DHZ-3D

Maarten Fokkinga en Jan Kuper
Vakgroep SETI

Versie van 7 november 1995

3D plaatjes hebben een heel regelmatige structuur. We construeren een eenvoudig programma waarmee gemakkelijk *DoeHetZelf 3D* plaatjes gemaakt kunnen worden.

Inleiding. Iedereen kent ze wel, die 3D plaatjes. En als je ze nog niet kent, is het tijd om even een boekhandel binnen te lopen. Er zijn al veel boeken op dit gebied, en je kunt voor weinig geld ook al prentbriefkaarten krijgen met een 3D plaatje er op. Ons doel is hier om de structuur van 3D plaatjes te analyseren, en vervolgens een programma te construeren waarmee je eenvoudig zelf 3D plaatjes kunt maken. We hebben al heel wat mensen (kinderen, moeders, collega's en onszelf) gelukkig gemaakt met een 3D plaatje waarin de eigen initialen in drie dimensies te zien zijn.

De input voor het programma is een plaatje met cijfers. De output is een 3D plaatje waarbij de gebieden met nullen als achtergrond gezien worden, en de gebieden met enen één niveau meer naar de voorgrond, en de gebieden met tweeën twee niveaus meer naar de voorgrond, enzovoorts. In het programma staan oneindig lange lijsten genoteerd (die worden natuurlijk maar voor een eindig deel uitgerekend). Hierdoor kunnen we heel gemakkelijk allerlei herhalingen uitdrukken, zonder ons direct te bekommeren om het *aantal* herhalingen.

Structuur van een 3D plaatje

We willen de structuur van een 3D plaatje achterhalen. Dat doen we in eerste instantie door een soort gedachte-experiment. We denken ons in dat jij iets ziet iets; wat je ziet noemen we R (van "Reality"). We zullen een plaat construeren van R. Wanneer je net zo naar die plaat kijkt als naar R, dan krijg je vrijwel dezelfde beelden op je netvliezen als bij het kijken naar R, en wordt je dus dezelfde diepte gewaar. Het zal blijken dat de "Virtual Reality", gezien in de plaat, niet exact gelijk is aan de Reality R, maar wel dezelfde *diepte-vorm* heeft. Het *uiterlijk* van de Virtual Reality zal uit een soort behang bestaan met repeterende patronen.

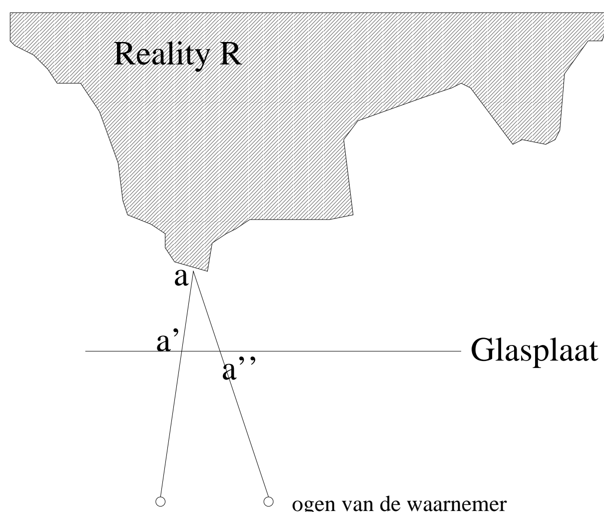
Pixels. Een *pixel* is een elementair deeltje van een tekening of afbeelding. Bij een zwart-wit raster-foto is een pixel een zwarte of witte stip. Bij een kleurenfoto is een pixel een gekleurde stip. Bij een "tekening" geproduceerd met een type-machine, of computer-printer, is een pixel één van de karakters `ABC...XYZ!@#$%^&*()+?`. We zullen daarom spreken van de *positie* en de *kleur* (of de vorm of het uiterlijk) van een pixel. Als twee pixels samenvallen, dezelfde positie hebben, dan zijn ze hetzelfde pixel, en hebben ze uiteraard ook dezelfde kleur. (In een niet-gediscretiseerde, continue, tekening is een pixel één punt, of het binnengebied van een willekeurig kleine cirkel.)

Constructiestap 1. De constructie begint simpel. Plaats een glasplaat tussen jou en reality R in, evenwijdig aan je ogen. De positie van jou, de glasplaat en reality R blijft gedurende het hele experiment hetzelfde. Uiteindelijk zal de glasplaat de 3D plaat worden.

Houd nu alleen je linkeroog open, en teken op de glasplaat precies wat je ziet. Op de glasplaat verschijnt dus een beeld R'. Als je nu alleen met je linkeroog kijkt, maakt het niets uit of je beeld R' bekijkt of reality R: je ziet hetzelfde.

Houd nu je rechteroog open en je linker dicht, en denk je in dat je het beeld R' niet ziet. Teken wederom op de glasplaat wat je ziet, zeg op de andere kant. Op de glasplaat verschijnt dus een beeld R''. Als je nu alleen met je rechteroog kijkt, maakt het niets uit of je beeld R'' bekijkt of reality R: je ziet hetzelfde.

Hieronder is de situatie weergegeven in een bovenaanzicht voor één punt a van R.



Omdat pixels a' en a'' beide het beeld zijn van a, hebben ze dezelfde kleur.

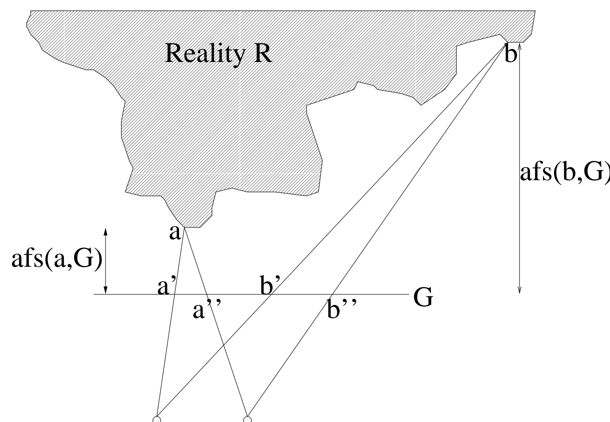
> *Terzijde.* Stel dat je R' groen getekend hebt en R'' rood. (Dus dan hebben a' en a'' niet meer dezelfde kleur.) Stel voorts dat je met je linkeroog alleen groene dingen ziet en je rechteroog alleen rode. Dat is gemakkelijk met een gekleurd brilletje te bereiken. Dan kan je de (glas)plaat met beide ogen tegelijk bekijken op exact dezelfde manier als waarop je in werkelijkheid R bekeek. Je moet je ogen dus niet richten op de plaat, maar er achter, op de virtuele R! (Opmerkelijk is dat je ogen zich toch, automatisch,

op de juiste afstand scherp stellen.) Dus je ziet de oorspronkelijke R weer terug, mét alle diepte-effecten, zij het dat alle kleuren nu louter een menging van groen en rood zijn. Dit soort plaatjes zijn al heel lang bekend.

Constructiestap 2. Hierboven hebben we gesuggereerd dat beeld R' aan de ene kant van de glasplaat getekend kan worden, en beeld R'' aan de andere kant. Maar we willen zowel R' alsook R'' in één plaat, de 3D plaat, hebben. Beschouw nu eens een pixel a'' van R'', dat samenvalt met een pixel b' van R'. Omdat ze samenvallen, zijn ze hetzelfde pixel: $a'' = b'$, en omdat a' en a'' dezelfde kleur hebben, en ook b' en b'', moet je dus de *verschillende* punten a en b van R met pixels van *dezelfde kleur* afbeelden op de (glas)plaat, alsof in werkelijkheid punten a en b dezelfde kleur hebben. We noemen dit: *unificatie* van a en b. Laten we aannemen dat je dit overal op de (glas)plaat zo gedaan hebt: alle pixels van R' en R'' die samenvallen zijn van dezelfde kleur, geünificeerd. (We zullen straks zien dat niet noodzakelijk álle pixels op de plaat van eenzelfde kleur zijn!) Dan zijn R' en R'' nog steeds getrouwe afbeeldingen van R behalve dat de kleuren niet meer helemaal kloppen. En, wanneer je nu de (glas)plaat met beide ogen tegelijk bekijkt (met de ogen gericht op R erachter), ziet het linkeroog nog steeds R', het rechteroog R'', en wordt jij dus een virtual reality R gewaar.

Deze "constructie" van een 3D plaat geeft nog weinig houvast om ook daadwerkelijk een 3D plaat te construeren. Hoe komt het dat we diepte zien? Welke punten moeten allemaal geünificeerd worden? Hoe kunnen we daadwerkelijk een 3D plaat maken, in plaats van het gedachte-experiment hierboven? Laten we daartoe het diepte-effect en de unificatie nog eens nader analyseren, en dan besluiten met een voorbeeld.

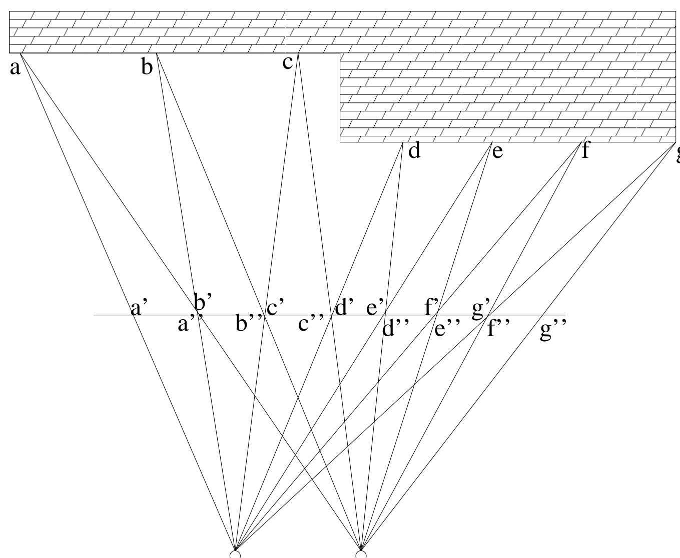
Het diepte effect. Beschouw een punt a van reality R, en de beelden a' en a'' ervan. De afstand tussen a' en a'' hangt uitsluitend af van de afstand van punt a tot de glasplaat, en omgekeerd (bij de gegeven vaste positie van jou, de glasplaat en R). Dit is met eenvoudige meetkunde te bewijzen. Dus van een punt a dicht bij de glasplaat liggen de beelden a', a'' dicht bij elkaar. Van een punt b ver van de glasplaat liggen de beelden b', b'' ver van elkaar. Punten op gelijke afstand tot de glasplaat krijgen beelden met gelijke afstand:



Het ligt voor de hand om te veronderstellen dat het precies de onderlinge afstanden zijn van bij-elkaar horende beeldpunten die bij het bekijken voor het diepte-effect zorgen. Deze veronderstelling blijkt te kloppen met het feit dat we inderdaad diepte zien in 3D plaatjes. (We spreken ons verder niet uit over de manier waarop dat diepte-effect tot stand komt; het gebeurt ergens in het traject van ogenstand en netvliesbeelden via hersenen tot gewaarwording. De bolling van de ogen, voor de scherpstelling, doet niet of minder terzake.)

Unificatie. Laat a en b punten zijn van R die geünificeerd worden: a' en b'' vallen samen, of a'' en b'. Er geldt dan dat a', a'', b', b'' op één lijn liggen, evenwijdig aan de lijn door beide ogen; laten we dit *horizontaal* noemen. Dit is weer met eenvoudige meetkunde te bewijzen. Dus alle punten die via-via met elkaar geünificeerd worden, liggen in de (glas)plaat op een horizontale lijn. Het gevolg is dat je, in een 3D plaat, op iedere horizontale lijn vele malen eenzelfde pixel zult aantreffen!

Een voorbeeld. Laat R een muur zijn met een verspringing erin. We beschouwen punten $a, b, \ldots, f, g$ in de muur. De punten zijn zo gekozen dat a'' samenvalt met b', b'' samenvalt met c', enzovoorts:



De afstanden tussen de beelden van a, b, c is iets groter dan die tussen de beelden van d, e, f, g, want a, b, c liggen verder weg. Al deze pixels a', $b'(=a'')$, $c'(=b'')$, ... hebben dezelfde kleur. Kiezen we type-machine letters `VWXYZ` als pixels, dan zijn $a', \ldots, g'$ allemaal dezelfde letter. Inderdaad, het volgende plaatje is een 3D plaatje van punten $a, \ldots, g$ van de muur. Om het *makkelijk* in 3D te zien moet je het zoveel vergroten dat de afstand tussen de eerste twee `V`'s ongeveer 3 cm is, en moet je het papier op ongeveer 25 cm van je ogen houden en je ogen richten op iets dat ongeveer 25 cm achter het papier is:

```
V          V          V          V        V        V        V
```

Een dergelijke andere reeks punten kan met een ander pixel getekend worden, zeg `W`:

```
V W        V W        V W        V W      V W      V W      V W
```

We zien dat vlakken evenwijdig aan de (glas)plaat met repeterende patronen worden afgebeeld. De patroonlengte wordt korter wanneer het vlak naar voren springt. Wanneer alle punten van de muur met de grove pixels `VWXYZ` worden afgebeeld, moeten er bij de verspringing in de muur, dus bij de verkorting van de patroonlengte, sommige pixels verdwijnen (in het voorbeeld hieronder: `Z`). Dat zijn de pixels van punten uit de muur die nog wel door het ene oog gezien worden en niet meer door het andere oog:

```
V W X Y Z V W X Y Z V W X Y Z V W X Y V W X Y V W X Y V W X Y
```

Op dit principe is ons programma hieronder gebaseerd: we vullen iedere lijn met een repeterend patroon, met weglatingen en toevoegingen van letters waar we een verspringing in de diepte willen hebben. Bij een overgang van een langer naar een korter patroon, of omgekeerd, valt het kortere patroon samen met een eindstuk van het langere patroon.

Het programma

We zullen nu een Miranda programma ontwikkelen waarmee 3D plaatjes geproduceerd worden. Er zijn nauwelijks wijzigingen nodig om het in Gofer of Amanda om te zetten; die zijn beide vrij verkrijgbaar en draaien op PCs. Herinner je de volgende standaard functies:

```
map      -- bv: map f [a,b,c ...] = [f a, f b, f c  ...]
repeat   -- bv: repeat "abc" = ["abc","abc","abc","abc","abc","abc",...
concat   -- bv: concat ["abc","defg" ...] = "abcdefg..."
take     -- bv: take 3 "abcde" = "abc"
drop     -- bv: drop 3 "abcde" = "de"
zip2     -- bv: zip2 [x,y,z...] [u,v,w...] = [(x,u), (y,v), (z,w) ...]
reverse  -- bv: reverse "abcde" = "edcba"
#        -- bv: # "abcde" = 5
```

Herinner je ook dat "abcd" een korte notatie is voor ['a','b','c','d','e'].

Laten we maar beginnen met een gewenst figuur te definiëren:

```
> figuur =
>   ["000000000000000000000000000000000000000000",
>   "000000000000000000000000000000000000000000",
>   "000000000000000000000000000000000000000000",
>   "000000000000000000001100000000000000000000",
>   "000000000000000000011110000000000000000000",
>   "000000000000000000111111000000000000000000",
>   "000000000000000001111111100000000000000000",
>   "000000000000000011111111110000000000000000",
>   "000000000000000000000000000000000000000000",
>   "000000000000000000000000000000000000000000",
>   "000000000000000000000000000000000000000000"]
```

Het 3D plaatje moet dus een vlakke achtergrond te zien geven waarin een driehoek naar voren springt. Om het intikken van `figuur` te vergemakkelijken, zullen we geen gebruik maken van het feit dat alle rijen even lang zijn. We kiezen ervoor om één vast patroon te gebruiken om het 3D plaatje te maken:

```
> patroon = "watzieniktoch"
```

Elk ander patroon van ongeveer 8 á 13 symbolen is ook goed, bijvoorbeeld "**$*$$*$**$$". Op iedere regel zullen we het patroon ontrollen, en bij een overgang van 0 naar 1 (van links naar rechts) een letter uit het patroon weglaten, en bij een overgang van 1 naar 0 weer een letter toevoegen. Algemener, bij een overgang van m naar n verlengen we het patroon met $m - n$ letters. Daarom maken we van `figuur` eerst een lijst van getallijsten:

```
> figuur' = map (map f) figuur
>           where f c = code c - code '0'
```

Bijvoorbeeld, de middelste getallijst in `figuur'` is [0,...,0,0,1,1,1,1,1,1,0,0,...,0]. Voor de grap laten we iedere regel met een andere rotatie van `patroon` beginnen:

```
> cycle        =  concat . repeat
> randomrij    =  cycle [0,2,7,3,11,5,9,4,8,1,6,10]
> rotatie xs i =  drop i xs ++ take i xs
> patronen     =  map (rotatie patroon) randomrij
```

Dus `cycle [1,2,3]` = [1,2,3,1,2,3,1,2...], en `patronen` is de oneindige rij:

```
  ["watzieniktoch","tzieniktochwa","iktochwatzien","zieniktochwat",...
```

We koppelen nu iedere regel uit `figuur'` aan een van de patronen, en maken daaruit één regel van het plaatje:

```
> plaatje      =  map mkregel (zip2 patronen figuur')
```

Rest ons alleen nog om `mkregel` te definiëren:

```
  mkregel (patr, ds) = ...
```

Parameter `patr` geeft het patroon dat we, van links naar rechts, gaan ontrollen “langs” de lijst `ds` (`ds` is één regel van `figuur'`, bestaande uit *d*iepte-getallen). Het ontrolde patroon is `cylcle patr`. We kiezen ervoor om de regel van het plaatje letter-voor-letter op te bouwen, aan de hand van de diepte-getallen in `ds`. Dus in eerste instantie ziet de definitie van `mkregel` er als volgt uit (met nog nader in te vullen extra parameters en argumenten):

```
  mkregel (patr, ds) = mkrgl ... (cycle patr) ds

  mkrgl ... (p:ps) []           =  []
  mkrgl ... (p:ps) (d:[])       =  p: []
  mkrgl ... (p:ps) (d: d':ds)
  =  p: mkrgl ... (cycle patrH) (d':ds),   if d < d'  || omHoog
  =  p: mkrgl ...  ps           (d':ds),   if d = d'  || vlak
  =  p: mkrgl ... (cycle patrL) (d':ds),   if d > d'  || omLaag
     where
     ...
```

Bij een diepte-overgang in `d:d':ds` hebben we extra informatie nodig, te weten: de voorraad letters die weer aan het patroon kan worden toegevoegd (bij een overgang van 1 naar 0), en het huidige patroon (om het nieuwe te maken). De voorrraad letters zullen we als een lijst representeren, die we volgens het LIFO principe zullen bespelen (zodat de letterverdeling in het 3D plaatje zoveel mogelijk gelijk blijft; LIFO = Last In First Out). Voor de initiële voorrraad nemen we een tamelijk grote verzameling: `cycle patr`. Het huidige patroon representeren we door louter zijn lengte, zeg n; we zullen er voor zorgen dat steeds het huidige patroon gelijk is aan `take` n `ps`. Dus bovenstaande definitie wordt nu vervolledigd tot de volgende:

```
> mkregel (patr,ds) = mkrgl (cycle patr) (#patr) (cycle patr) ds
>
> mkrgl voorraad n (p:ps) []          =  []
> mkrgl voorraad n (p:ps) (d:[])      =  p: []
> mkrgl voorraad n (p:ps) (d: d':ds)
> =  p: mkrgl voorraadH nH (cycle patrH) (d':ds),   if d < d'  || omHoog
> =  p: mkrgl voorraad  n  ps            (d':ds),   if d = d'  || vlak
> =  p: mkrgl voorraadL nL (cycle patrL) (d':ds),   if d > d'  || omLaag
>    where
>    patr      = take n ps                                 || huidig patroon
>
>    patrH     = drop (d'-d) patr                          || H: patr slinkt
```

```
>    nH        = n - (d'-d)
>    voorraadH = (reverse . take (d'-d)) patr  ++ voorraad
>
>    patrL     = take (d-d') voorraad ++ patr            || L: patr groeit
>    nL        = n + (d-d')
>    voorraadL = drop (d-d') voorraad
```

Hiermee is het programma klaar. Net zo als `figuur`, is `plaatje` een lijst van strings. We kunnen deze tonen, iedere string op een nieuwe regel en zonder de string quotes en lijsthaken, door hem te onderwerpen aan de standaardfunctie `lay`. (Ter herinnering, `lay ["abc","defg" ...] = "abc\ndefg\n..."`.) De evaluatie van `lay plaatje` geeft:

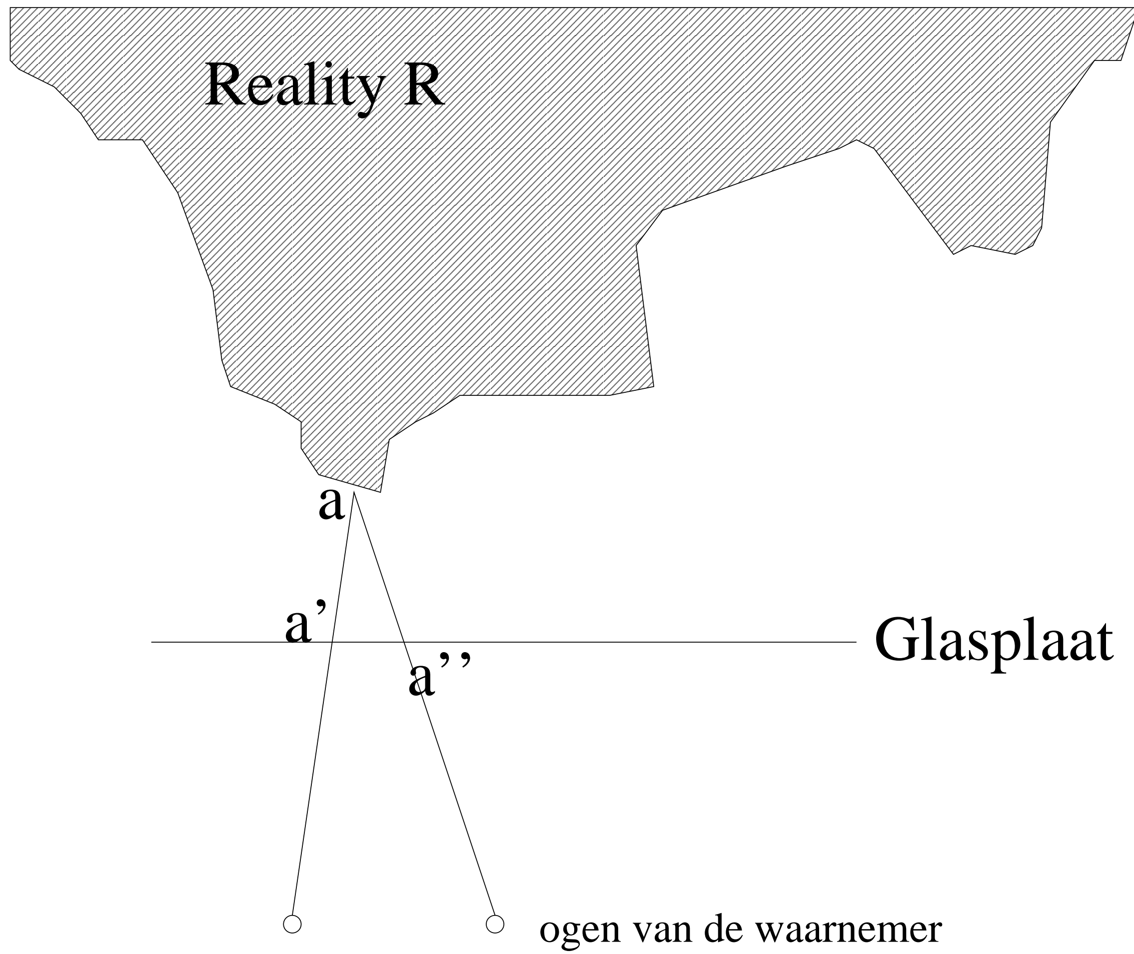
```
              +            +
watzieniktochwatzieniktochwatzieniktochwat
tzieniktochwatzieniktochwatzieniktochwatzi
iktochwatzieniktochwatzieniktochwatzienikt
zieniktochwatzieniktchowatzieniktchowatzie
chwatzieniktochwatzenikitochwatzenikitochw
eniktochwatzieniktchwatzoieniktchwatzoieni
tochwatzieniktochatzienikwtochatzienikwtoc
ieniktochwatzienktochwatziienktochwatziien
ktochwatzieniktochwatzieniktochwatzienikto
atzieniktochwatzieniktochwatzieniktochwatz
niktochwatzieniktochwatzieniktochwatzienik
```

Als hulp voor het bekijken heb ik er twee plustekens boven gezet op een afstand die gelijk is aan de achtergrond-patroonlengte (13 = `#patroon`); je moet de ogen dan zo richten dat je die twee +'en als één duidelijke + ziet met aan weerszijden een vage ernaast. Als je de ogen verkeerd richt, maar toch zo dat er steeds twee patronen over elkaar vallen, krijg je niet helemaal het bedoelde diepte-effect.

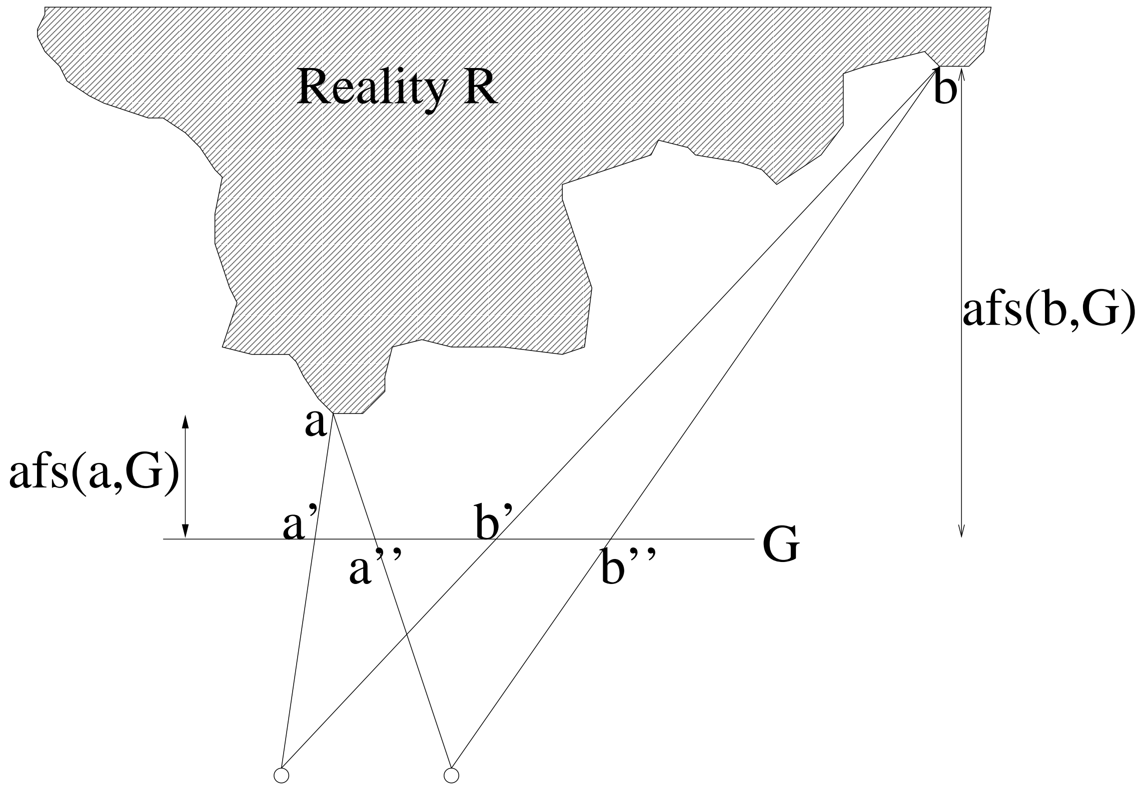
Het programma kan nog iets korter opgeschreven worden door functiecompositie te gebruiken en de tussenresultaten `figuur'` en `plaatje` niet afzonderlijk te benoemen:

```
output = (lay. map mkregel. zip2 patronen. map (map f)) figuur
         where f c = code c - code '0'
```

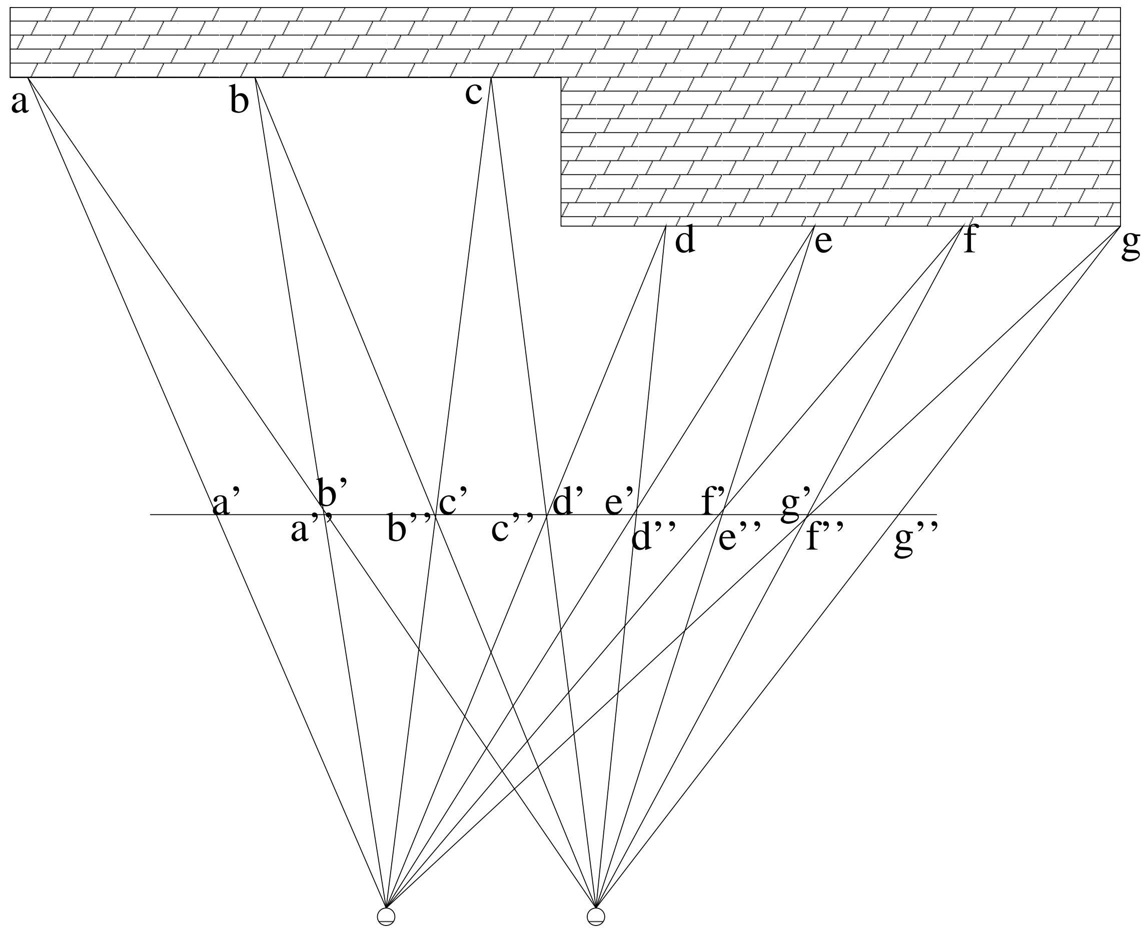
Figuur 1:



Figuur 2:



Figuur 3:



Figuur 4:

V V V V V V V

Figuur 5:

V W V W V W V W V W V W V W

Figuur 6:

```
V W X Y Z V W X Y Z V W X Y Z V W X Y V W X Y V W X Y V W X Y
```

Figuur 7:

```
              +            +
watzieniktochwatzieniktochwatzieniktochwat
tzieniktochwatzieniktochwatzieniktochwatzi
iktochwatzieniktochwatzieniktochwatzienikt
zieniktochwatzieniktchowatzieniktchowatzie
chwatzieniktochwatzenikitochwatzenikitochw
eniktochwatzieniktchwatzoieniktchwatzoieni
tochwatzieniktochatzienikwtochatzienikwtoc
ieniktochwatzienktochwatziienktochwatziien
ktochwatzieniktochwatzieniktochwatzienikto
atzieniktochwatzieniktochwatzieniktochwatz
niktochwatzieniktochwatzieniktochwatzienik
```