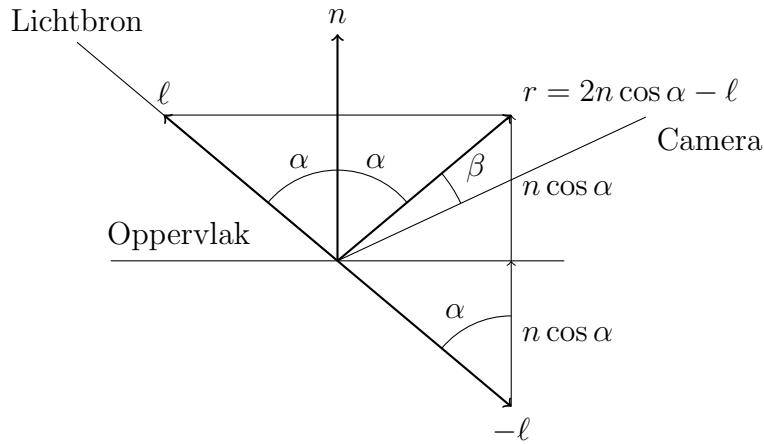


Fig. 34: Berekening van de reflectierichting r

waarbij β de hoek is tussen de reflectievector r en de vector die het punt van het oppervlak waarvoor we de bijdrage willen berekenen, verbindt met het cameraoogpunt. De parameter m_s is een materiaal eigenschap die aangeeft hoe blinkend het materiaal is. Grote waarden van m_s geven aan dat het object harder blinkt, waardoor het weerkaatste licht zich in een nauwere bundel rond r concentreert. Indien de cosinus negatief is, dan is de speculaire bijdrage gelijk aan nul. In Figuur 35 en 36 zien we het resultaat van een bron met een ambiente, diffuse en speculaire component op een bol en een kubus voor verschillende m_s waarden. Hieruit blijkt dat het effect van m_s erg afhankelijk is van de vorm van het object en de invalshoek van de camera. Het gebruik van de factor $\cos(\beta)^{m_s}$ voor de berekening van de speculaire component noemt men Phong reflection.

Om de speculaire reflectie te berekenen moeten we dus r eerst bekomen, de normaal vector n hebben we reeds eerder in detail besproken. Uit Figuur 34 blijkt dat we deze kunnen berekenen door bij $-l$ de vector n op te tellen indien we n herschalen zodat deze lengte $2 \cos(\alpha)$ verkrijgt, met α de hoek tussen n en l , of nog

$$r = 2 \cos(\alpha) n - l.$$

Merk op dat we de $\cos(\alpha)$ reeds berekend hebben bij onze bespreking van diffuse puntbronnen door het scalaire product van l met n te nemen (daar deze beide lengte één hebben).

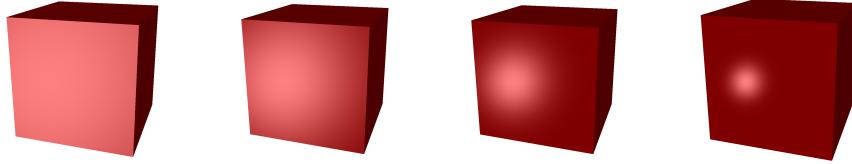


Fig. 35: Impact van m_s : $O_a = O_d = (1, 1, 1)$, $L_a = (0.2, 0, 0)$, $L_d = (0.3, 0, 0)$ en $L_s = (0.5, 0, 0)$, voor $m_s = 2, 10, 50$ en 250

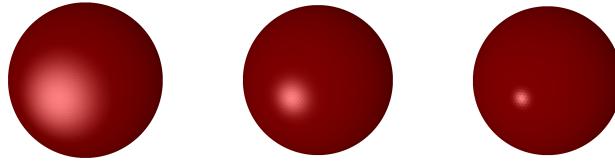


Fig. 36: Impact van m_s : $O_a = O_d = (1, 1, 1)$, $L_a = (0.2, 0, 0)$, $L_d = (0.3, 0, 0)$ en $L_s = (0.5, 0, 0)$, voor $m_s = 2, 10$ en 50

Tot slot kijken we in Figuur 37 nog even naar een voorbeeld waarbij we elke lichtcomponent apart weergegeven wordt, alsook het resultaat van de optelling. In dit geval is $O_a = O_d = O_s = (1, 1, 1)$ en hebben we twee lichtbronnen. De eerste heeft $L_a = (0.3, 0, 0)$, $L_d = (0.2, 0, 0)$ en $L_s = (0.5, 0.5, 0.5)$. Deze bron gebruikt diffuus licht op oneindig en staat op positie $(5, -1, -1)$ wat een lokatie voor het object betreft. De tweede bron heeft $L_a = (0, 0, 0)$, $L_d = (0, 0.6, 0)$ en $L_s = (0, 0.4, 0)$. Het betreft een diffuse puntbron die zich op positie $p = (-0.5, -0.2, 0)$ bevindt, wat een punt achteraan in de Menger spons is. Merk op dat de som van alle groene componenten groter is dan één in dit voorbeeld, wat in principe niet mag. In dit geval zorgt dit echter voor geen problemen daar de groene component in geen enkel punt de waarde van één overschrijdt.

17 Schaduw

In deze sectie geven we aan hoe we schaduw kunnen toevoegen indien we gebruik maken van een Z-buffering algoritme. We gaan ons hierbij beperken tot lichtbronnen die zich buiten de te visualiseren objecten bevinden, net zoals we ook steeds verondersteld hebben dat ons cameraoogpunt zich eveneens niet tussen of in de af te beelden objecten bevond. Verder is het niet enkel de bedoeling om een vlak achter en/of onder de objecten toe te voegen waarop

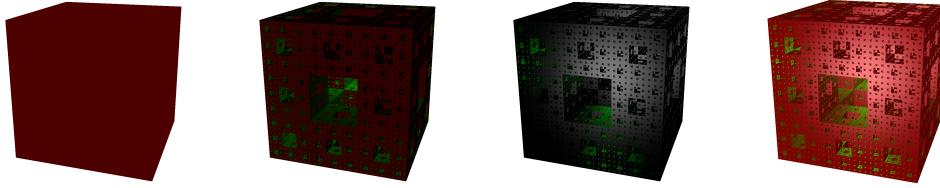


Fig. 37: De ambiente, diffuse en speculaire bijdrage van het licht

een schaduw valt, maar wensen we ook dat de schaduw van een object te zien is op andere objecten, wat iets meer complexiteit met zich meebrengt.

Om schaduw toe te voegen aan een scene die wordt belicht door n lichtbronnen zullen we $n + 1$ maal het Z-buffer algoritme uitvoeren. Gedurende de eerste n keer nemen we telkens de positie van één van de n lichtbronnen als cameraoogpunt. Dit houdt in dat we vanuit elke lichtbron de Z-buffer opstellen die aangeeft wat de (inverse) z-coördinaat is van het dichtsbijgelegen object dat wordt afgebeeld op elke pixel. Na het berekenen van deze n Z-buffers, waar vaak wordt naar verwezen als de *n shadow masks*, voeren we het Z-buffer algoritme zoals steeds uit vanuit het cameraoogpunt.

Wanneer deze $n + 1$ stappen zijn uitgevoerd gaan we, lichtbron per lichtbron en pixel per pixel, bepalen welke kleurcomponenten we moeten toevoegen. Hierbij zullen we voor een punt dat gelegen is in de schaduw van lichtbron L , en dat dus niet zichtbaar is vanuit lichtbron L , enkel de ambiente component toevoegen. Is het punt wel zichtbaar dan voegen we eveneens de diffuse en speculaire component van lichtbron L toe, zoals aangegeven in de voorgaande sectie.

Om te bepalen of een punt P dat wordt geprojecteerd op pixel (x^*, y^*) ook effectief zichtbaar is vanuit een bepaalde lichtbron L , moeten we enkel nagaan of de diepte van P in het coördinaatsysteem met L als cameraoogpunt gelijk is aan de waarde opgeslagen in de juiste pixel in de shadow mask van L . Hiervoor gaan we als volgt te werk. Daar het punt op pixel (x^*, y^*) wordt afgebeeld kennen we de geprojecteerde coördinaten (x', y') , alsook de diepte z_E in het Eye-coördinaatsysteem (aangezien deze is opgeslagen in de Z-buffer entry (x^*, y^*)). Via de relatie

$$x_E = \frac{-z_Ex'}{d} \text{ en } y_E = \frac{-z_Ey'}{d},$$

uit Sectie 6.2, bekomen we dus de coördinaten (x_E, y_E, z_E) van het betreffende punt P in het Eye-coördinaat systeem. Deze laatste kunnen we terug

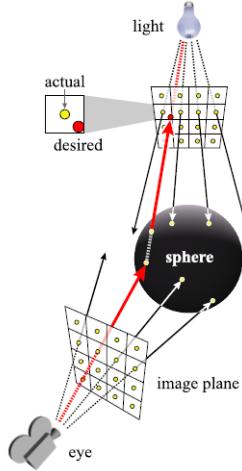


Fig. 38: Z-buffering met schaduwen: is de pixel belicht of niet?

omzetten in de input coördinaten (x, y, z) van het punt P door

$$(x, y, z, 1) = (x_E, y_E, z_E, 1)V^{-1},$$

omwille van de relatie $(x_E, y_E, z_E, 1) = (x, y, z, 1)V$ bekomen in Sectie 6.1 en doordat de matrix V inverteerbaar is. Dit laatste kunnen we als volgt inzien. Uit de hoofdformule van de goniometrie weten we dat $\sin^2 \alpha + \cos^2 \alpha = 1$ (wat overeenstemt met de formule van Pythagoras). Daardoor zullen de determinanten van alle rotatiematrixen uit Sectie 5.3 gelijk zijn aan één. De translatiematrix T uit Sectie 6.1 zijn determinant is eveneens één en V is het product van de twee rotatiematrixen $M_z(\pi/2 + \theta)$ en $M_x(\phi)$ met T . Daar elk van deze matrixen inverteerbaar is en de inverse van een matrix product steeds bestaat indien alle matrixen waaruit het product is opgebouwd inverteerbaar zijn (aangezien $(A_1 A_2 \dots A_n)^{-1} = A_n^{-1} \dots A_2^{-1} A_1^{-1}$), mogen we besluiten dat V steeds inverteerbaar is.

Eens we de coördinaten van (x, y, z) bekomen hebben, zetten we deze om in de coördinaten (x_L, y_L, z_L) van het systeem dat de lichtbron L als oorsprong heeft door

$$(x_L, y_L, z_L, 1) = (x, y, z, 1)V_L,$$

waarbij V_L identiek is aan V behalve dat θ , ϕ en r nu vervangen zijn door θ_L , ϕ_L en r_L , de bolcoördindaten van het positie van lichtbron L . Vervolgens

bepalen we de geprojecteerde coördinaten (x'_L, y'_L) van (x_L, y_L, z_L) via de relatie

$$x'_L = \frac{dx_L}{-z_L} \text{ en } y'_L = \frac{dy_L}{-z_L},$$

zoals aangegeven in Sectie 6.2. Deze laatste stemmen overeen met pixel (x_L^*, y_L^*) van de shadow mask van lichtbron L .

Om na de gaan of punt P in de schaduw licht van lichtbron L ligt lijkt het dus te volstaat om te kijken of z_L , de diepte coördinaat van P vanuit lichtbron L , gelijk is aan de waarde opgeslagen in entry (x_L^*, y_L^*) van de shadow mask van L (op een ϵ , b.v., 10^{-10} , na daar we met floating point getallen werken).

Figuur 38 stelt deze procedure grafisch voor en geeft ook onmiddellijk aan welk probleem zich nog aandient. Namelijk de bekomen coördinaten (x_L, y_L) vallen zelden samen met de exacte (x, y) -waarde van een pixel in de shadow mask. Dit betekent dat zelfs wanneer het punt P zichtbaar is vanuit lichtbron L , de diepte z_L van het punt P niet overeenstemt met de z -waarde in (x_L^*, y_L^*) , daar we hebben afgerond tot de dichtsbijgelegen pixel. Deze afronding heeft ernstige gevolgen daar de $1/z$ -waarde heel sterk kan toe- of afnemen wanneer een oppervlak bijna parallel is aan de kijkrichting. Vandaar dat we niet zomaar kunnen vergelijken met de inhoud van de dichtsbijgelegen pixel, want de afwijking door deze afronding is onbegrensd.

Om een betere diepte te bekomen, waarmee we $1/z_L$ gaan vergelijken, zullen we daarom een interpolatie uitvoeren tussen de vier dieptes van de omliggende pixels A, B, C en D , zoals aangegeven in Figuur 39. We gaan hiervoor steunen op het feit dat de $1/z$ -waarde lineair interpoleert wanneer we gebruik maken van een perspectiefprojectie zoals aangegeven in Sectie 11. Concreet bepalen we eerst α_x en α_y door

$$\alpha_x = x'_L - \lfloor x'_L \rfloor \text{ en } \alpha_y = y'_L - \lfloor y'_L \rfloor.$$

Vervolgens schatten we de diepte van het punt E gelegen tussen A en B dat eveneens x'_L als x -coördinaat heeft, zijnde

$$1/z_E = (1 - \alpha_x)/z_A + \alpha_x/z_B$$

en analoog voor het punt F tussen C en D via

$$1/z_F = (1 - \alpha_x)/z_C + \alpha_x/z_D.$$

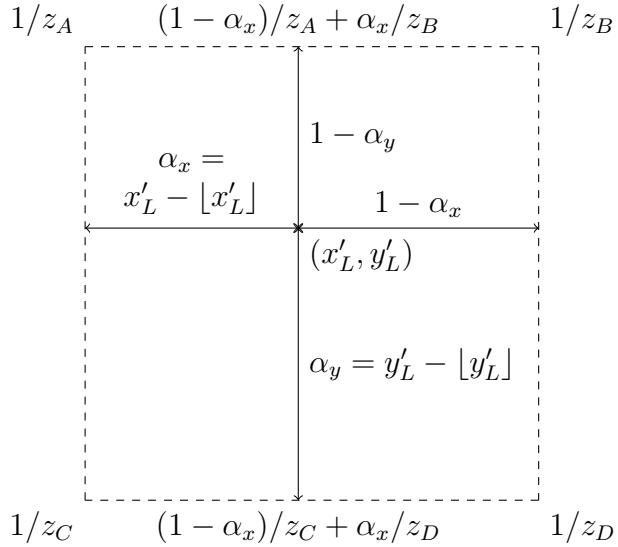


Fig. 39: Interpolatie om de diepte te schatten waarmee we $1/z_L$ wensen te vergelijken

Tot slot kunnen we de diepte waarmee we $1/z_L$ wensen te vergelijken schatten als

$$\alpha_y/z_E + (1 - \alpha_y)/z_F.$$

Wanneer de vier punten A, B, C en D behoren tot dezelfde driehoek (of veelhoek) dan zal deze schatting een exact resultaat opleveren, temmeste wanneer we de 1.0001 waarde uit Sectie 13 verwijderen! Behoren de vier punten tot twee verschillende oppervlakken dan geeft deze schatting een diepte ergens tussen de twee oppervlakken in, waardoor het punt P als zichtbaar beschouwd wordt indien het behoort tot het dichtsbijgelegen van beide oppervlakken en als niet zichtbaar indien het behoort tot het verder gelegen oppervlak (gezien vanuit lichtbron L).

In Figuur 40 zien we het resultaat wanneer we de schaduw aan een afbeelding toevoegen. In het bovenstaande voorbeeld is er gebruik gemaakt van twee lichtbronnen, die zich bevonden op de posities $(3, -3, -2)$, of nog, links voor- en onderaan, en $(-2, 2, 1)$, of nog, rechts achter- en bovenaan. De beide bronnen bestaan uit wit difuus en specular licht met intensiteit 0.4 en 0.6, respectievelijk. Merk op, indien een lichtbron een diffuse component

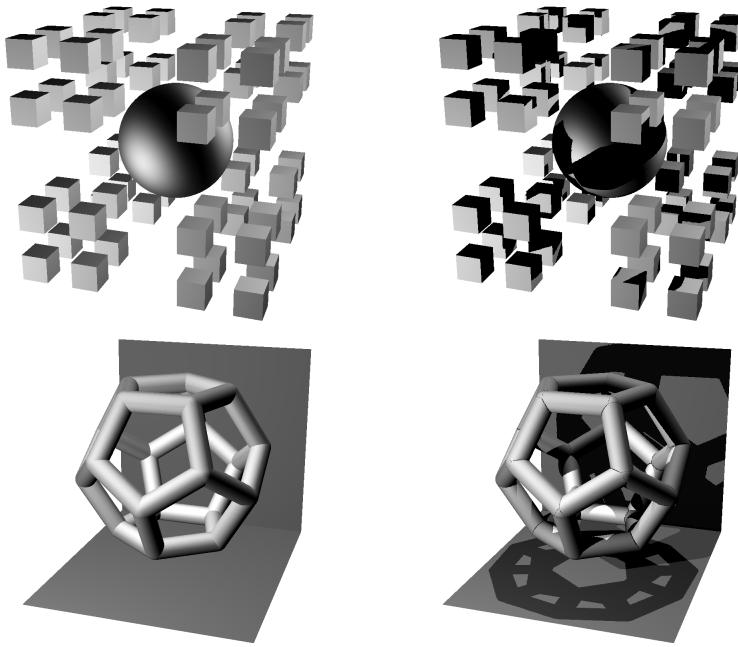


Fig. 40: Voorbeeld van Z-buffering met (rechts) en zonder (links) schaduw

heeft op oneindig, wordt bij het bepalen of een object al dan niet dit diffuus licht ontvangt, dezelfde berekening gemaakt als wanneer het een puntbron zou zijn op positie p . Indien we bij het bepalen van de schaduw de bron effectief op oneindig wensen, dan moeten we naast de perspectiefprojectie deze lichtbron als oorsprong eveneens een parallelprojectie uitvoeren.

Deze techniek kan in sommige gevallen wel tot een slecht resultaat leiden. Meer bepaald wanneer de invalshoek van de lichtbron zodanig is dat een hele reeks naast elkaar gelegen pixels van onze afbeelding allemaal terecht komen tussen dezelfde vier pixels van de shadow mask. In zulk een geval kan de schaduw een hoekige vorm aannemen. Een eenvoudige manier om dit te demonstreren bestaat erin de resolutie van onze shadow mask, die typisch uit evenveel pixels bestaat als de Z-buffer die we vanuit het oogpunt genereren, drastisch te verlagen. In de rechtse afbeelding van Figuur 41 hebben we de resolutie van de shadow mask met een factor 16 verkleind (waardoor deze slechts 50×50 is, terwijl de figuur links met een 800×800 mask werd gemaakt, wat overeenstemt met de resolutie van de afbeelding). Dit effect wordt doorgaans aangeduid met de term *aliasing*.

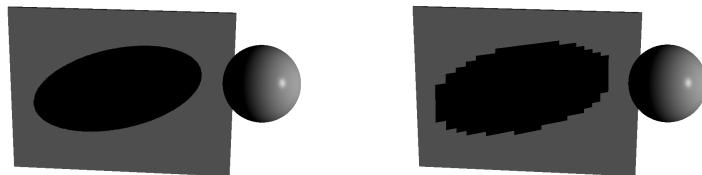


Fig. 41: Impact van de resolutie van de schadow mask op de kwaliteit van de schaduw (rechts 50×50 , links 800×800)

18 Texture mapping

Tot slot gaan we eens kijken hoe we oppervlakken in onze 3D-ruimte kunnen voorzien van een bepaalde textuur. Zo willen we bijvoorbeeld de indruk wekken dat sommige objecten van hout gemaakt zijn, andere van metaal of steen. In plaats van te trachten deze detailstructuur te modelleren met een set van erg kleine driehoeken, wat een uiterst trage oplossing zou zijn, zullen we een 2D-afbeelding gebruiken van de gewenste textuur en deze mappen op het oppervlak in kwestie. We hoeven dus enkel over een afbeelding van de textuur te beschikken en kunnen deze methode dan ook gebruiken voor gelijk welke afbeelding, bijvoorbeeld om een schilderij in een kader te plaatsen. Het mappen van afbeeldingen op objecten in onze 3D-ruimte wordt doorgaans aangeduid met de term *texture mapping*.

De voornaamste stap bij het toevoegen van een textuur is het bepalen van een eenvoudige methode die de punten in onze 3D-ruimte (die zich bevinden op het oppervlak dat we wensen te bekleden) omzet naar de pixelnummers waaruit onze afbeelding is opgebouwd. De pixels waaruit deze afbeelding bestaat zullen we *texture elements* of kortweg *texels* noemen. We zoeken dus een snelle methode om (x, y, z) -coördinaten gelegen op een oppervlak om te zetten in een texelnummer (u, v) . In de volgende sectie kijken we hoe we zulk een mapping eenvoudig kunnen uitvoeren indien het oppervlak dat we van een textuur wensen te voorzien vlak is.

18.1 Vlakke oppervlakken

Indien het oppervlak dat we wensen te bekleden in een vlak gelegen is, bijvoorbeeld het betreft een zijde van een kubus, menger spons of de cirkelvormige zijde van een cylinder, volstaat het een rechthoek in de 3D-ruimte te definiëren die het oppervlak omvat. Wanneer het gaat om een zijde van een kubus, dan kunnen we gewoon het overeenkomstige vierkant nemen. Voor de cirkelvormige zijde van een cylinder, nemen we een vierkant waarvan de lengte van de zijdes gelijk is aan de diameter van de cirkel. In dit geval zullen we enkel de texels van de ingeschreven cirkel van de afbeelding gebruiken. Het oppervlak mag dus gerust opgebouwd zijn uit een hele set van driehoeken, zoals eveneens het geval is bij de zijwand van een menger spons, zolang deze maar in een vlak gelegen zijn.

We starten bijgevolg met het specifiëren van de rechthoek die het oppervlak omvat. Dit doen we door middel van drie vectoren: $p = (p_x, p_y, p_z)$, $a = (a_x, a_y, a_z)$ en $b = (b_x, b_y, b_z)$. De eerste vector p geeft de positie aan van de linkerbenedenhoek van de rechthoek, wat het hoekpunt is dat overeenstemt met de linkerbenedenhoek van onze textuur afbeelding. De vector a start in p en eindigt in de rechterbenedenhoek, terwijl b eveneens in p start, maar eindigt in de linkerbovenhoek. Wanneer we nu een punt (x, y, z) nemen dat gelegen is op de rechthoek die ons oppervlak omvat, dan kunnen we dit dus op een unieke manier schrijven als

$$(x, y, z) = (p_x, p_y, p_z) + u(a_x, a_y, a_z) + v(b_x, b_y, b_z),$$

met u en v gelegen tussen 0 en 1. Wanneer de afbeelding van de textuur uit 200 op 300 pixels bestaat, dan betekent dit dat we texel nummer $(1 + 199u, 1 + 299v)$ zullen gebruiken voor het punt (x, y, z) . We moeten $199u$ en $299v$ natuurlijk wel afronden naar een geheel getal, wat neerkomt op het gebruiken van de RGB kleur van de dichtsbijgelegen texel.

Gegeven (x, y, z) kunnen we deze unieke (u, v) als volgt bepalen. Ten eerste zien we dat

$$(x - p_x, y - p_y, z - p_z) = (u, v) \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}.$$

Dit stelsel heeft drie vergelijkingen en twee onbekenden, echter daar (x, y, z) in een vlak gelegen is, volstaat het twee van de drie coördinaten vast te leggen. We moeten echter voorzichtig zijn bij de keuze van deze drie coördinaten. Bijvoorbeeld voor het vlak met vergelijking $x + y = 1$, volstaat het niet



Fig. 42: Enkele voorbeelden van een menger spons met een drietal textures

x en y vast te leggen, maar wel x en z (of y en z). Om te weten welke twee coördinaten volstaan, hoeven we enkel te kijken welke van de drie onderstaande matrices een determinant heeft die (numeriek) verschilt van nul (daar dit impliceert dat de matrix inverteerbaar is)

$$\begin{bmatrix} a_x & a_y \\ b_x & b_y \end{bmatrix} \text{ of } \begin{bmatrix} a_y & a_z \\ b_y & b_z \end{bmatrix} \text{ of } \begin{bmatrix} a_x & a_z \\ b_x & b_z \end{bmatrix}.$$

Indien bijvoorbeeld de determinant van de tweede matrix verschilt van nul, dan is (u, v) gelijk aan

$$(u, v) = (y - p_y, z - p_z) \begin{bmatrix} a_y & a_z \\ b_y & b_z \end{bmatrix}^{-1}.$$

Indien meer dan één determinant verschilt van nul, dan leveren deze allemaal dezelfde oplossing (u, v) op.

We gaan nu tijdens het uitvoeren van het Z -buffer algoritme voor elke pixel (x^*, y^*) gelegen op een driehoek (die deel uitmaakt van een oppervlak dat nadien van textuur wordt voorzien), naast de $1/z$ -diepte ook het nummer van de driehoek opslaan die het dichtsbijgelegen is (in plaats van de RGB kleur). Na het uitvoeren van het Z -buffer algoritme vervangen we deze nummers door de RGB kleur van de juiste texel. We doen dit precies zoals in Sectie 17, door de pixel nummer (x^*, y^*) om te zetten in de coördinaten (x, y, z) van het punt P dat werd geprojecteerd op deze pixel. Vervolgens bepalen we de texel via de bekomen oplossing voor (u, v) .

Aangezien we vaak verschillende texturen wensen te gebruiken in een enkele afbeelding en we tevens ook eenzelfde textuur willen hergebruiken voor verschillende oppervlakken, maken we eveneens gebruik van volgende tabellen/informatie. Ten eerste houden we voor alle rechthoeken die een te-

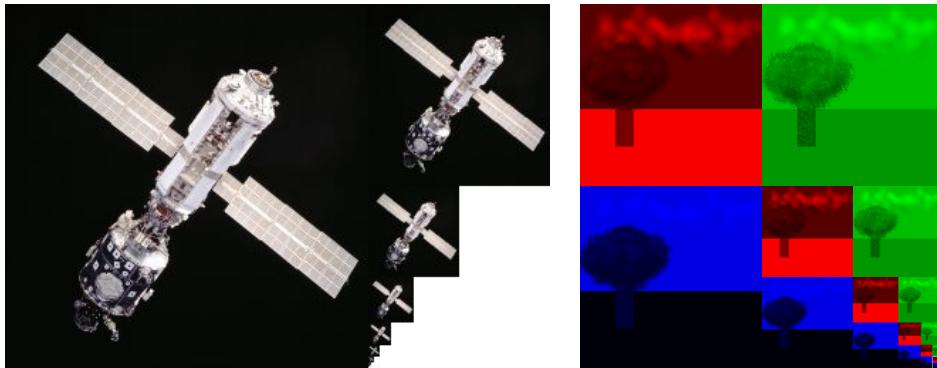


Fig. 43: Mipmapping

bekleden oppervlak omvatten de vectoren p , a en b bij. Verder gebruiken we een tabel die het nummer van de driehoek als index gebruikt en die aangeeft in welke rechthoek de driehoek gelegen is. Tot slot hebben we eveneens een tabel die aangeeft welke textuur afbeelding we associëren met elke rechthoek (daar sommige rechthoeken gebruik zullen maken van dezelfde afbeelding, die we dan ook maar één maal in het geheugen opslaan). De textuur afbeeldingen zelf moeten we natuurlijk eveneens bijhouden.

In Figuur 42 zien we een voorbeeld van een menger spons die bekleed werd met verschillende texturen. In de meest rechtse afbeelding hebben we naast de textuur eveneens gebruik gemaakt van de verschillende soorten belichting besproken in Sectie 16.

Om een goed resultaat te bekomen moeten we er wel steeds voor zorgen dat het aantal texels in de textuur vergelijkbaar is met het aantal gebruikte pixels in de afbeelding. Indien de textuur een te hoge resolutie zou hebben, dan kunnen we de afbeelding best eerst even opslaan in een lagere resolutie. In de praktijk zal men gebruik maken van *mipmapping* om te vermijden dat men tijdens het renderen van de afbeelding eerst nog de resolutie van een afbeelding moet gaan verlagen. Mipmapping houdt in dat we de textuur gaan opslaan met verschillende resoluties door gebruik te maken van een enkele afbeelding (zie Figuur 43), waarbij de geschikte resolutie op een dynamische manier wordt gekozen. Elke afbeelding in de reeks afbeeldingen bevat 4 keer minder texels dan de voorgaande, dit maakt dat de hoeveelheid extra

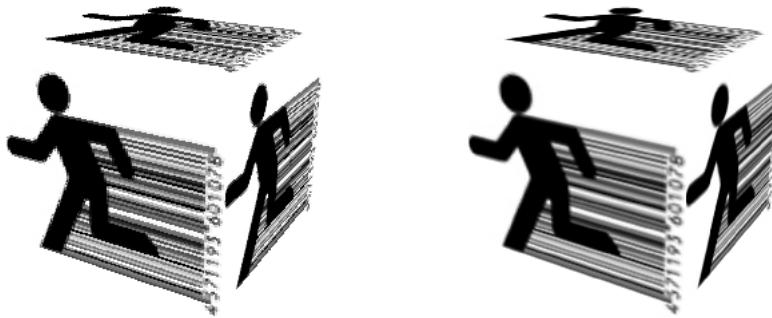


Fig. 44: Links: een texture map met te lage resolutie, rechts: dezelfde texture map na interpolatie

geheugen die hiervoor nodig is gelijk is aan

$$\sum_{n=1}^{\infty} \frac{1}{4^n}.$$

Drie maal deze som kunnen we schrijven als

$$\sum_{n=1}^{\infty} \frac{3}{4^n} = \sum_{n=1}^{\infty} \left(\frac{4}{4^n} - \frac{1}{4^n} \right) = \sum_{n=1}^{\infty} \left(\frac{1}{4^{n-1}} - \frac{1}{4^n} \right) = 1.$$

Met andere woorden, er is 33% meer geheugen vereist bij mipmapping. Dit is trouwens ook direct duidelijk wanneer we kijken naar de rechtse afbeelding in Figuur 43.

Is de resolutie te laag, dan zullen tal van pixels worden afgebeeld op dezelfde texel en dat geeft aanleiding tot een afbeelding van slechte kwaliteit, dewelke vergelijkbaar is met hetgeen we zagen bij onze schaduwen indien de shadow mask een te lage resolutie had. In dit geval kunnen we de kwaliteit verbeteren door de RGB kleur van een pixel te bepalen via een lineaire interpolatie tussen de vier omliggende texels. Daar de interpolatie voor de verschillende pixels met dezelfde dichtsbijgelegen texel doorgaans wel in een andere kleur resulteren (tenzij alle omliggende texels dezelfde kleur hebben), verbetert dit de kwaliteit van de afbeelding een beetje. We zien hiervan een voorbeeld in Figuur 44. Let wel, de interpolatie verhoogt de rekentijd aanzienlijk, vandaar dat een texture met voldoende hoge resolutie verkiesbaar is, want dit gaat sneller en geeft toch nog steeds het beste resultaat.

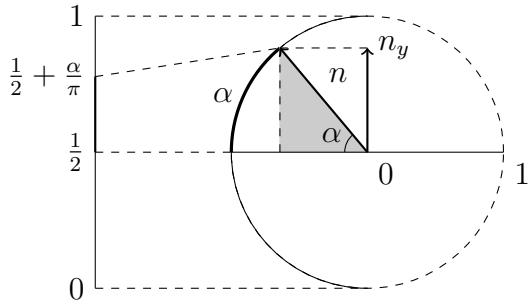


Fig. 45: Texture mapping op een bol: formules

18.2 Bolvormige oppervlakken

In deze sectie lichten we een mogelijke oplossing toe om een bolvormig lichaam van een textuur te voorzien. Deze methode zal enkel de zichtbare zijde van het lichaam bekleden met een textuur en is dus niet echt geschikt indien we dit lichaam wensen te roteren in een aantal opeenvolgende afbeeldingen. Net zoals in de voorgaande sectie zouden we een mapping kunnen maken die alle punten (x, y, z) gelegen op het bolvormige lichaam afbeeldt op een texel.

Echter, voor elke pixel die overeenstemt met een punt P op het lichaam, gaan we enkel de coördinaten (x_E, y_E, z_E) van P in het Eye-coördinaatsysteem bepalen. Daarna berekenen we de vector die het middelpunt O van de bol verbindt met P . Merk op, we geven naast de te hanteren textuur, dus eveneens het middelpunt $O = (O_x, O_y, O_z)$ van het bolvormige lichaam op (en zetten de coördinaten ervan om in het Eye-coördinaatsysteem). Indien we deze vector normalizeren bekomen we de vector $n = (n_x, n_y, n_z)$ die de normaalvector van de bol in het punt P vormt. Met behulp van deze vector n bepalen we de u en v , gelegen tussen 0 en 1, eenvoudig door

$$u = \sin^{-1}(n_x)/\pi + 0.5 \text{ en } v = \sin^{-1}(n_y)/\pi + 0.5.$$

Deze vergelijkingen kunnen we eenvoudig begrijpen door te kijken naar Figuur 45. De lengte van het cirkelsegment in de figuur is gelijk aan $|\alpha|$ (aangezien de volledige cirkelomtrek lengte 2π heeft). De grijze rechthoekige driehoek geeft aan dat $\alpha = \sin^{-1}(n_y)$. Daar α gelegen is tussen $-\pi/2$ en $\pi/2$, levert delen door π een getal op tussen $-1/2$ en $1/2$, met als gevolg dat u en v gelegen zijn tussen 0 en 1. Voor een afbeelding van 200 op 300 pixels

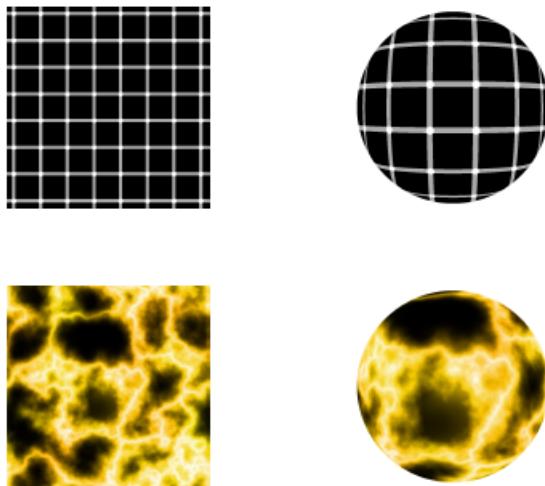


Fig. 46: Texture mapping op een bol

betekent dit wederom dat we texel $(1 + 199u, 1 + 299v)$ zullen hanteren voor de betreffende pixel. Het resultaat voor twee verschillende texturen zien we in Figuur 46.

18.3 Willekeurige oppervlakken

Uitgaande van de twee voorgaande secties lijkt het of we voor objecten met verschillende vormen steeds een andere mapping moeten construeren. Om een perfect resultaat te krijgen is dit in zekere zin het geval. Echter, dit zou het toevoegen van texturen aan een scene die is opgebouwd uit tal van objecten niet alleen erg complex maken, maar ook vrij traag daar sommige van deze mappings veel rekentijd zouden vergen.

Een alternatieve aanpak bestaat erin terug te vallen op de methode uit Sectie 18.1. We definiëren eveneens een rechthoek in de ruimte, gebruik makend van een punt p en twee vectoren a en b , die onze textuur bevat, maar daar het oppervlak niet vlak is, omvat deze alle punten P op het oppervlak niet. Laat V het vlak zijn waarin deze rechthoek gelegen is en P' de projectie van P op dit vlak. Indien we er nu voor zorgen dat deze rechthoek alle geprojecteerde punten P' bevat, dan kunnen we identiek dezelfde methode hanteren als in Sectie 18.1, door de texel die met P' overeenstemt toe te kennen aan P . We gaan hiervoor als volgt te werk.

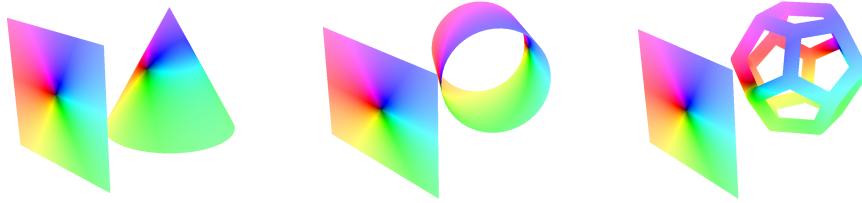


Fig. 47: Texture mapping op een willekeurig lichaam

Eerst bepalen we opnieuw de coördinaten (x, y, z) van P . Laat $c = (c_x, c_y, c_z)$ gelijk zijn aan het vectoriële product tussen a en b , dat is

$$\begin{aligned} c_x &= a_y b_z - a_z b_y, \\ c_y &= a_z b_x - a_x b_z, \\ c_z &= a_x b_y - a_y b_x. \end{aligned}$$

Vervolgens gaan we de coördinaten (u, v, w) van P bepalen in het assenstelsel met $p = (p_x, p_y, p_z)$ als oorspong en met de vectoren a , b en c als ons assenstelsel, of nog, we zoeken u , v en w zodat

$$(x, y, z) = (p_x, p_y, p_z) + u(a_x, a_y, a_z) + v(b_x, b_y, b_z) + w(c_x, c_y, c_z),$$

wat we kunnen schrijven in matrix notatie als

$$(x - p_x, y - p_y, z - p_z) \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}^{-1} = (u, v, w).$$

Wanneer de afbeelding van de textuur uit 200 op 300 pixels bestaat, dan betekent dit dat we texel nummer $(1 + 199u, 1 + 299v)$ zullen gebruiken voor het punt (x, y, z) .

In Figuur 47 zien we een voorbeeld voor een kegelvormige lichaam, de mantel van een cylinder en voor een geraamte van een dodecahedron. In de gehanteerde afbeelding is eveneens de gebruikte rechthoek weergegeven ter verduidelijking, deze maakt natuurlijk doorgaans helemaal geen deel uit van de afbeelding. Tot slot zien we in Figuur 48 nog een voorbeeld van een afbeelding die alle besproken technieken uit dit hoofdstuk combineert.

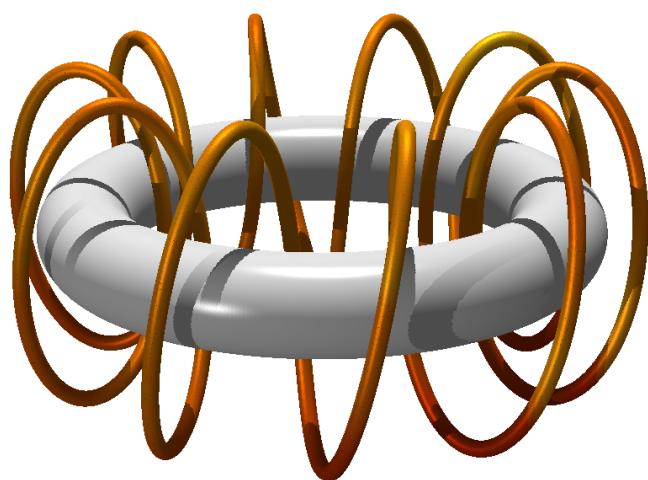


Fig. 48: Voorbeeld afbeelding met verschillende types licht, schaduw en texture mapping