1:

3D Graphics

3D-technieken zijn al jaren de norm in de gamewereld maar ook in andere toepassingen winnen ze meer en meer aan belang. Daarbij maakt de steeds vorderende technologie in soft- en hardware het mogelijk om steeds meer realisme te verkrijgen, zelfs in het geval van realtime rendering.

Tijdens deze sessie bekijken we eerst in grote lijnen hoe een 3D-beeld tot stand komt en belichten we een aantal belangrijke begrippen binnen de 3D-wereld. Daarna gaan we even in op de wiskunde achter een aantal eenvoudige 3D manipulaties (o.a. coördinatentransformaties).

Wat volgt is beslist een serieuze boterham, maar hou er rekening mee dat de tekst méér inhoud bevat dan strikt noodzakelijk is om de bijbehorende opdracht te maken.

2:

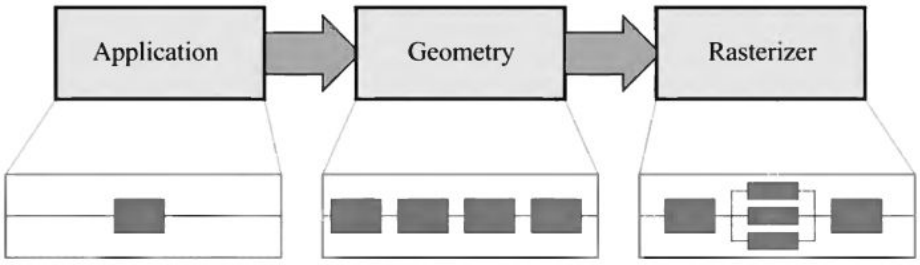
Inhoud theorie

Overzicht van de 3D-Graphics Pipeline

Bij een 3D-toepassing is het de bedoeling dat we de gebruiker op een 2-dimensionaal scherm, gebaseerd op een bitmap bestaande uit individuele pixels, een realistische weergave geven van een (virtuele) wereld in 3 dimensies.

Het proces van de 'berekening' van de virtuele wereld tot en met het bepalen van de uiteindelijke kleur van elk pixel op het scherm wordt opgesplitst in een aantal bewerkingen.

Grofweg ziet die 'rendering pipeline' er als volgt uit:



In de eerste stap zal de **toepassing** (game, simulatie…) de eigenschappen (positie, oriëntatie, kleur, uitzicht…) berekenen van de objecten die samen de virtuele 'wereld' vormen.

De informatie hierover wordt uitgedrukt in coördinaten in het 'World Coördinate System, (WCS).

Vervolgens zullen een aantal **geometrische bewerkingen** moeten gebeuren op de objecten in de virtuele 3D-wereld: coördinatentransformaties, bepalen welke objecten (of delen ervan) zichtbaar zijn, hoe de belichting invloed heeft op de kleuren van de objecten …

Uiteindelijk zal overgegaan worden van de 3D-wereld (waarin de objecten beschreven worden door hun coördinaten) naar de 2D-omgeving van het beeldscherm waarin enkel pixels bestaan: dit is de fase van de **rastering** en raster- of **pixel-bewerkingen**.

3:

Het grootste gedeelte van het werk in deze pipeline gebeurt in het 3D-framework en in de Graphical Processing Unit op je videokaart. Toch moet je een idee hebben van wat er precies achter de schermen gebeurt om zelf een 3D-toepassing te schrijven.

In de volgende paragrafen zullen we dus wat dieper ingaan op de geometrische en pixel-bewerkingen.

De geometrische bewerkingen (Vertex-bewerkingen)

Vertices – polygonen

Om de complexiteit te beperken en efficiënt te kunnen werken, zullen in de render-pipeline geen complexe objecten gemanipuleerd worden: die worden opgesplitst in kleine veelhoeken (polygonen), in praktijk bijna altijd driehoeken (triangles).

De positie van zo een driehoek wordt opgeslagen onder de vorm van de coördinaten van de drie hoekpunten (vertex – vertices).

Het complete model wordt dan opgeslagen als een lijst van driehoeken in een zogenaamde ‘mesh’. Voor de vertices van die driehoeken worden coördinaten in een ‘Local Coördinate System’ (LCS) gebruikt. Elk model heeft zo zijn eigen LCS.

Wanneer zo een model in de ‘wereld’ geplaatst wordt, zullen alle coördinaten via transformaties herrekend moeten worden naar het WCS (zie verder).

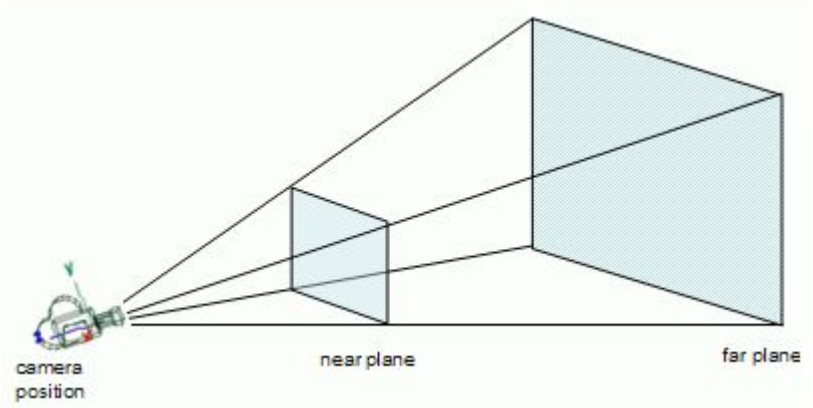
4:

Uiteindelijk krijgen we een scene bestaande uit alle nodige objecten, elk op zijn eigen positie met alle coördinaten uitgedrukt in het WCS.

De camera – Frustrum

Om dat beeld uiteindelijk op het scherm te tonen wordt gebruik gemaakt van een zogenaamd ‘cameramodel’: er wordt verondersteld dat een camera op de scene gericht is en dat wat die camera ‘ziet’ op het scherm getoond zal worden.

In praktijk zal slechts een beperkt deel van de ‘wereld’ zichtbaar zijn door de camera (en dus op het scherm). Dit deel wordt wordt bepaald door de positie, richting, maar ook bv. openingshoek van de camera. Om praktische redenen (vervorming en rekenwerk) wordt het beeld ook nog beperkt tot alles wat zich tussen een ‘near plane’ en een ‘far plane’ bevindt.



Het gebied dat effectief zichtbaar gemaakt zal worden op het scherm wordt het ‘ (viewing) frustrum’ genoemd. Dit frustrum is altijd een afgeknotte pyramide zoals in de tekening hierboven.

Binnen een toepassing kan je zowel de wereld zelf als de camera manipuleren om het beeld op het scherm te wijzigen.

Rendermethodes

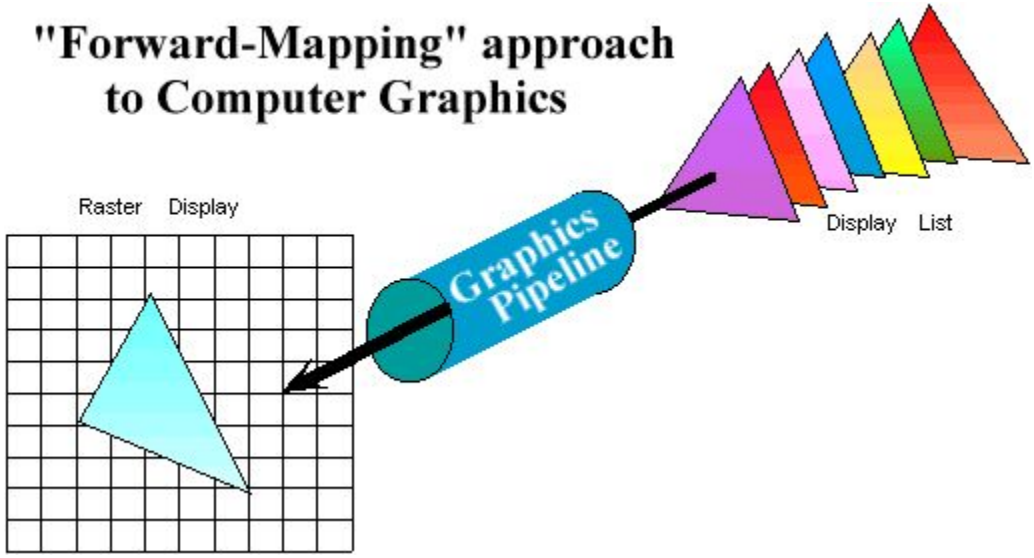
De omzetting van een 3D scène met al haar objecten, lichten, effecten en allerlei andere parameters naar een 2D beeld kan op verschillende manieren aangepakt worden. Afhankelijk

van de manier van werken zal het renderen sneller of trager, meer of minder nauwkeurig gebeuren.

Forward Mapping

Bij forward mapping worden alle berekeningen vanuit de objecten naar de kijker toe gedaan. Dit betekent dat het analytische rekenwerk eerst gebeurt. Alle elementen in de scène worden apart uitgerekend. Pas daarna kan het sampling proces (verwerken naar pixels) gebeuren.

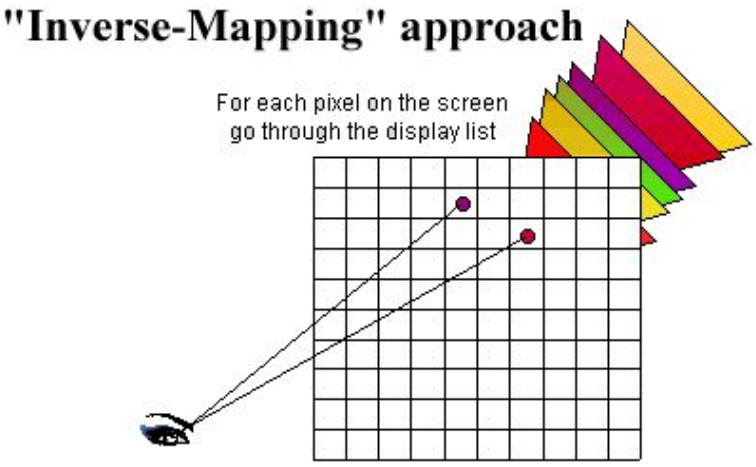
Dit is traditioneel de (meest) gebruikte methode voor realtime 3D-rendering omdat ze met het minste rekenwerk kan gebeuren. Nadeel is wel dat sommige effecten (spiegeling, reflectie, transparantie, breking, schaduw) moeilijker weer te geven zijn.



Inverse Mapping

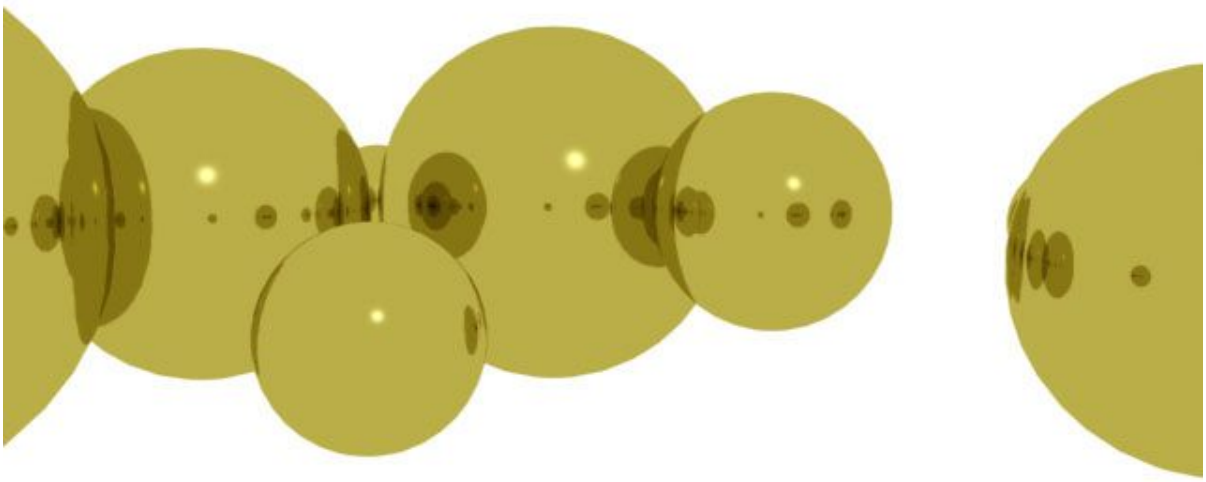
Het omgekeerde kan ook: voor elke pixel worden de primitieven doorrekend. Het samplen gebeurt hier dus eerst. Het analytisch rekenwerk op het einde. Inverse mapping wordt ook wel ray-casting genoemd.

6:

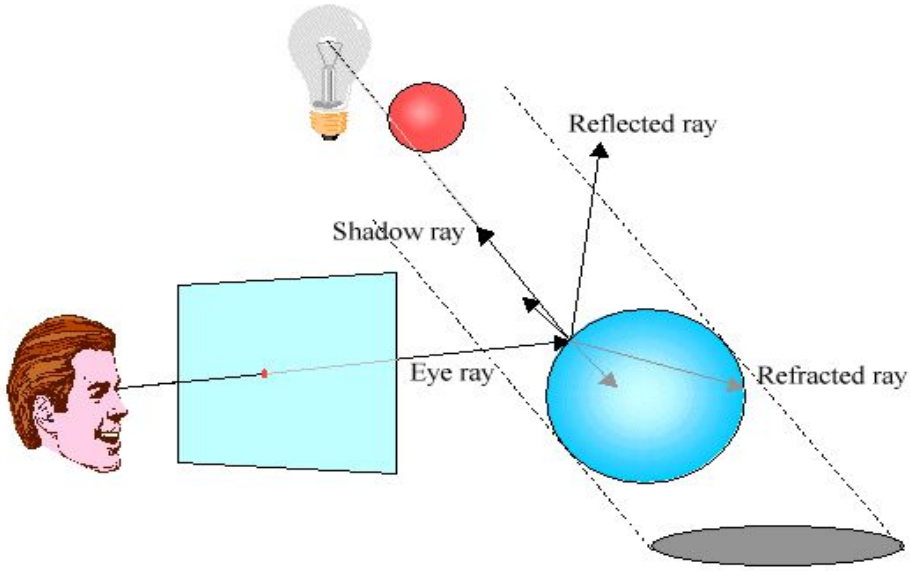


Ray Tracing

Ray casting is erg rekenintensief en daardoor niet geschikt voor previews of realtime toepassingen. De interesse in deze rendermethode was daarom aanvankelijk eerder beperkt. Door recursieve toepassing van ray casting kunnen echter wel zeer nauwkeurig schaduw, reflecties en breking berekend worden! Dit heet ray tracing.



Bij ray tracing wordt (zoals bij ray casting) vertrokken van uit het oog waarna bepaald wordt welk object zichtbaar is voor een bepaald pixel. De lichtbanen van elke lichtbron naar de plaats in kwestie op het object worden nagekeken. Als bijvoorbeeld een ander object hier tussenin ligt, kan deze lichtbron genegeerd worden. Breking en reflectie worden op een gelijkaardige manier berekend.



Het enige nadeel bij ray tracing is dat de gerenderde beelden er vrij steriel - eigenlijk een beetje té perfect - uitzien. Door andere technieken (zoals bijvoorbeeld samengestelde materialen, radiosity, …) met ray tracing te combineren kunnen de beelden beter tot leven gewekt worden.

In wat volgt bespreken we de 'Forward mapping' techniek zoals die in alle klassieke grafische kaarten toegepast wordt.

Culling – Clipping

Bij de originele berekening door je toepassing worden een aantal objecten gecreëerd en gemanipuleerd. Omdat het renderen van het beeld zeer rekenintensief is, willen we daarbij enkel rekening houden met objecten die effectief zichtbaar zullen zijn op het scherm.

Een eerste bewerking is dus het wegfilteren van alle polygonen die buiten het frustrum liggen ('**Culling**'). Conventioneel wordt ook van elke polygoon slechts één zijde gerenderd (het renderen van bv. de binnenkant van een bol zou weinig zin hebben). Alle polygonen die met hun 'achterkant naar de camera gericht zijn, worden dus ook weggefilterd ('**backface culling**').

Tenslotte kunnen er ook polygonen zijn die gedeeltelijk in het frustrum liggen: die snijden dus één of meerdere zijvlakken ervan. Deze polygonen worden opgesplitst in een aantal deelpolygonen die samen het deel binnen het frustrum vormen ('**clipping**'). De deelpolygonen die buiten het frustrum liggen worden ook weggelaten in de volgende berekeningen.

8:

Shading

Om een realistische scene weer te geven is het niet voldoende om de vorm en de positie van de verschillende objecten juist weer te geven, ook het uitzicht ervan moet juist zijn. Dit uitzicht wordt bepaald door de materiaaleigenschappen (kleur, oppervlakte) en de invloed van de verschillende lichtbronnen daarop. Voor elke vertex van elke polygoon zal op basis van de materiaaleigenschappen en de verschillende lichtbronnen berekend worden welke kleur dat hoekpunt moet krijgen. Deze berekening wordt '**shading**' genoemd.

Er bestaan verschillende shading-technieken. Twee ervan kunnen alle 3D chips hardwarematig uitvoeren: '**flat shading**' en '**Gouraud shading**'. Daarnaast zijn er ook nauwkeuriger technieken en technieken die een speciaal effect creëren, die over het algemeen niet in real-time toepassingen gebruikt worden maar bijvoorbeeld wel bij het renderen van animatiefilms.

Flat shading

Flat shading is de simpelste manier van inkleuren: bij dit systeem krijgt ieder polygoon uit een 3D object één egale kleur. Een voorbeeld van flat shading zien we bij de torusknoop hieronder. Welke kleur er wordt gebruikt volgt uit de kleur en oriëntatie van de polygoon ten opzichte van de lichtbron(nen).

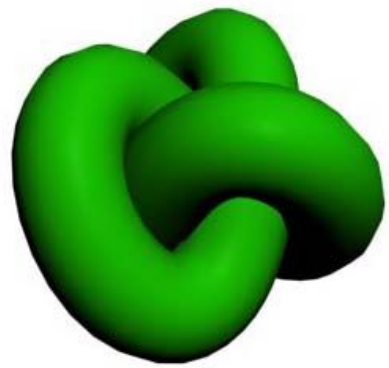
Omdat de oriëntatie van twee aanliggende polygonen verschillend kan zijn, kan ook de kleur van die polygonen verschillend zijn. Dit is in het beeld heel duidelijk zichtbaar aan de 'randen' van de polygonen.

Gourad shading

Een mooiere manier van shaden is **Gouraud shaden**. Deze methode vraagt wel heel wat meer rekenkracht van de 3D chip. Bij dit systeem wordt voor ieder hoekpunt (vertex) van elke polygoon de gewenste kleur berekend. Voor gemeenschappelijke vertices tussen verschillende

polygonen wordt als eindkleur het gemiddelde van de berekening van de kleur voor de verschillende polygonen gebruikt.

De kleuren van de pixels binnen de polygonen worden berekend via interpolatie, zodat er een mooie kleurovergang binnen alle driehoeken ontstaat. Onderstaande figuur toont dezelfde torusknoop als hierboven, maar dan ingekleurd met de Gouraud shading techniek. Het verschil is duidelijk. Aangezien Gouraud shading heel wat meer rekenkracht vraagt, is het ook wel te verklaren dat dit systeem vroeger, toen er nog geen 3D hardware was, nooit werd toegepast bij real-time 3D programmatuur. Sinds de komst van de 3D versnellers wordt juist flat shading bijna niet meer toegepast.



Naast flat shading en Gouraud shading zijn er ook nog andere gebruikte shading technieken, die echter meer rekenkracht vergen en (nog) niet bruikbaar zijn voor real-time toepassingen.

Phong shading

Bij **Phong shading** worden de normalen in de hoekpunten over de lijnen van elk polygoon geïnterpoleerd. Daarna wordt op elk van deze normalen het belichtingsmodel toegepast om de kleur van de betreffende pixel te berekenen.

Andere…

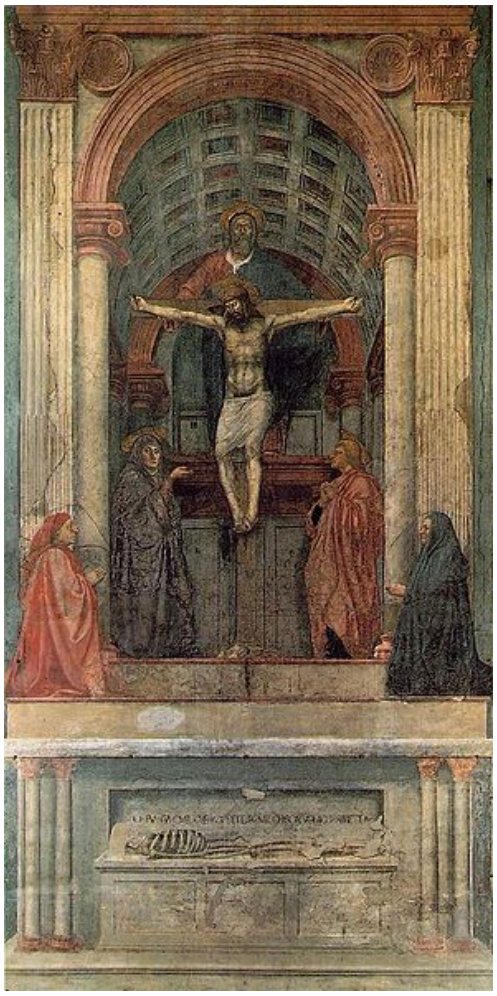
Andere voorbeelden zijn Metal shading voor het creëren van metaalachtige reflecties of Ink shading voor een tekenfilm effect. Merk op dat Ink shading op zich niet echt een shading-techniek is, maar eerder een bewerking die achteraf gebeurt op de berekende kleurwaarden om dit bepaalde effect te bekomen.

Deze laatste technieken worden nog niet real-time toegepast omdat ze te veel rekenkracht vergen. We vinden ze dan ook alleen maar terug bij 3D rendering pakketten zoals 3D Studio MAX. Wellicht zullen in een nabije toekomst ook Phong en Metal shading in 3D acceleratoren worden opgenomen.

Projectiemethodes

In alle stappen tot nu toe gebeurden alle bewerkingen op de polygonen en hun hoekpunten op basis van 3-dimensionale ruimtemeetkunde (geometrie). Om te beslissen waar elk van de objecten op het scherm getoond zal worden, moet overgegaan worden naar een 2-dimensionale projectie van die objecten op een vlak dat we uiteindelijk op het scherm zullen weergeven.

De basistechnieken hiervoor werden uitgewerkt in de renaissance door kunstenaars als Brunneleschi, Masaccio, Van Eyck, Vermeer… bij de ontwikkeling van technieken om op een correcte manier met perspectief te werken.

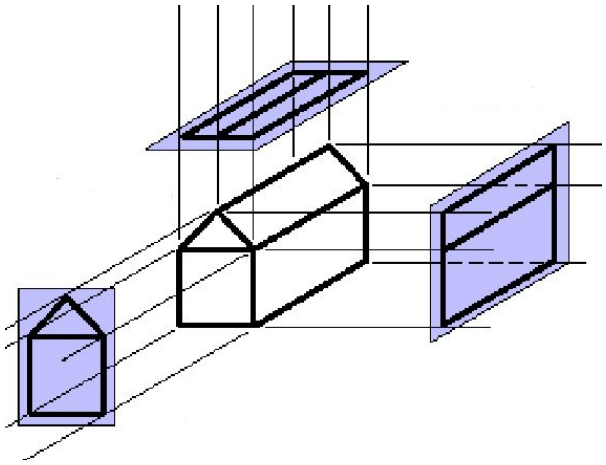


Die projectie kan op verschillende manieren gebeuren. De meest gebruikte projectiemethoden in computertoepassingen:

12:

Orthografische projectie

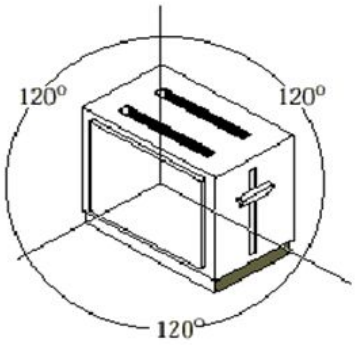
Dit type projectie wordt gebruikt door architecten om het grondplan en de aanzichten van een huis uit te tekenen. De projectoren staan hierbij loodrecht op het projectievlak (vandaar de naam “orthografisch”).

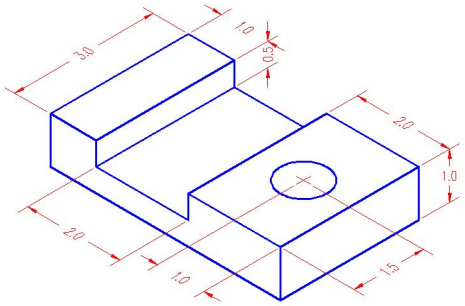


De projectievlakken worden meestal evenwijdig gekozen met het coördinatenstelsel van het uit te tekenen object.

Isometrische Projectie

Er wordt gewerkt met drie assen die onderling een hoek vormen van 120°. Hierbij kan de figuur een ruimtelijk gevoel geven. Isometrische projectie wordt vooral gebruikt bij het tekenen van schema’s.



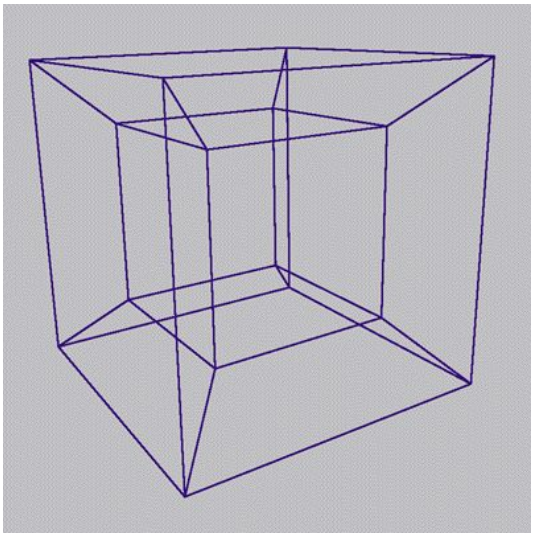


Soms wordt isometrische projectie gebruikt in computerspelletjes (Sims, sommige platform-games…).

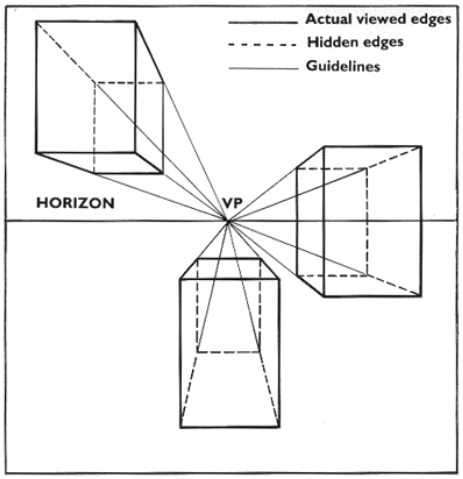
Perspectivistische projectie

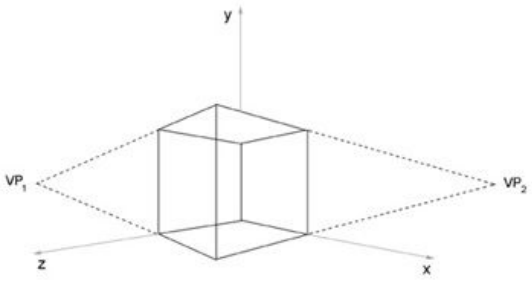
Hierbij wordt rekening gehouden met het verschijnsel dat objecten van gelijke grote kleiner lijken te worden naarmate ze verder verwijderd zijn van de kijker. Als we in een computertoepassing een realistisch beeld willen krijg, werkt men bijna altijd met perspectivistische projectie.

13:



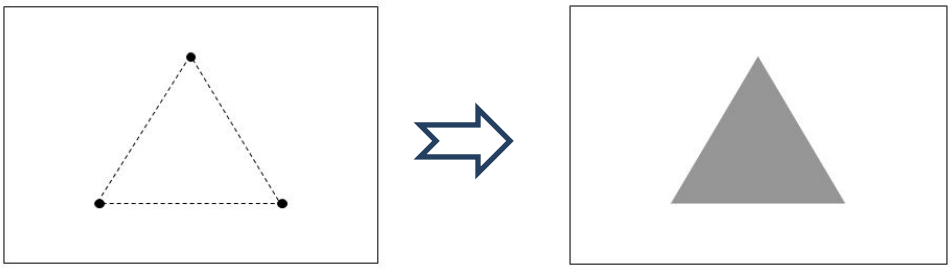
Hierbij kan met één of meerdere vluchtpunten gewerkt worden:





Rastering

Op basis van de gebruikte projectie, zal nu voor elke zichtbare polygoon (tot nu toe voorgesteld door zijn vertices) een aantal pixelfragmenten berekend worden die samen op het scherm de polygoon zullen voorstellen.



Vanaf nu zullen er geen bewerkingen meer gebeuren op de coördinaten van de polygonen, maar wel op de pixels in de individuele fragmenten.

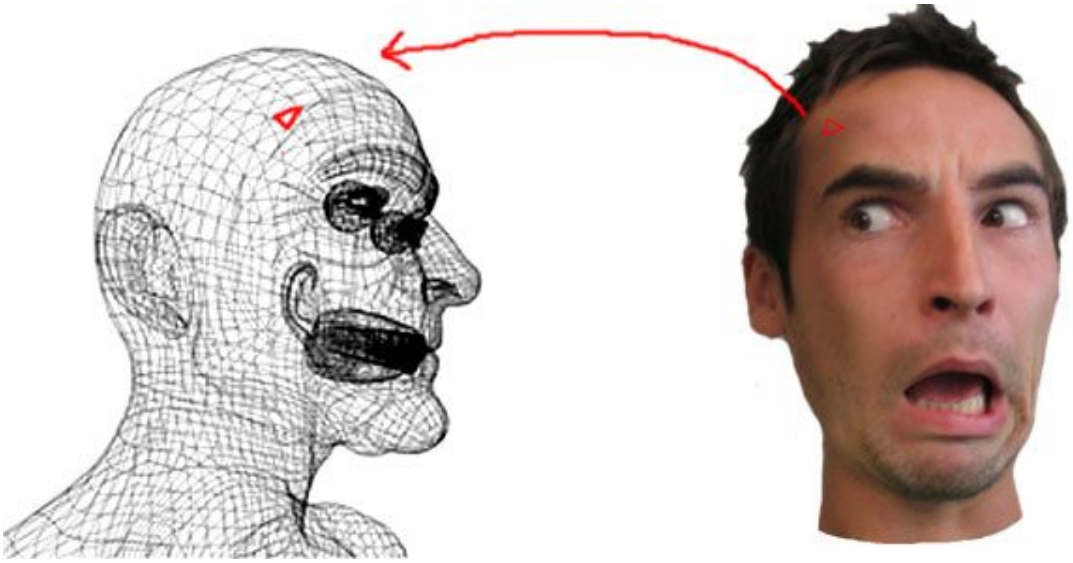
Raster-bewerkingen

Werken met texturen

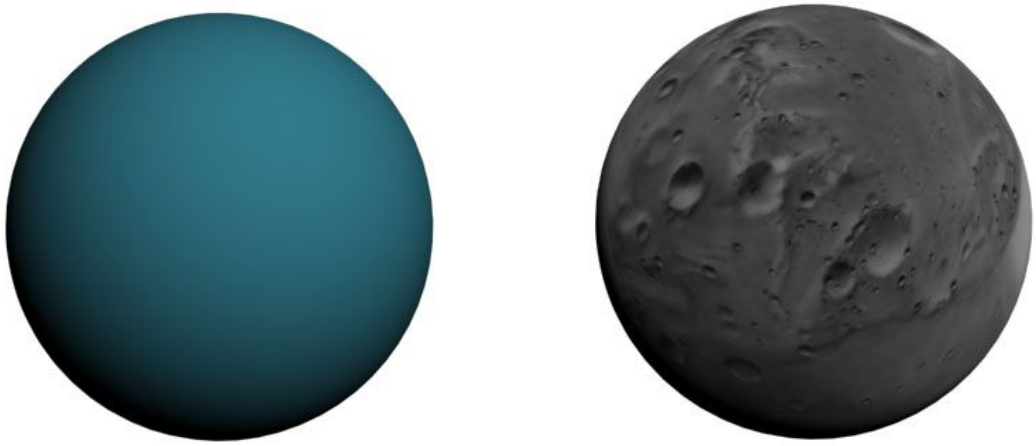
Textures

Een extra manier om 3D objecten in te kleuren en tegelijk het realisme van de 3D afbeelding te vergroten, is het toepassen van texture mapping. Hierbij wordt een bitmap geplaatst op de gewenste 3D polygonen. Zo ontstaan zeer realistische afbeeldingen zonder dat daar erg veel polygonen voor nodig zijn. Elke driehoek komt overeen met een welbepaald gebied van de texture map:

15:



De maan zouden we bijvoorbeeld kunnen modelleren door miljarden kleine polygonen met elk de juiste kleur van het plaatselijke terrein type. Zelfs de snelste computers zouden dit 3D object dan nog niet real-time kunnen manipuleren. Simpeler is het om gewoon een standaard bol te nemen en daar een afbeelding van een maanlandschap overheen te plakken. We hebben dan maar erg weinig polygonen nodig, terwijl het resultaat er vrijwel even realistisch uitziet.



Bij huidige 3D spellen wordt vrijwel alleen nog maar gebruik gemaakt van texture mapping. Aangezien het toepassen van texture mapping veel minder processorkracht vereist dan het opdelen van 3D objecten in veel meer polygonen, is het mogelijk om zeer realistische 3D beelden toch nog met meer dan 30 frames per seconde (of meer) over het scherm te laten rollen. In spellen als Quake zijn de muren, vloeren en plafonds niet meer dan platte vierhoeken met daarop een texture van bijvoorbeeld bakstenen of tegels.

16:

De toepassing zal op voorhand definiëren hoe een bepaalde bitmap geplaatst moet worden op de polygonen die een object vormen. Alle geometrische bewerkingen die gebeuren op een polygoon , gebeuren ook op de textuur die aan die polygoon gekoppeld is, en bij de rastering wordt voor de kleur van elk pixel ook rekening gehouden met die textuur.

Texture Filtering

De oorspronkelijke texture zal nooit precies hetzelfde formaat hebben als het 3D object waar de afbeelding op geplaatst moet worden. Uit de pixels van de texture (die in het 3D jargon 'texels' worden genoemd) moeten dus op één of andere manier de kleuren van alle pixels van het 3D object worden afgeleid.

Point Sampling

De makkelijkste en snelste manier om dit te doen heet **point sampling** of **point filtering**. Hierbij wordt voor iedere pixel de kleur van de dichtstbijzijnde texel gekopieerd. Dit systeem wordt daarom ook wel de **nearest** methode genoemd.

Hoewel point sampling erg snel gaat, heeft het ook een aantal nadelen. Indien het 3D object veel kleiner is dan de oorspronkelijk texture kan er veel detail verloren gaan, omdat veel texels gewoonweg verwaarloosd worden. Als een dergelijk object beweegt kan het daarneboven zijn dat het een beetje knipperend overkomt, aangezien kleine details in de texture soms wel en soms weer niet zichtbaar zijn. Dit wordt **undersampling** genoemd.

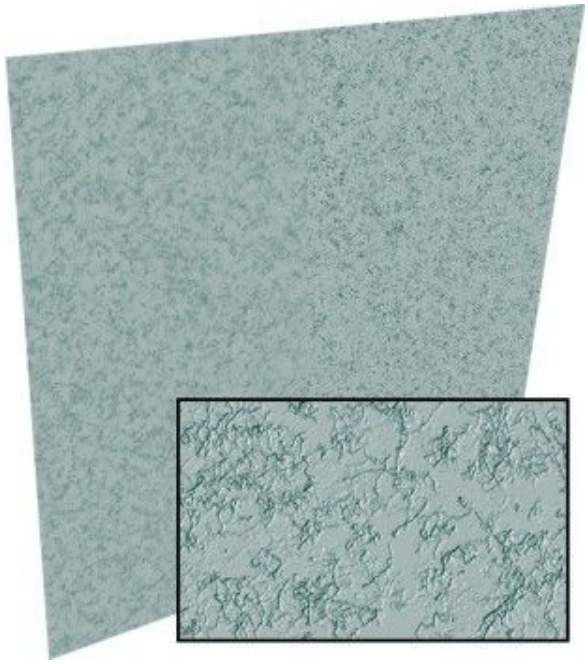
Als het 3D object echter groter is dan de oorspronkelijk texture ontstaat er weer een ander probleem: **oversampling**. Het resultaat wordt dan erg blokkerig, aangezien een enkele texel dan identiek op meerdere pixels wordt geprojecteerd. Iedereen herinnert zich nog wel hoe blokkerig de graphics van spellen als Doom 1 en Doom 2 werden als je dicht tegen een muur kwam te staan. Dit komt doordat in deze games alleen maar van point sampling gebruik werd gemaakt.

Bilineair filteren

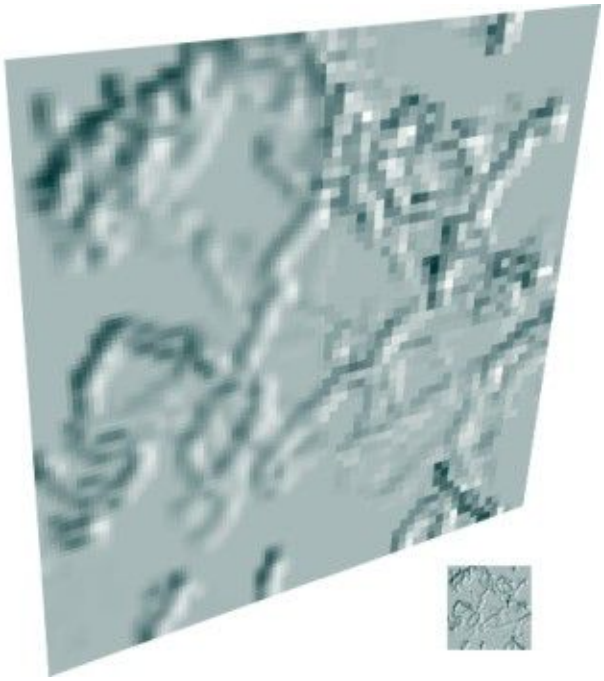
Een systeem om deze twee problemen al een beetje tegen te gaan is bilinear filtering. Bij bilinear filtering wordt niet alleen naar de dichtstbijzijnde texel gekeken, maar wordt de uiteindelijke kleur berekend door interpolatie tussen de kleuren van de vier dichtstbijzijnde. Dit systeem is dus in principe te vergelijken met het anti-aliasing systeem, waarbij met maximaal vier texels rekening wordt gehouden. De resultaten zijn in ieder geval stukken beter dan met point sampling.

17:

In de volgende figuren zien we duidelijk de voordelen van bilinear filtering ten opzichte van point sampling. De eerste figuur toont een 3D object waarbij de texture flink verkleind is. Rechts zien we het resultaat met point sampling (waar veel detail verloren gaat) en links het resultaat met bilinear filtering. Een gedeelte van de oorspronkelijke texture is ook in de afbeelding te vinden. De tweede figuur toont het oversampling probleem, waarbij een texture flink vergroot moet worden (zoals bijvoorbeeld nodig is bij het "tegen een muur aanlopen" in een 3D game). Rechts zie je het erg blokkerige resultaat van point sampling (zoals in Doom 1 en 2) en links het al betere resultaat van bilinear filtering (zoals bijvoorbeeld in Quake). In de afbeelding is ook de oorspronkelijk texture te zien.



De voordelen van bilinear filtering bij sterk verkleinde textures
(links: bilinear filtering / rechts: point filtering)



De voordelen van bilinear filtering bij sterk vergrote textures
(links: bilinear filtering / rechts: point filtering)

MIP-maps

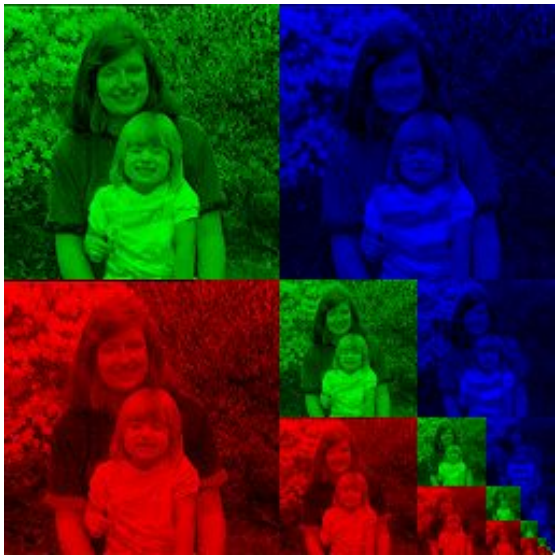
Iets wat in 3D games steeds meer gebruikt wordt zijn de zogenaamde MIP-maps. MIP is een afkorting voor het Latijnse "Multum in Parvo" wat "Meerdere in Één" betekent.

In een MIP-map worden dan ook meerdere versies van een texture opgeslagen. Elke nieuwe variant is de helft kleiner als de vorige en door de auteur zorgvuldig met behulp van anti-aliasing verkleind.

19:

Bij het toepassen van texture mapping zoekt de 3D videokaart het meest passende textureformaat uit de MIP-map. Op de gekozen texture wordt dan (meestal) bilinear filtering toegepast. Om de kwaliteit van MIP-mapping te behalen met een filtering methode zouden veel grote tweedimensionale filters noodzakelijk zijn.

MIP-maps kunnen overigens heel efficiënt worden opgeslagen. Onderstaande figuur illustreert dit:



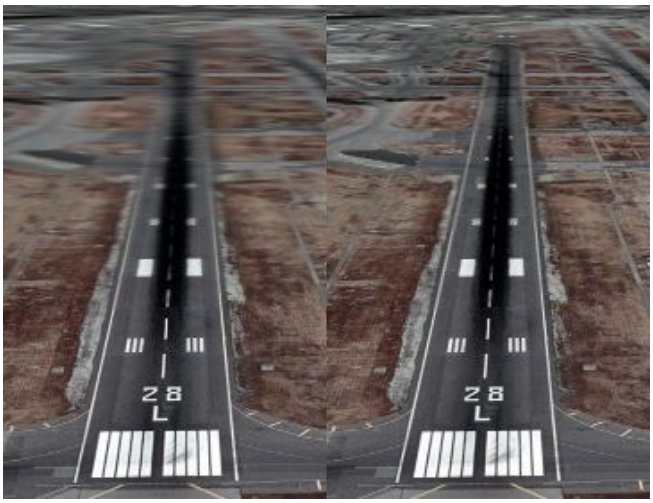
Toch heeft ook MIP-mapping een nadeel. Als een 3D object zich naar voren of naar achteren verplaatst, wordt er op een gegeven moment van MIP level veranderd. Deze resolutiesprong is soms goed zichtbaar. Om dit tegen te gaan is er nog een trilinear filtering systeem.

Trilineair filteren

Bij trilinear filtering wordt eerst op de twee meest ideale MIP levels bilinear filtering toegepast. Uit de twee resultaten wordt door middel van interpolatie de uiteindelijke kleur gefilterd. Trilinear filtering is de methode waarbij verreweg de mooiste resultaten worden behaald, maar deze techniek vraagt zeer veel van de 3D chip. Ook is er bij trilinear filtering veel meer geheugen-bandbreedte nodig, aangezien er steeds twee MIP levels gebruikt worden.

Bij games als Quake 3 werd trilinear filtering al zonder problemen toegepast. De resultaten zijn absoluut verbluffend in vergelijking met wat daarvoor kon bereikt worden, maar de snelheid kan flink terugvallen ten opzichte van standaard bilinear filtering.

Voor situaties waar ook trilineaire filtering problemen geeft (bv. texturen die onder een extreme hoek bekeken worden) kan men '**anisotropic filtering'** gebruiken. Dit is een variant die nog wat meer rekenwerk en geheugenbandbreedte vraagt, maar ondertussen in hardware ondersteund wordt op de meeste videokaarten:



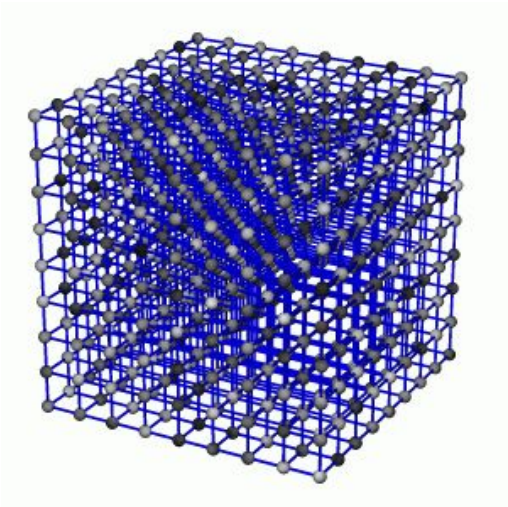
Links zie je een beeld gerenderd met trilineaire filtering, rechts met anisotrope filtering.

Solid Textures

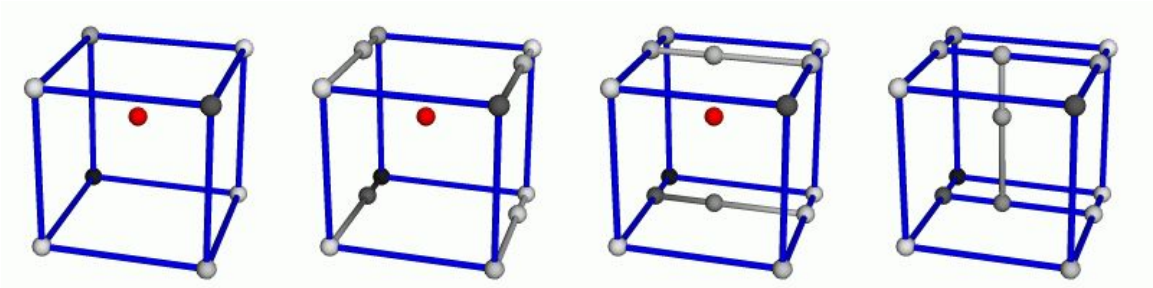
Een speciaal soort textures zijn de zogenaamde solid textures. Dit zijn 3D-texturen (geen 2D zoals bij gewone texture maps) die door de applicatie zelf gegenereerd worden. Ze zijn ideaal voor het creëren van organische materialen zoals hout, water, marmer, enz… Deze materialen worden gegenereerd door een bepaalde 3D-functie over een 3D-volume te laten uitvoeren. Over het algemeen wordt bovenop deze basisfunctie vaak ook een noise- of turbulence-functie geplaatst (op zich ook een 3D-texture) om in de textuur ook wat random variaties te verkrijgen. Dit geeft het materiaal een nog natuurlijker, organischer uitzicht.

Noise

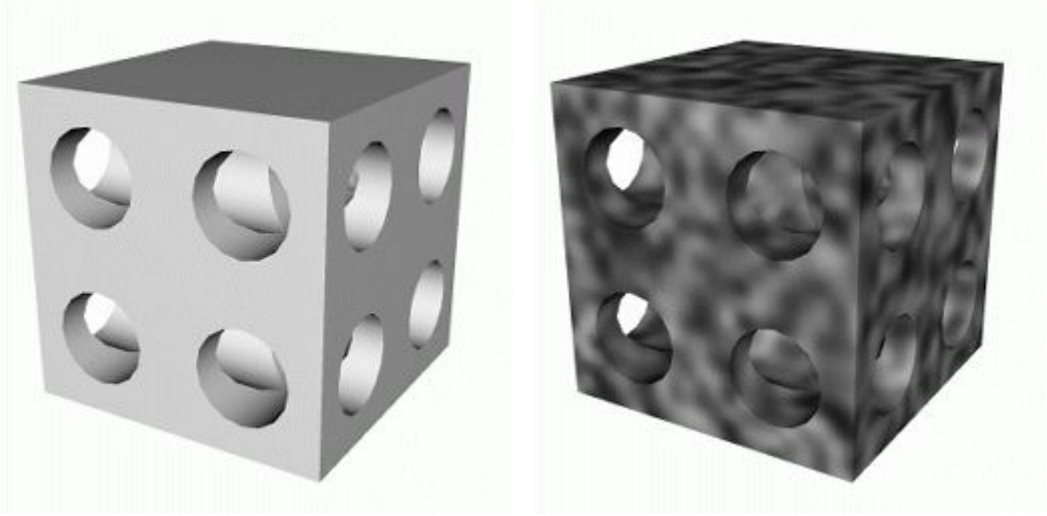
De creatie van 3D-noise komt in essentie neer op het toekennen van random helderheidswaardes aan de punten van een driedimensionaal rooster, zoals hieronder wordt afgebeeld:



Door middel van interpolatie kan dan de helderheidswaarde van elk punt in het 3D-object bepaald worden, nadat eerst gecheckt wordt in welk deel van het rooster dit punt zich bevindt. Dit wordt hieronder geïllustreerd.



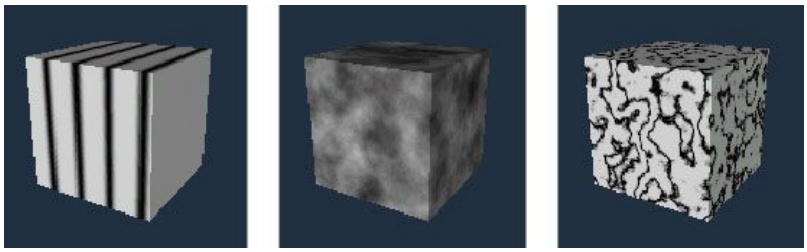
Op die manier krijgt elke punt in het volume een bepaalde noisewaarde:



Noise op zich is niet erg mooi. Een veel bruikbaarder resultaat kan verkregen worden door meerdere noise-functies met verschillende frequenties te combineren. Het resultaat noemt men…

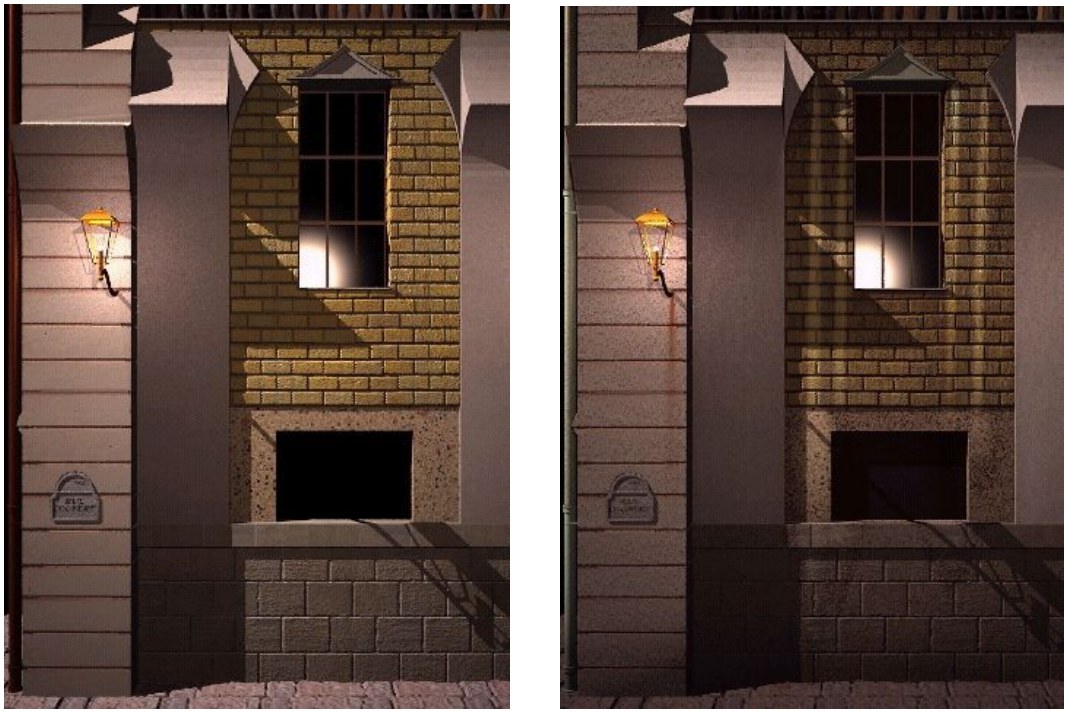
Turbulence

De frequentie van een noise-functie slaat op de dichtheid van het rooster waarop de random-functie wordt toegepast. Door noise met verschillende frequenties te combineren, wordt een mooier effect verkregen. Turbulence wordt vaak in combinatie met andere texturen gebruikt zoals hieronder te zien is:



Samengestelde materialen

Met de materialen die met behulp van bovenvermelde shading- en texturing technieken vervaardigd worden, krijgen we al heel mooie resultaten. Toch zien dat soort gerenderde beelden er vaak nogal steriel uit. Om ze wat meer leven in te blazen, werkt men vaak met samengestelde materialen. Die worden samengesteld uit verschillende van de hierboven vermelde textures. Het marmer uit de vorige paragraaf was hier al een voorbeeld van. De afbeelding hieronder is een ander voorbeeld. De rechter figuur bevat samengestelde materialen.



Belichting

Het berekenen van de belichting van een scene is een erg rekenintensieve taak. Maar ook het instellen op zich van de belichting is niet gemakkelijk. Het belang van dit aspect van het modelleren van een 3D scène wordt maar al te vaak onderschat. Zonder een goede belichting kan een prachtig opgebouwd 3D-model er erg triestig uitzien. Het is in de eerste plaats de belichting die alles tot leven wekt.

Voor elke lichtbron kunnen een aantal parameters ingesteld worden zoals lichtkleur, helderheid, schaduwmethode, verzwakking, speciale effecten…

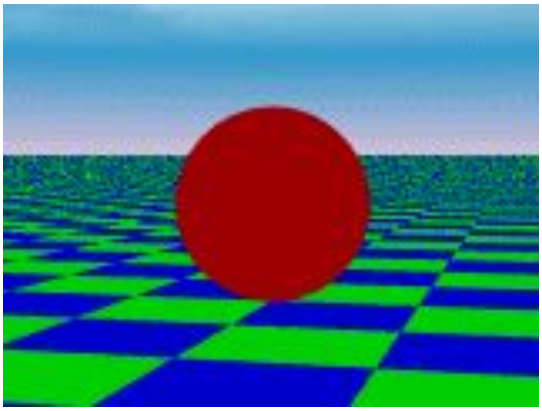
Een goede belichting instellen is erg moeilijk en er zijn geen regels waaraan je je kan vastklampen om het zeker goed te doen. Ervaring opdoen door veel te experimenteren is hier de boodschap.

Er zijn verschillende soorten lichtbronnen. Hieronder worden de belangrijkste vermeld.

25:

Ambient Light

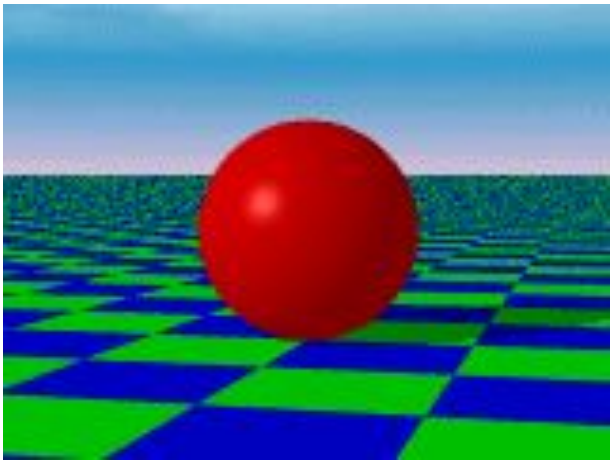
Ambient Light is een lichtbron zonder oorsprong of richting. Het licht is als het ware alom tegenwoordig. De oppervlakte-eigenschappen van de voorwerpen bepalen hoeveel van dit licht weerkaatst wordt. Het voordeel van een ambient light is dat je hele scène meteen verlicht is, het nadeel is echter dat er geen schaduwwerking optreedt waardoor alles een vrij steriele en “platte” indruk geeft.



Directional Light

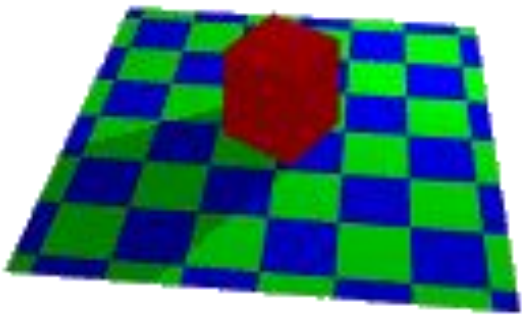
Dit is een lichtbron die zich zogezegd “op oneindig” bevindt. Het gevolg is dat alle lichtstralen die ze uitzendt evenwijdig zijn. Dit soort licht gebruikt men bijvoorbeeld om in een scene een belichting te voorzien zoals die van de zon afkomstig is.

Aangezien de zon vrij ver van ons verwijderd is, lijkt het voor ons alsof de stralen ervan evenwijdig lopen. Dat kun je gemakkelijk zien aan de schaduw van een object met evenwijdige zijdes. De overeenkomstige schaduwlijnen zullen ook evenwijdig lopen.



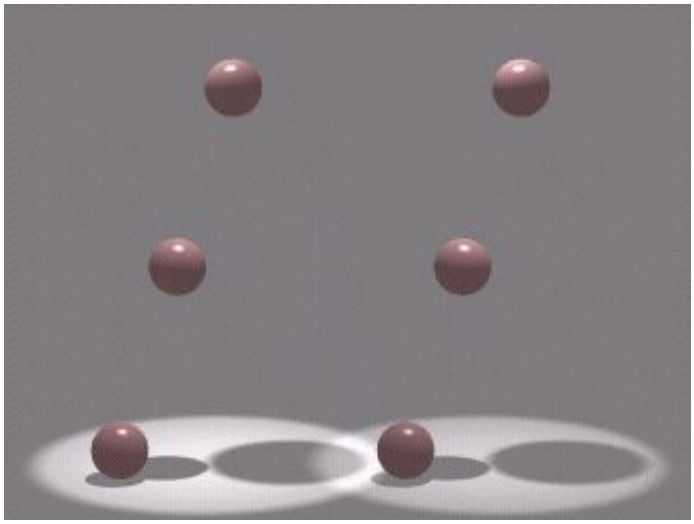
Point Light

De naam zegt het zelf: een point light is een puntlichtbron. Deze stuurt het licht in alle kanten uit. Een gloeilamp die in een ruimte ophangt is bijvoorbeeld een puntlichtbron.



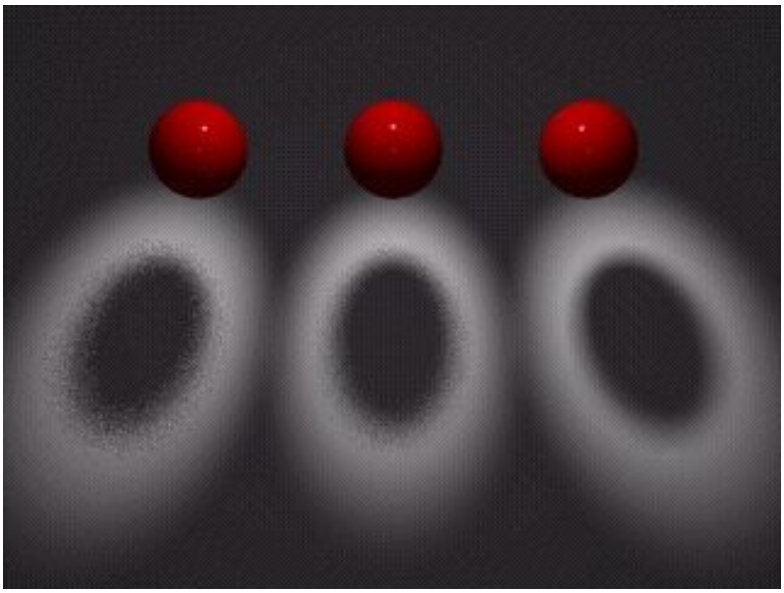
Spot Light

Een spot light kun je je voorstellen als een puntlichtbron onder een conische kap. Een belangrijk verschil met een point light is wel dat de lichtintensiteit afneemt naar buiten toe (van de centrale kegelas weg). De lichtkern met 100% helderheid wordt de hotspot genoemd, het deel erbuiten – waarin de lichtintensiteit afneemt van 100% tot 0% - de falloff.



Area Light Source

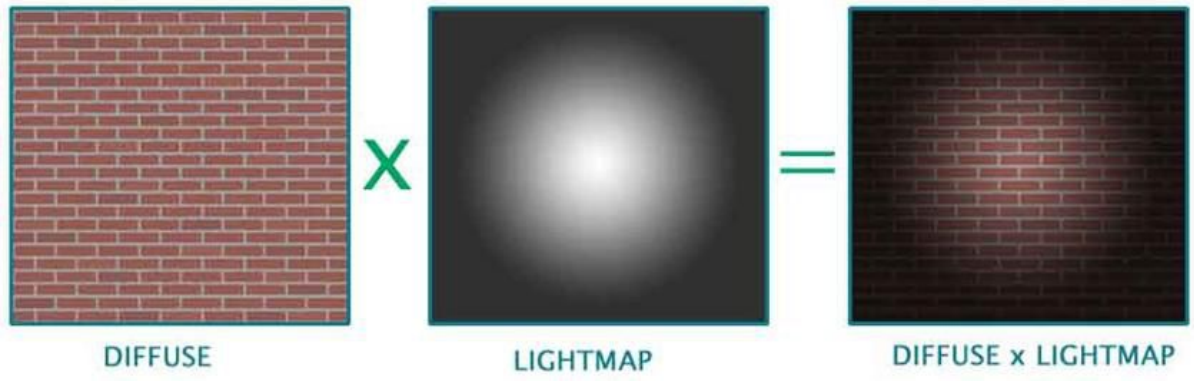
Bij een area light source doet een oppervlak dienst als lichtbron. Het licht wordt op die manier “diffuser” verspreid, en de schaduwen die door dergelijke bron “geworpen” worden, zijn veel zachter dan deze van de vorige besproken bronnen.



Light Maps

Voor real-time applicaties (vooral games) is het bijna onmogelijk om de berekening van de belichting snel genoeg uit te voeren. In dergelijke toepassingen wordt dan ook meestal gebruik gemaakt van een techniek die heel wat minder rekenkracht vergt, namelijk light mapping.

Een light map is niet veel meer dan een grijswaarden-afbeelding die de lichtintensiteitsverdeling voorstelt over een bepaald oppervlak. Dergelijke maps kunnen eenvoudig real-time vermenigvuldigd worden met de basistextures uit de 3D-scène.



Op deze manier kan zonder al te veel belasting van de videochip snel een sfeervolle 3D-ruimte gecreëerd worden. Het onderstaande voorbeeld uit Quake illustreert dit:

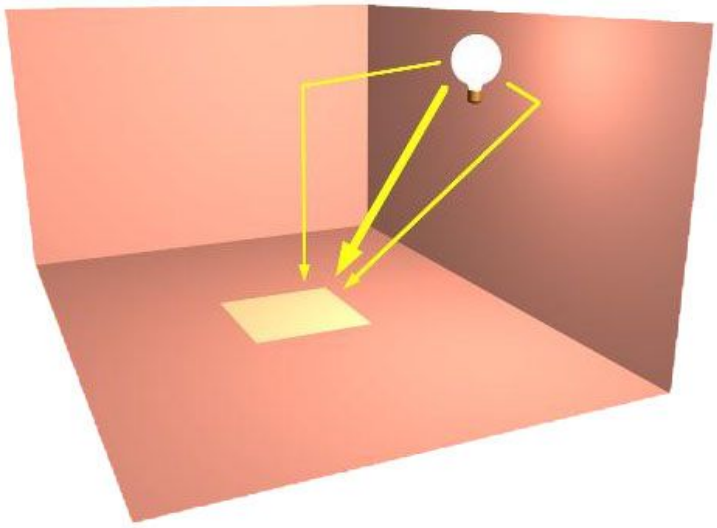
enkel texture maps

texture maps en light maps

29:

Radiosity

In werkelijkheid gebeurt lichtverdeling een stuk complexer dan tot hier toe besproken werd. Een belicht oppervlak zal zich op zijn beurt ook weer als lichtbron gedragen, gezien dit oppervlak het ontvangen licht (deels) weerkaatst. Belichting gebeurt dus zowel direct als indirect. “Radiosity” is een belichtingstechniek die op dit principe gebaseerd is. Radiosity levert veel realistischer (en sfeervoller) beelden op. De kostprijs hiervoor is de véél zwaardere belasting van de GPU.



directe en indirecte belichting

De radiosity methode voor het genereren van 3D-beelden is gebaseerd op de fysica van de warmteuitwisseling. De thermodynamica beschrijft radiatie als energietransport vanuit een oppervlak op het moment dat dit oppervlak thermisch geëxciteerd werd. Op het moment dat een bepaald oppervlak energie opvangt heeft dit oppervlak dus op zich weer energie om zelf uit te sturen. Deze omschrijving van thermische radiatie kan ook gebruikt worden om andere vormen van energieuitwisseling te beschrijven. Lichtenergie is daar ook een voorbeeld van.

De meest eenvoudige radiosity algoritmes gaan ervan uit dat elk oppervlak licht perfect diffuus uitzenden en reflecteren (dit is uniform over het hele oppervlak). Op diffuse reflectie komen we verder trouwens nog terug. Er wordt ook van uit gegaan dat er een evenwichtssituatie bestaat. Dit is meteen ook de eindsituatie voor het algoritme.

The "radiosity vergelijking" beschrijft de hoeveelheid energie die door een oppervlak kan uitgestuurd worden als de som van de energie inherent aan het oppervlak en de energie, afkomstig van een ander object, die op het oppervlak inwerkt.

30:

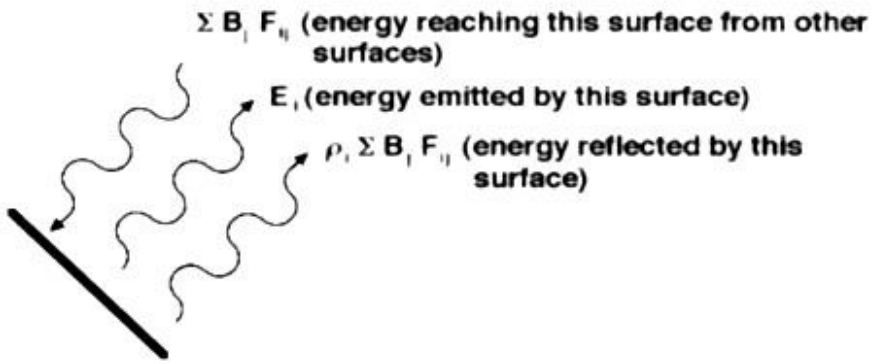
De energie die een bepaald oppervlak (oppervlak “j”) verlaat en een op een ander oppervlak (oppervlak “i”) inwerkt wordt nog eens door twee factoren beïnvloed:

- De vormfactor tussen de oppervlakken i en j, die de fysische verhouding tussen beide oppervlakken in rekening brengt.
- De reflectiviteit van het oppervlak i, dat slechts een deel van de energie die erop inwerkt zal absorberen.

De radiosity vergelijking luidt als volgt:

$$B_i = E_i + \rho_i \Sigma B_j F_{ij}$$

B_i = Radiosity of surface i
E_i = Emissivity of surface i
ρ_i = Reflectivity of surface i
B_j = Radiosity of surface j
F_{ij} = Form Factor of surface j relative to surface i



Je merkt al waarom radiosity zo rekenintensief is. Als je voor elk oppervlak rekening gaat houden met de reflectie van de reflectie van de reflectie van de reflectie van de refl… enzovoort, ben je wel een tijdje bezig met rekenen! Meestal wordt het aantal “bounces” echter beperkt tot een drietal.

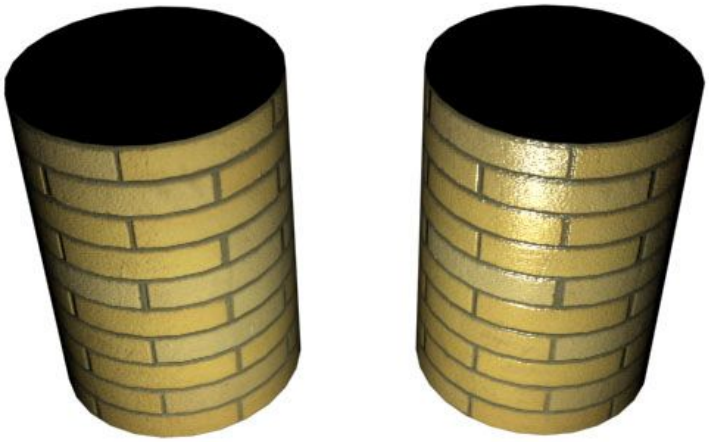
Er bestaan verschillende meer geavanceerde radiosity render-methodes die we hier niet allemaal zullen bespreken, maar eentje is interessant: de zogenaamde progressieve radiosity rendermethode. Dit algoritme rendert per “bounce” een afbeelding. Op die manier kun je tijdens het renderproces zelf beslissen wanneer het resultaat dicht genoeg aanleunt bij wat je zelf wou bereiken, en hoef je niet per se te wachten tot alle weerkaatsingen berekend zijn.

Onderstaande afbeelding illustreert dit:

van links naar rechts:
geen weerkaatsingen, 1 weerkaatsing, 3 weerkaatsingen

Reflectie

Met een goed verzorgde texturing en belichting zal een gerenderd 3D object reeds een zeer realistische indruk geven. Toch is het zo dat niet enkel het kleurpatroon bepalend is voor de “feel” van een materiaal. Reflectie en absorptie van licht speelt hierin namelijk een niet te onderschatten rol!



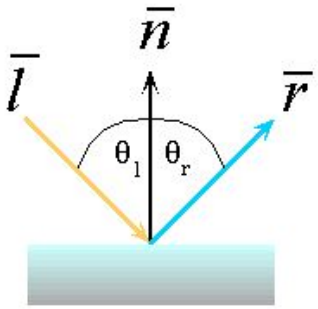
Reflectie en absorptie van licht door materialen is een relatief complexe materie, in die zin dat er geen wetenschappelijk gefundeerde formules bestaan die dit gedrag van materialen wiskundig uitdrukken. De reflectie-modellen die gebruikt worden door 3D-toepassingen zijn dan ook over het algemeen niet gebaseerd op empirische waarnemingen, maar eerder het resultaat van een zoektocht op basis van trial and error.

In wat volgt, wordt een kort overzicht gegeven van de voornaamste reflectieprincipes uit de optica, om ons daarna te focussen op één vaak gebruikt reflectie-model, het Phong model (niet te verwarren met de Phong shader!)

Ideale reflectie

Een spiegel weerkaatst elke lichtstraal die erop invalt volledig, en dit onder een hoek die gelijk is aan de invalshoek van die lichtbundel.

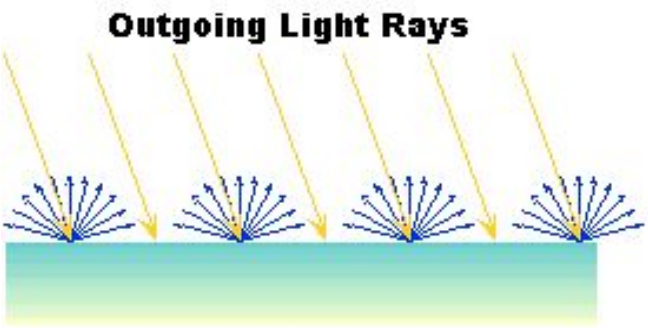
$$n_l \sin\theta_l = n_r \sin\theta_r$$



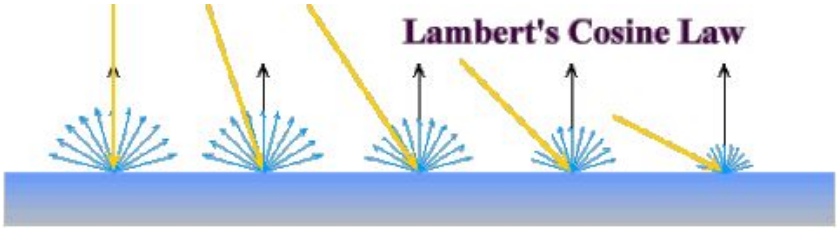
33:

Ideale diffuse reflectie

Bij ideale diffuse reflectie wordt het licht dat invalt op een oppervlak in alle richtingen evenveel weerkaatst:



Dit is een goed model voor ruwe oppervlakken. Merk op dat de hoeveelheid weerkaatst licht afhankelijk is van de invalshoek van het licht (niet van de kijkhoek):



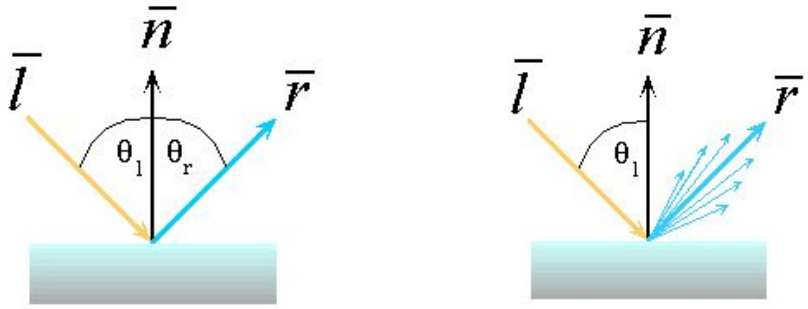
Dit wordt de wet van Lambert genoemd. De intensiteit van het weerkaatste licht is hierbij recht evenredig met de cosinus van de hoek die het invallende licht maakt met de oppervlak-normaal:

$$I \sim \cos\theta$$

Speculaire reflectie

Glanzende oppervlakken (metaaloppervlakken bijvoorbeeld) geven speculaire reflectie: een niet-ideale reflectie ergens tussen spiegelende en diffuse reflectie in. Dit vereist een aanpassing van de wet van Snellius voor spiegels:

$$n_l \sin\theta_l = n_r \sin\theta_r$$

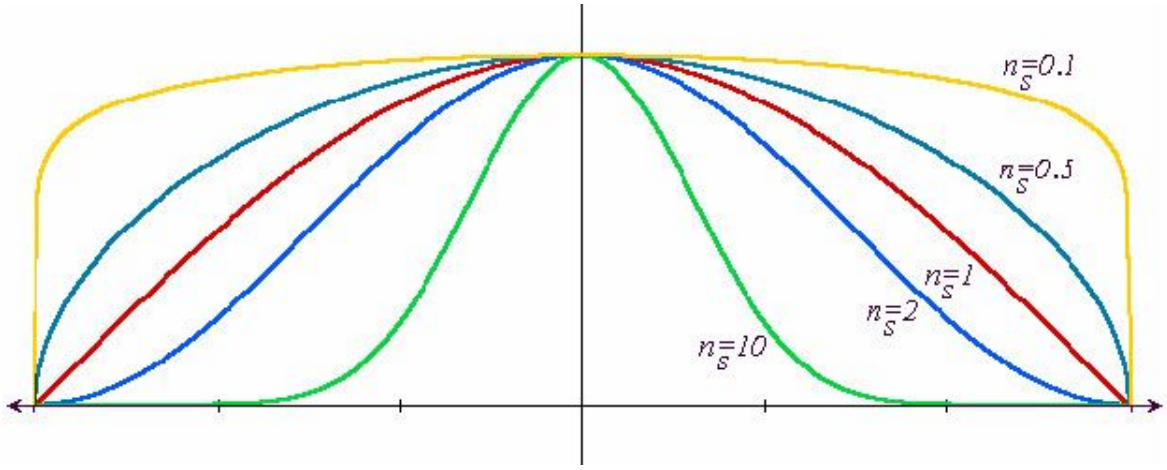


Phong reflectie

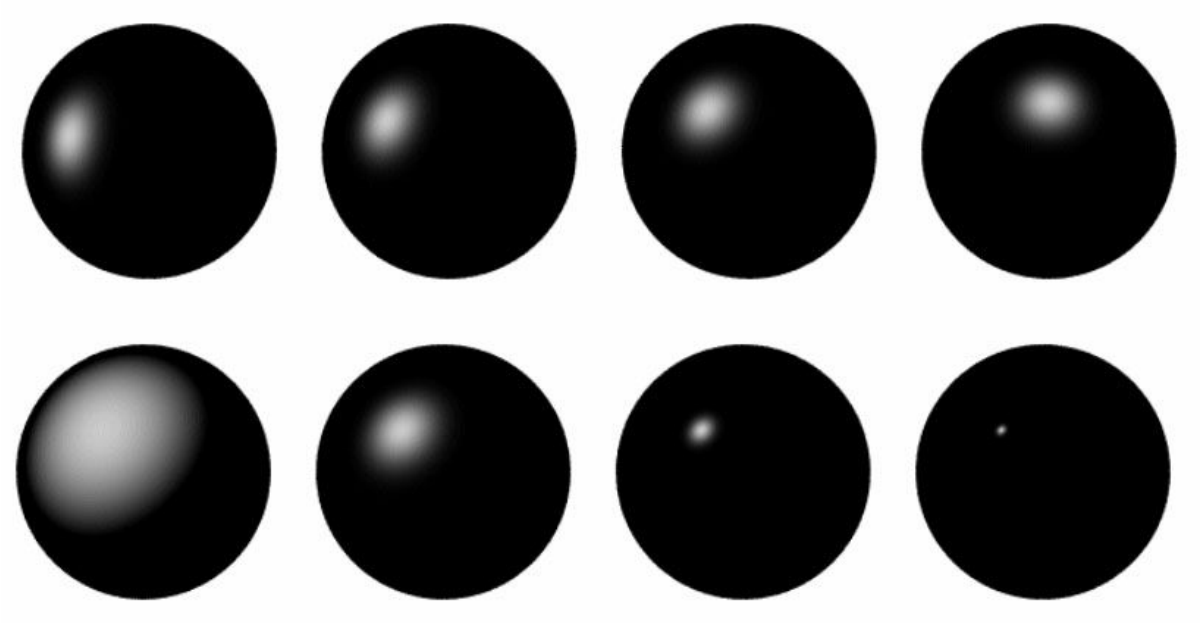
Een vaak gebruik reflectiemodel gebaseerd op speculaire reflectie is het Phong-model. De formule heeft geen basis in de fysica, maar is makkelijk bruikbaar:

$$I_{specular} = I_{light}(\cos\phi)^{n_{shiney}}$$

Door de exponent aan te passen kunnen eenvoudig verschillende effecten bekomen worden:



aanpassing van de "shiney-factor"in het Phong reflectiemodel



voorbeelden van Phong-materialen met variërende shiney-exponent

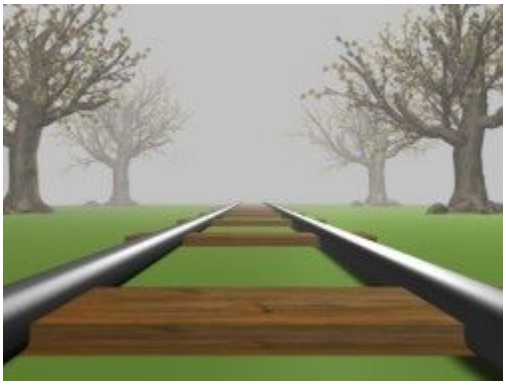
Extra effecten via pixel-shading

Atmosferische Effecten

Als door middel van shading of texture mapping de globale kleur van een pixel bekend is geworden, kunnen er aan het eind van de pixel rendering pipeline nog enkele atmosferische effecten worden op toegepast om een nog realistischer effect te krijgen. Twee veel gebruikte (en relatief simpele) atmosferische effecten zijn het 'fog' (mist) effect en het 'depth-of-field' effect.

Fog

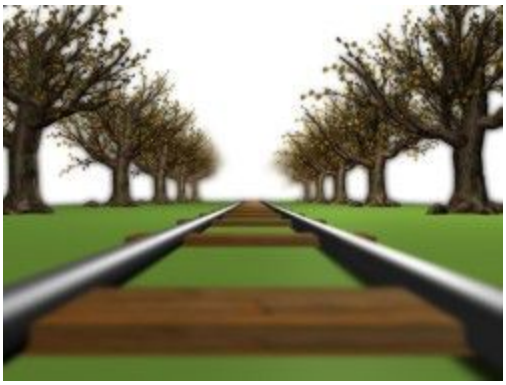
Bij het fog effect worden de kleuren steeds meer vermengd met de bepaalde fog-kleur naarmate het 3D object zich verder van de kijker bevindt. Zoals de naam het al aangeeft kan het fog effect bijvoorbeeld perfect gebruikt worden om mist of rook te simuleren.



Depth-of-field

Bij het depth-of-field effect wordt er slechts één bepaalde dieptegraad 100% scherp weergegeven. Objecten die ofwel ver weg ofwel dicht bij zijn worden door het depth-of-field effect waziger weergegeven.

Als je een foto maakt of filmt krijg je automatisch een depth-of-field effect. Bij 3D graphics moet dit opzettelijk apart worden gegenereerd.

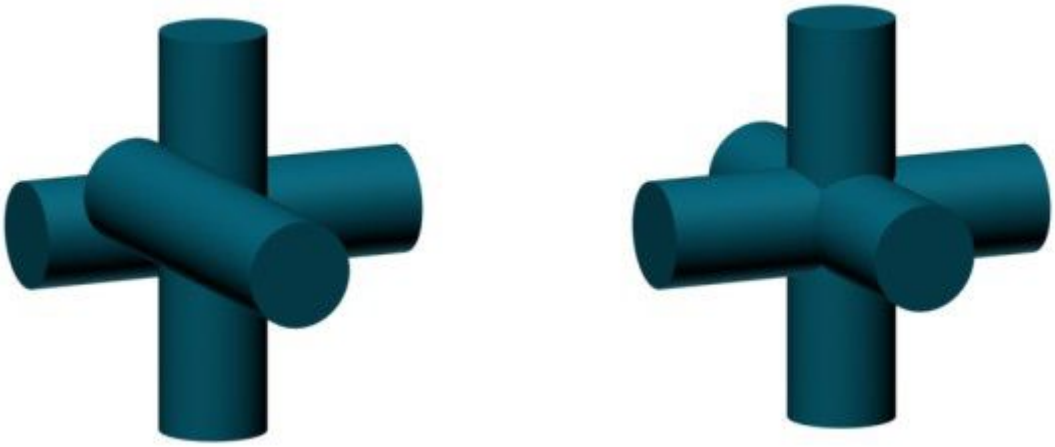


Alpha blending

Een ander effect dat we zeker niet mogen vergeten is alpha blending. Bij alpha blending kunnen objecten gedeeltelijk transparant zijn. De kleur van een pixel wordt dan voor een gedeelte vermengd met de kleur van de pixel die er achter ligt. In 3D spellen wordt bijvoorbeeld bij ramen alpha blending toegepast. Aan alle pixels kunnen zogenaamde alpha-waarden toegekend worden die aangeven hoe groot de transparatie-waarde van een pixel is.

Z-Buffering

De Z-buffer houdt voor iedere pixel bij hoe ver deze zich bevindt ten opzichte van de kijker. Dat deze Z-buffer noodzakelijk is wordt hieronder geïllustreerd. De eerste figuur is gegenereerd zonder Z-buffering, de tweede met:



Objecten die het verst weg zijn worden het eerst gerenderd en objecten die dichterbij zijn worden daarna over de verdere objecten heen geplaatst.

Dit Z-sorting systeem voldoet indien er geen objecten zijn die in elkaar overgaan. Zodra dit gebeurt moet voor elk pixel bijgehouden worden wat zijn diepte is. Deze informatie wordt opgeslagen in de zogenaamde Z-buffer. Als er nu op de plaats van een bestaande pixel een nieuwe pixel moet worden getekend, wordt deze alleen verwerkt als hij zich dichterbij bevindt dan het bestaande pixel. In de eerste figuur hierboven wordt niet nagegaan of in de Z-buffer al een pixel aanwezig is dat dichter ligt. Het resultaat is dat de cilinders "op elkaar" liggen. In de rechter figuur is de Z-buffer wél ingeschakeld, zodat de cilinders mooi in elkaar haken zoals het hoort…

Aanvullende 3D-technieken

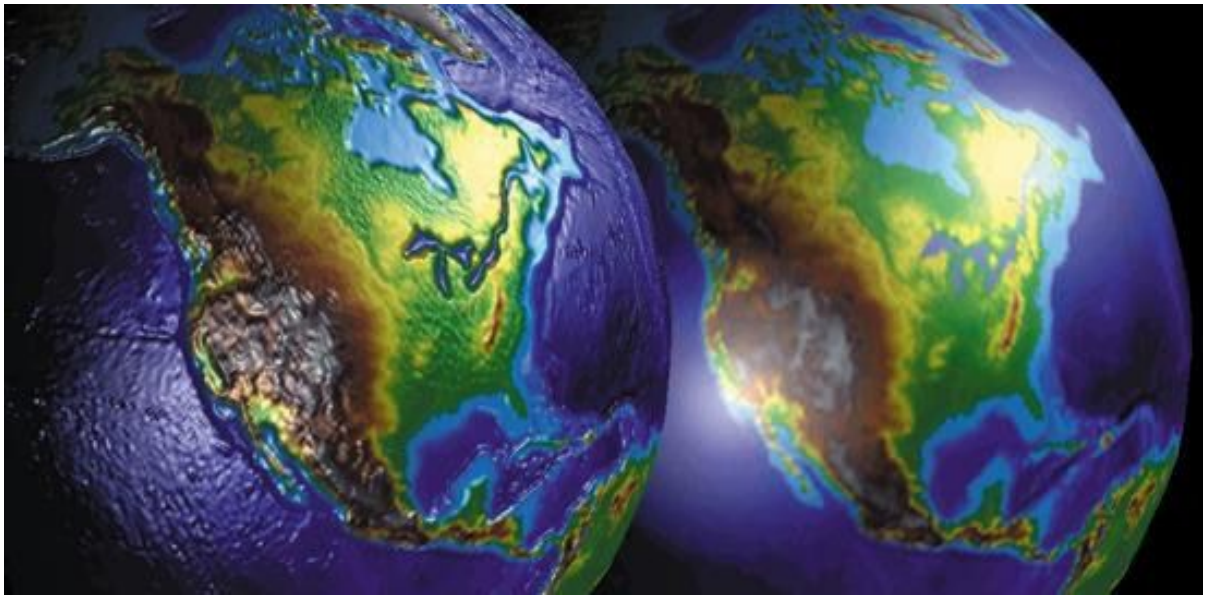
Naast het voorgaande zijn er nog heel wat 3D-technieken die gebruikt worden en vaak ook in GPU-hardware geïmplementeerd worden. Hier beschrijven we nog enkele van die technieken…

Bump mapping

We zagen al hoe het gebruik van textures voor zeer realistische resultaten kan zorgen. Probleem blijft echter dat de 3D objecten er 'plat' uit blijven zien. Met bump mapping kan er echter een soort reliëf worden gecreëerd. Dit reliëf ontstaat door de lichtuitval per pixel te wijzigen. Het object blijft plat, maar het ziet er uiteindelijk niet meer plat uit.

38:

Onderstaande figuur toont een voorbeeld van bump mapping. Rechts zien we een 3D model van de aarde. Merk dat op het oppervlak geheel plat is. Op de linker afbeelding van de aarde is bump mapping toegepast. We zien hier dan ook een duidelijke reliëfwerking. Als deze reliëfwerking gemodelleerd zou moeten worden met meerdere polygonen zou dat tientallen keren meer processorkracht vragen.



Een mooi voorbeeld van het gebruik van real-time bump mapping is het spel Expendable, waarin met deze techniek enkele prachtige wateroppervlak-effecten gegenereerd worden:

39:

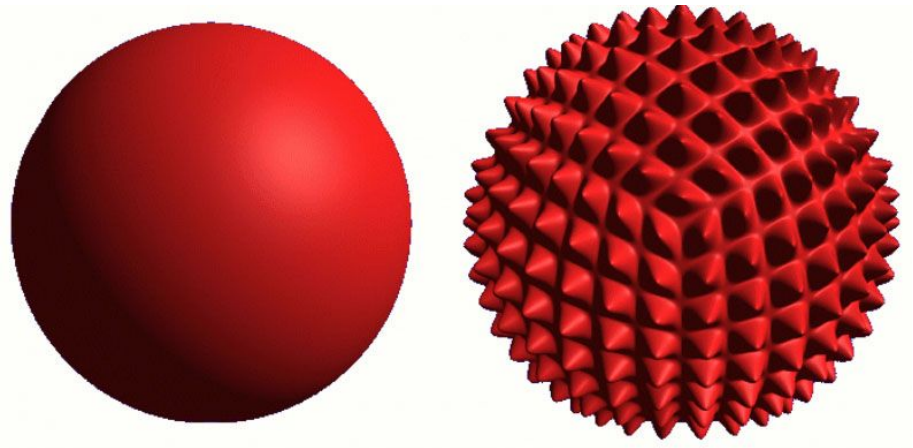
Bump mapping verandert de geometrie van de objecten niet, maar manipuleert de lichtberekeningen door de oppervlaktenormalen aan te passen. Een bump map is in wezen dan ook niet meer dan een grijswaardenafbeelding waarbij de helderheid een maat is voor de normaal-aanpassing:

Je kan op deze afbeelding ook heel goed zien dat bump mapping het object zelf niet verandert, maar enkel de indruk van reliëf opwekt. Kijk maar eens goed naar de rand van de bol… Die is nog even mooi afgerond als de bol links!

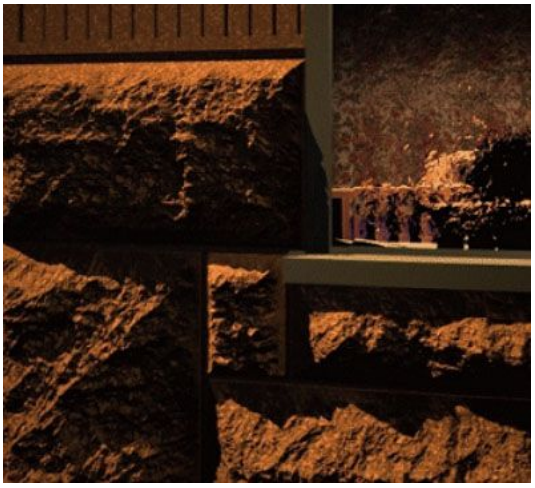
Merk op dat voor het renderen van bump map effecten wel een shader gebruikt moet worden die de belichting in elk punt berekent, zoals Phong en Gourad. Flat shading bijvoorbeeld zal weinig resultaat leveren…

Displacement mapping

Een zelfde techniek kan ook gebruikt worden om de geometrie van een object aan te passen. Deze geometrie-omrekening moet vanzelfsprekend wel gebeuren vóór de belichtingsberekeningen. Net als bij bump mapping is ook een displacement map in essentie een grijswaardenafbeelding die de vervormingsinformatie van het object bevat. Bemerk ook hoe het silhouet van de bol dit keer wél aangepast is:



Onderstaande render toont de krachtige mogelijkheden van displacement mapping. Met bump mapping zouden de schaduwen niet op de naburige stenen vallen:



Dat displacement mapping véél meer rekenkracht vergt dan bump mapping is duidelijk!

Texture compression

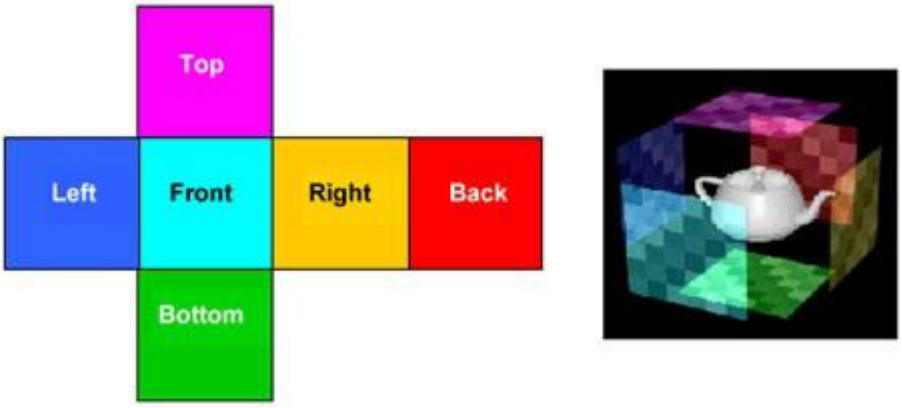
Het ligt voor de hand dat hoe gedetailleerder de texturen zijn, hoe mooier de uiteindelijke 3D beelden kunnen zijn. Erg gedetailleerde (en dus erg grote) texturen vragen echter zeer veel geheugenbandbreedte. Het is daarom niet altijd mogelijk om veel gedetailleerde texturen in één 3D scène te gebruiken. Texture compression verzacht dit probleem.

Bij deze techniek worden texturen gecomprimeerd opgeslagen en uitgewisseld met de GPU. De vereiste bandbreedte kan zo tot 8 keer worden verkleind. Texture compression maakt het dus mogelijk om zeer detailrijke textures te gebruiken in games, iets wat het realisme uiteraard ten goede zal komen. Nadeel van dit systeem is dat het decomprimeren van de textures een extra rekenstap inhoudt die met ongecomprimeerde textures niet nodig is.

Cube Environment Mapping

Cube environment mapping was tot voor kort enkel geïmplementeerd in professionele 3D rendering software als 3D Studio Max. Voor real-time toepassing van environment mapping zijn de grafische kaarten pas sinds enkele jaren krachtig genoeg.

De werking van cube environment mapping komt in principe op het volgende neer: het 3D object waar een weerspiegeling op moet komen wordt als het ware "ingepakt" in een kubus. Vanuit alle zes de kanten van de kubus wordt vanuit het 3D object naar buiten gekeken en wordt de 3D scène nogmaals opnieuw gerenderd. Zo krijg je een 100% realistische weerspiegeling vanuit die zes richtingen. Weerspiegelingen die niet precies vanuit één van deze zes richtingen komen, worden door interpolatie bepaald.



Het handige van een kubus is dat vanuit één punt altijd maar maximaal drie kanten zichtbaar zijn (tenzij natuurlijk er meer weerspiegelende object bij elkaar staan). Meer dan drie extra beelden hoeven dus in de meeste gevallen niet gerenderd te worden.

De scène hieronder uit de film Terminator 2 (1991) is een mooi voorbeeld van deze techniek. Hier wordt als environment map overigens geen gerenderd beeld gebruikt, maar wel een gefilmd beeld, zodat het gezicht van de acteur in de metaalachtige massa weerspiegeld wordt.

43:

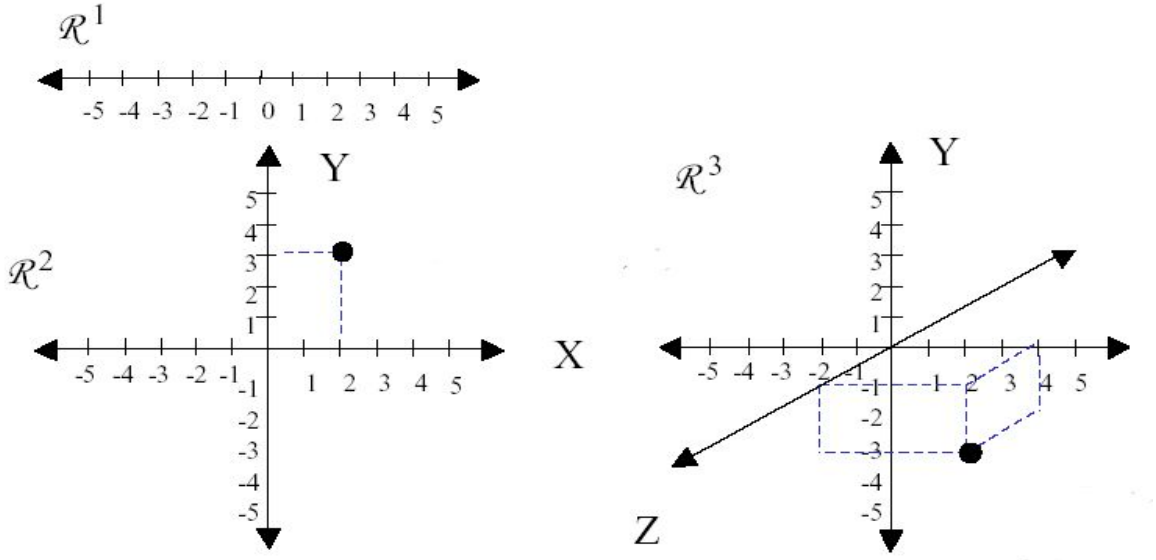
Punten – Coördinaten – Matrixvoorstelling

In dit deel gaan we wat dieper in op de wiskunde die bij de grafische toepassing komt kijken en waarvan de GPU een flink deel op zich neemt. De bedoeling in de context van dit vak is om je alvast een klein tipje van de wiskundige sluier op te lichten. Dit deel behandelt bijvoorbeeld geen berekeningen van licht, berekeningen van effecten, enz… We behandelen enkel de (relatief eenvoudige) basisbewerkingen die nodig zijn voor het bewegen en bekijken van 3D-objecten.

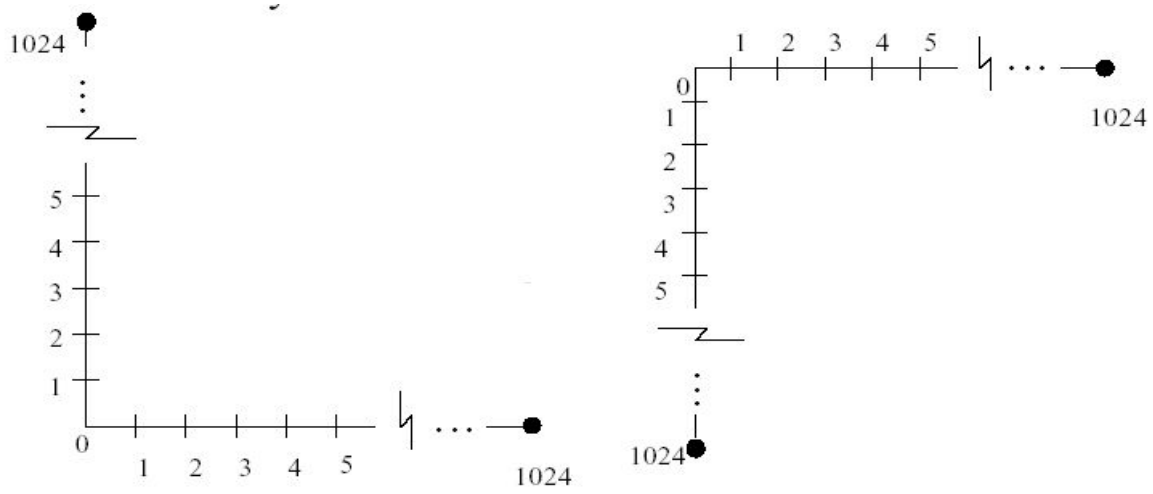
Vooraleer we écht van start gaan wordt het een en ander nog even opgefrist. Het betreft zaken die je zeker moet begrijpen om de rest van dit hoofdstuk te snappen (vooral rond vectoren).

Even opfrissen: Carthesisch coördinatenstelsel

Onderstaand schema toont Carthesische coördinatenstelsels voor één-, twee- en driedimensionale werelden.

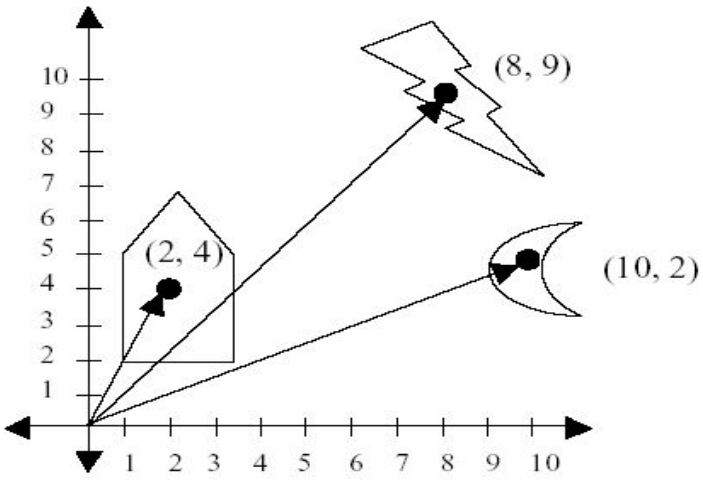


Merk op dat er een groot verschil zit tussen zo'n coördinatenstelsel, en het assenstelsel van een computerscherm. Ten eerste werkt een scherm met gehele waardes (pixels), en niet met reële zoals in de stelsels hierboven. Ten tweede is de zin van de Y-as bij een scherm omgekeerd: de oorsprong bevindt zich namelijk steeds in de linker bovenhoek van het scherm!



Wat is een vector?

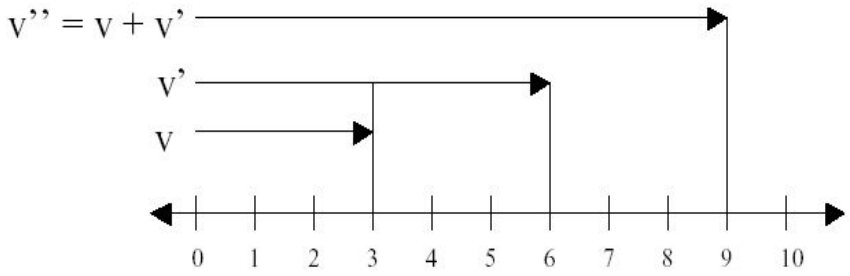
Een vector (in deze context) bepaalt de plaats van een punt ten opzichte van een vooraf gekozen referentiepunt (de oorsprong). Een vector wordt bepaald door zijn lengte, richting en zin. Een vector kan worden voorgesteld door een kolommatrix.



45:

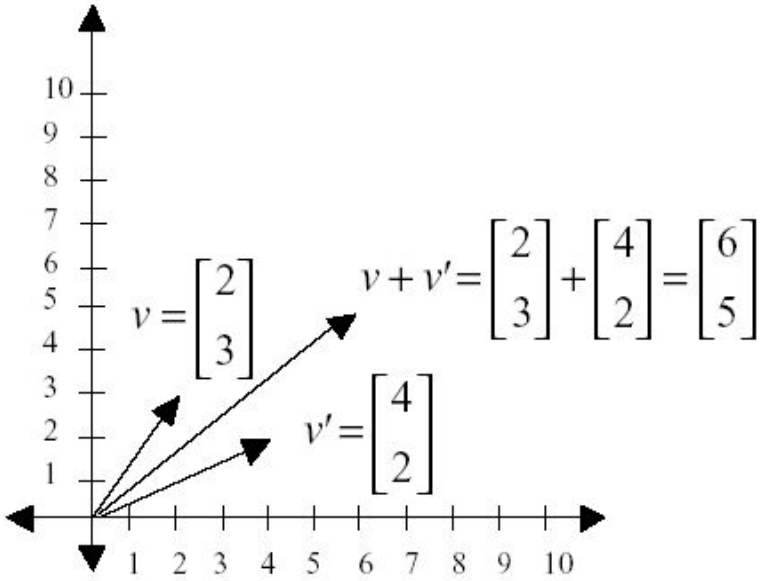
Bewerkingen met vectoren

Eendimensionaal optellen van vectoren:

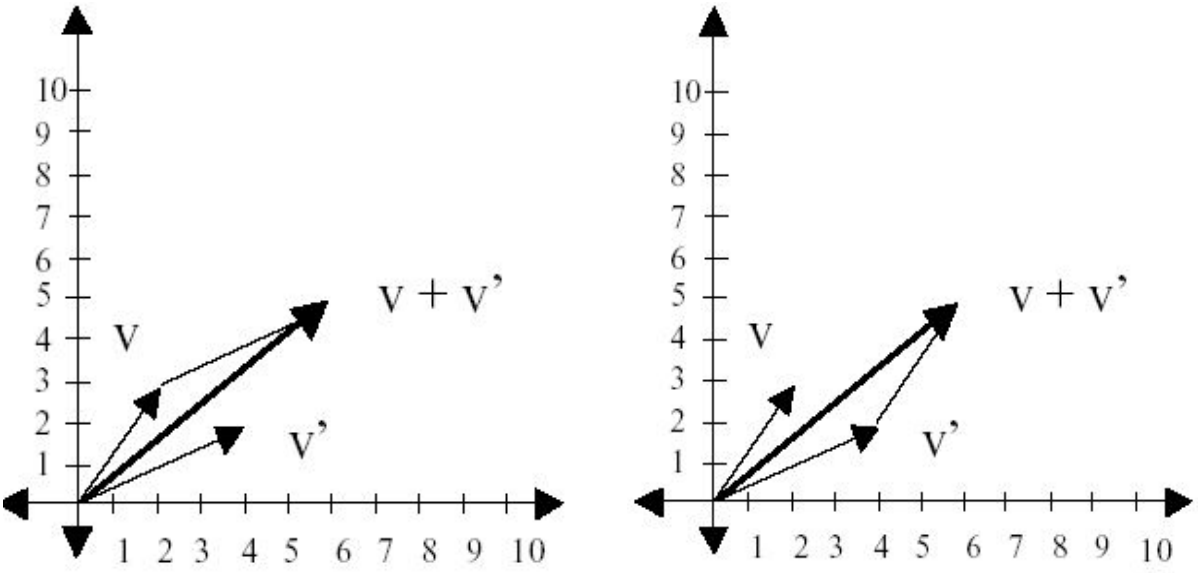


v = [3], v' = [6], v'' = [9]

Tweedimensionaal optellen van vectoren:

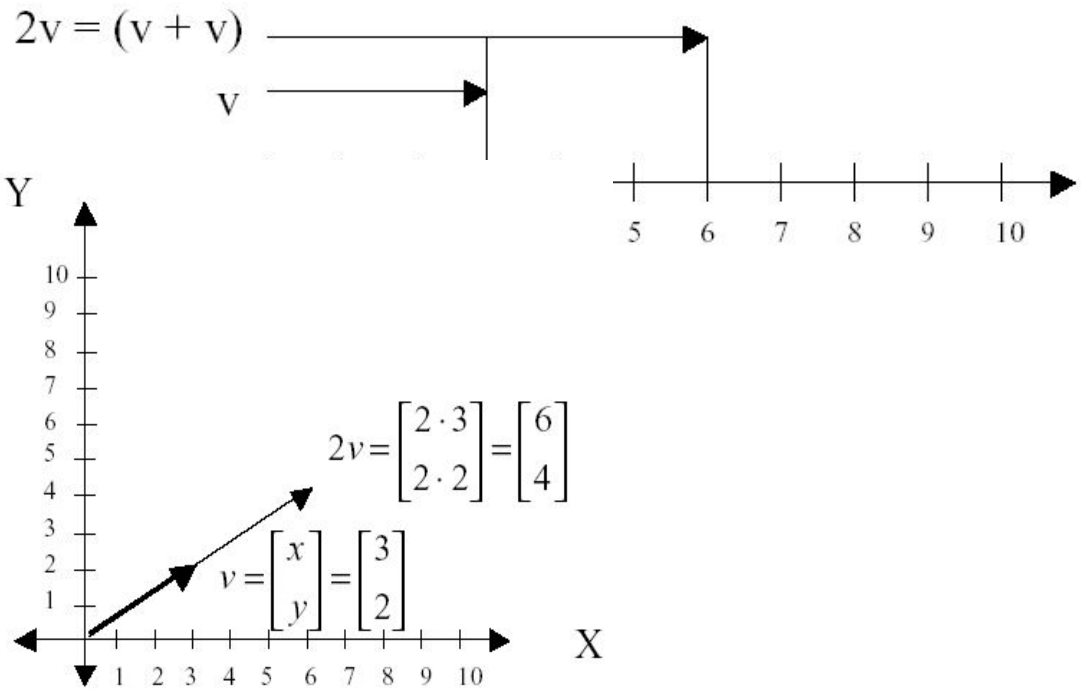


De som van twee tweedimensionale vectoren kan ook gemakkelijk geconstrueerd worden met behulp van de zogenaamde paralellogramregel:



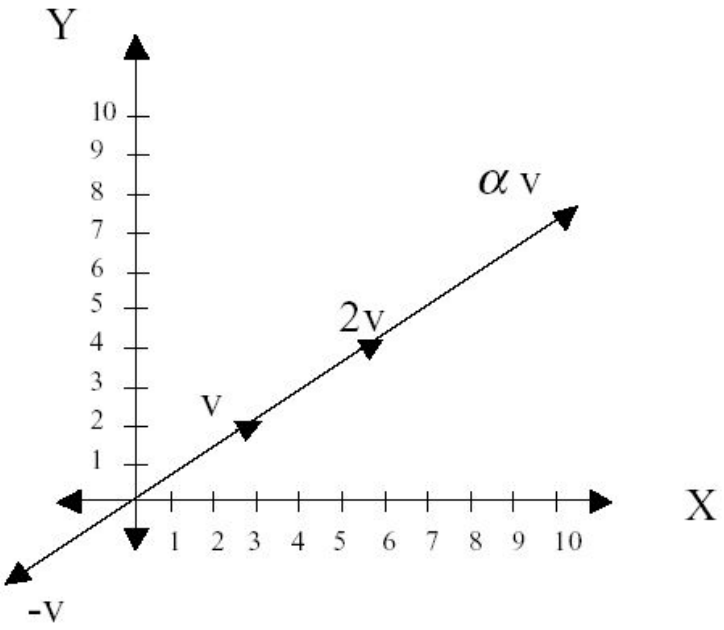
De figuur toont duidelijk dat het optellen van vectoren commutatief is (v+v' = v'+v).

Product van een vector met een getal (1D en 2D):



Lineaire afhankelijkheid van vectoren

Twee vectoren zijn lineair afhankelijk als de ene vector een (reëel) veelvoud is van de andere. De verzameling van alle veelvouden van een vector is een rechte door de oorsprong:



Basisvectoren van het vlak

De basisvectoren van een vlak worden gedefinieerd als de eenheidsvectoren (lengte = 1) die op x en y-as liggen. Elke vector in het vlak kan worden uitgedrukt als de som van veelvouden van deze basisvectoren:

$$\begin{bmatrix} x \\ y \end{bmatrix} = x.\begin{bmatrix} 1 \\ 0 \end{bmatrix} + y.\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

48:

Transformaties

Soorten transformaties

Afhankelijk van bepaalde eigenschappen van transformaties, kunnen we drie grote groepen onderscheiden:

- **Lineair transformatie:** behoudt evenwijdigheid van lijnen. De nulvector wordt altijd getransformeerd in de nulvector. Voorbeelden zijn het schalen en de rotatie.

- **Affiene transformatie:** behoudt evenwijdigheid van lijnen. De nulvector wordt niet altijd op de nulvector geprojecteerd. Voorbeelden: schalen, rotatie (alle lineaire transformaties zijn immers ook affien), translatie.

- **Projectie:** evenwijdigheid wordt niet noodzakelijk behouden. Voorbeeld: het omrekenen van een 3D-scène naar het 2D beeldscherm. Ook alle affiene transformaties zijn projectief.

Een transformatie kan wiskundig worden uitgedrukt met behulp van een matrix. Het product van deze zogenaamde “transformatie-matrix” met een vector geeft het beeld van deze vector. In wat volgt worden de belangrijkste transformaties besproken, telkens met een klein rekenvoorbeeldje. We bespreken ook hoe we transformaties kunnen combineren.

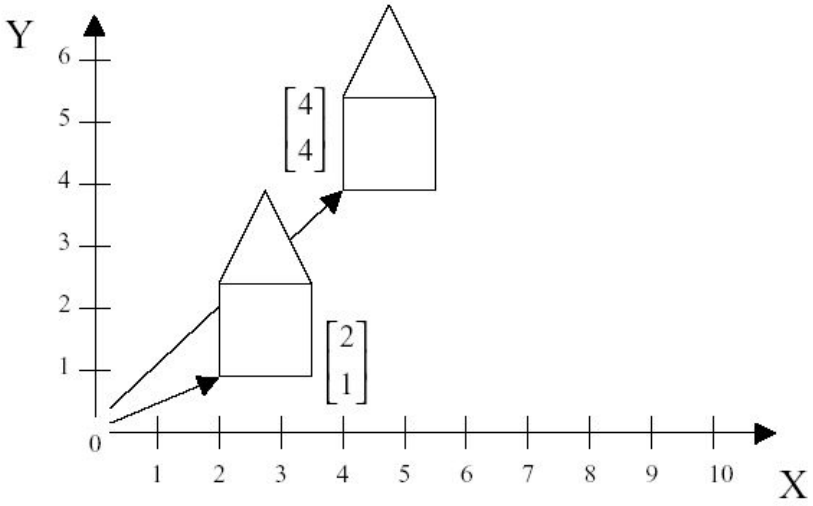
Opgelet: voor de eenvoud hebben we het eerst over transformaties en vectoren in het vlak (dus in 2 dimensies).

Translatie

De translatie wordt ook wel verschuiving genoemd. Ze is meteen al een uitzondering op de regel dat het beeld van een vector na transformatie berekend kan worden door het product van een transformatiematrix met de vector. Een translatie wordt namelijk uitgedrukt door gebruik te maken van een translatievector die bij de te verschuiven vector wordt opgeteld:

$$\vec{v}' = \vec{v} + \vec{t} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix} \qquad \begin{aligned} x' &= x + dx \\ y' &= y + dy \end{aligned}$$

dx = 2
dy = 3

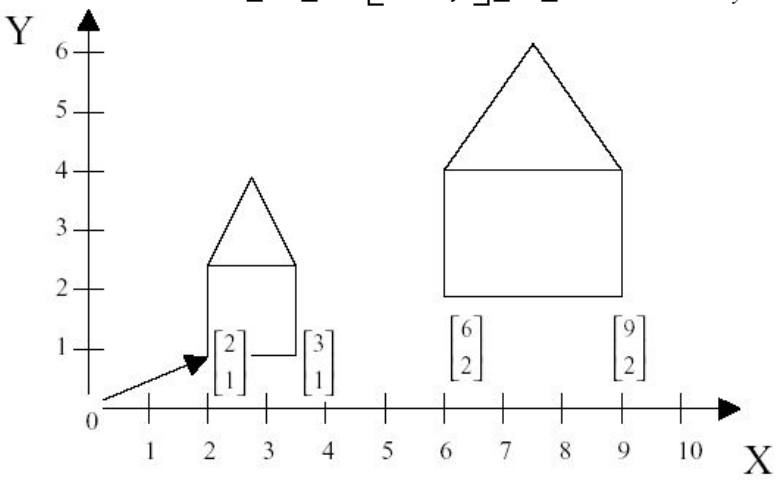


Schalen

Bij het schalen maken we onderscheid tussen de schaalfactor in de x- en de y-richting. Merk op dat het schalen gebeurt ten opzicht van de oorsprong!

$$v' = S.v \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{aligned} x' &= s_x.x \\ y' &= s_y.y \end{aligned}$$

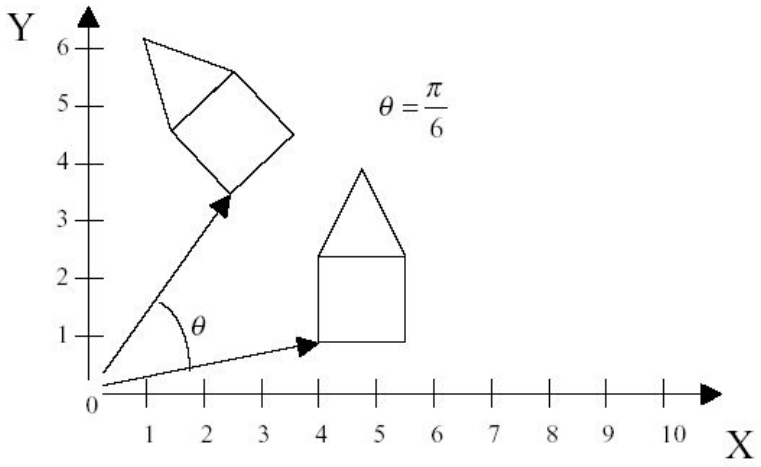
$s_x = 3$
$s_y = 2$



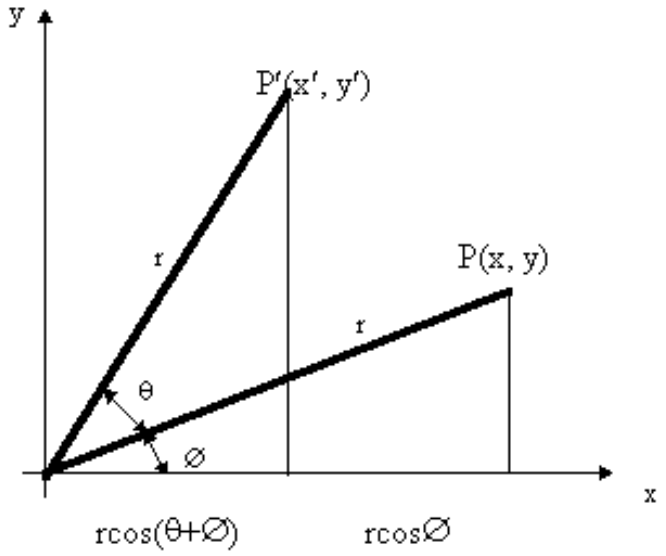
Rotatie

Ook de rotatie gebeurt ten opzichte van de oorsprong:

$$v' = R_\theta . v \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{aligned} x' &= x.\cos\theta - y.\sin\theta \\ y' &= x.\sin\theta + y.\cos\theta \end{aligned}$$



De vorige twee transformaties waren makkelijk te begrijpen, maar waar komt de rotatiematrix precies vandaan? Laat ons de formule eens opbouwen:



De figuur toont de rotatie van de vector P rond de oorsprong. Uit de figuur kan gemakkelijk het volgende worden afgeleid:

$$x = r\cos\phi, y = r\sin\phi$$
$$x' = r\cos(\theta+\phi) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\theta+\phi) = r\cos\phi\sin\theta - r\sin\phi\cos\theta$$

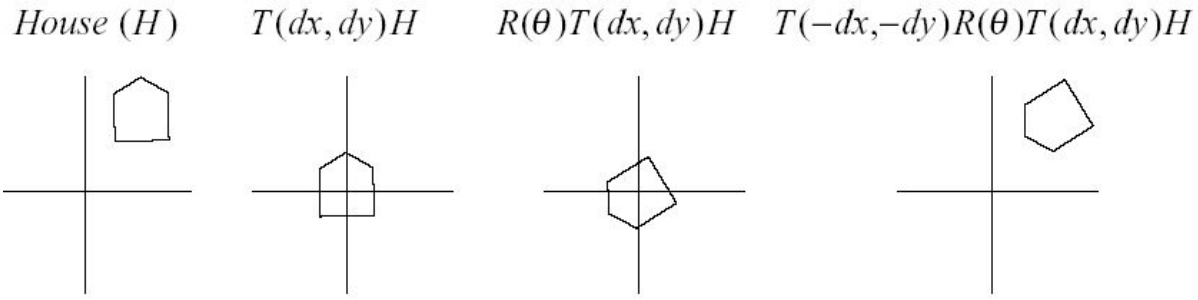
Na substitutie krijgen we:

$$x' = x.\cos\theta - y.\sin\theta$$
$$y' = x.\sin\theta + y.\cos\theta$$

Compositie

We vermeldden eerder al dat het schalen en het roteren relatief ten opzichte van de oorsprong gebeuren. In de vorige paragraaf bijvoorbeeld, lijkt het alsof het huis dat we roteren ook verplaatst wordt. Stel dat we dit object nu eens rond zijn centrum willen roteren (zodat het ter plaatse blijft), hoe kunnen we dit dan bereiken?

De oplossing bestaat in het samenstellen van transformaties tot we het gewenste resultaat bereikt hebben. We kunnen het huis bijvoorbeeld eerst verplaatsen naar de oorsprong, daarna roteren, en daarna terugplaatsen op z'n oorspronkelijke plaats:



Dergelijke samenstelling van transformaties noemen we ook een *compositie*. Als we zo'n compositie wiskundig willen uitdrukken komen we wel in de problemen. We zagen al dat niet alle transformaties op dezelfde manier worden uitgedrukt:

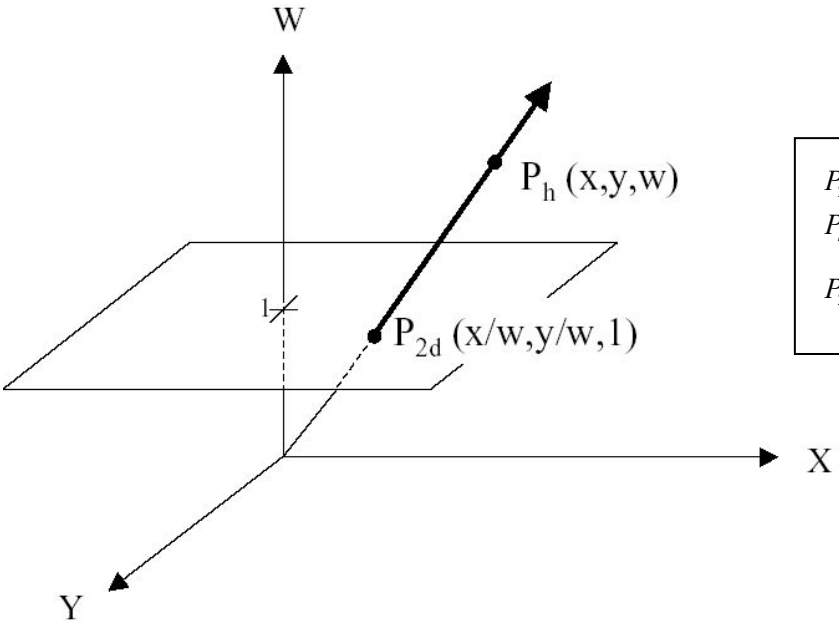
v' = v + t
v' = S v
v' = R v

Het ware veel handiger als we ook de translatie als een product van een matrix en een vector konden uitdrukken…

De homogene transformatiematrices

Homogene coördinaten

En dat kan… als we gebruik maken van homogene coördinaten! Hierbij wordt bij elke vector een extra coördinaat toegevoegd. De figuur hieronder illustreert het principe van homogene coördinaten. We gaan er verder niet dieper op in.



$$P_{2D}(x,y) \rightarrow P_h(wx,wy,w), w \neq 0$$
$$P_h(x',y',w), w \neq 0$$
$$P_{2D}(x,y) = P_{2D}\left(\frac{x'}{w},\frac{y'}{w}\right)$$

In homogene coördinaten wordt ook de translatie uitgedrukt met behulp van een matrix:

$$T(dx,dy) = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = T(dx,dy)v$$

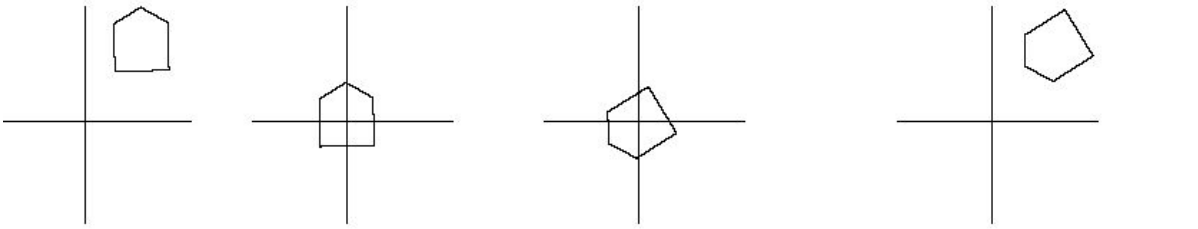
$$S(s_x,s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = S(s_x,s_y)v$$

$$R(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = R(\phi)v$$

Compositie revisited

Laten we nog even terugkeren naar het compositie-voorbeeld uit de vorige pagina. Het wordt nu duidelijk dat het samenstellen van transformaties in wezen niets anders is dan het vermenigvuldigen van de respectievelijke transformatiematrices:

$House\ (H)$ $T(dx,dy)H$ $R(\theta)T(dx,dy)H$ $T(-dx,-dy)R(\theta)T(dx,dy)H$



Merk hierbij op dat de eerste transformatie *laatst* in de matrixvermenigvuldiging staat. Verwissel de matrices zeker niet van plaats! Het vermenigvuldigen van matrices is immers niet commutatief! Probeer in onderstaand voorbeeld maar eens eerst de rotatie uit te voeren en dan pas de schaling…3D-transformaties

3D transformatie matrices

Tot hiertoe bespraken we enkel tweedimensionale transformaties. In drie dimensies gebeurt alles analoog. Hieronder worden de voornaamste homogene transformatie matrices opgesomd voor driedimensionale bewerkingen. Merk op dat er drie rotatiematrices zijn. De rotatie kan namelijk gebeuren rond één van de drie assen (daar waar we in twee dimensies enkel kunnen roteren rond een punt van het vlak).

translatie

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

schalen

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotatie rond de X-as

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotatie rond de Y-as

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotatie rond de Z-as

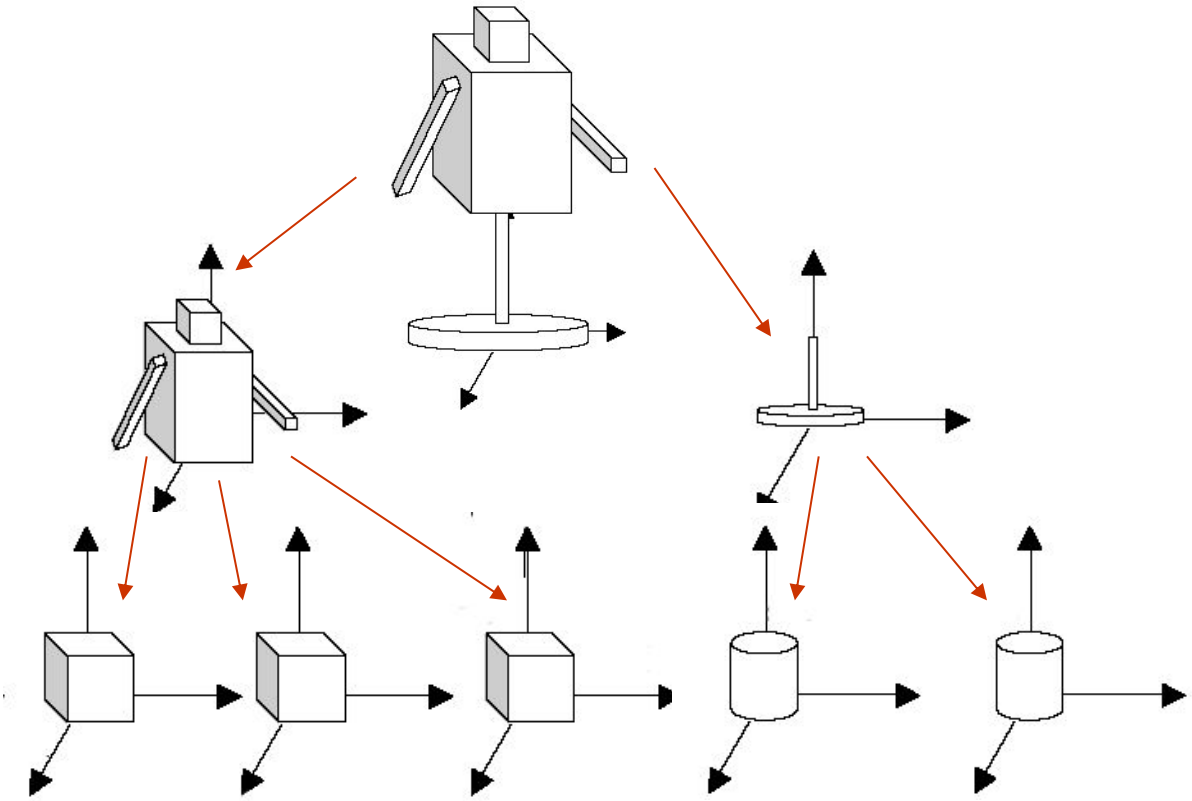
$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hiërarchische 3D-scènes

Bij het berekenen van 3D-scènes moet rekening gehouden worden met de relaties tussen de verschillende objecten.

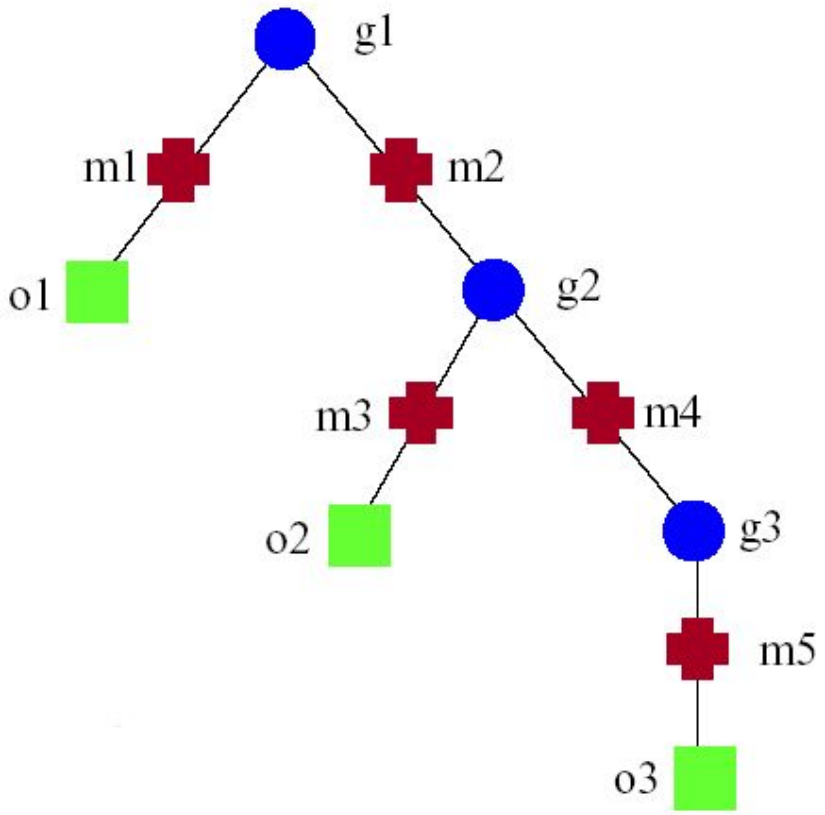
Onderstaand schema toont een vrij eenvoudig 3D-model waarin toch al heel wat matrixcomposities berekend moeten worden:

55:



Het lichaam van dit figuurtje is opgebouwd uit kubussen die elk op zich vervormd zijn via niet uniforme schaling (op zich al telkens een compositie). De armpjes zijn dan nog eens geroteerd ten opzichte van het lichaam. Het lichaam kan draaien ten opzichte van de voet. Dit houdt in dat als het lichaam draait, ook de arm moet meedraaien relatief ten opzichte van de aansluiting van lichaam en voet.

Onderstaand schema verduidelijkt het een en ander. Een *g* staat voor een groep objecten. Een groep is *parent* van zogenaamde *child-objects*. Zo'n child kan opnieuw een groep zijn of een object. Een *o* staat voor een object. Een object bevindt zich altijd op het einde van een tak, en heeft zelf geen child objects. De *m* in het schema staat voor de matrix die instaat voor de transformatie relatief ten opzichte van de vorige knoop in de boomstructuur.


g1
m1
m2
o1
g2
m3
m4
o2
g3
m5
o3

Raamwerken voor het ontwikkelen van 3D-toepassingen

Wanneer je zelf een toepassing wil ontwikkelen, zal je in praktijk gebruik maken van één of meerdere frameworks/API's die de toegang tot de hardware van de videokaart en de manipulaties voor 3D-graphics flink vereenvoudigen.

DirectX (enkel Windows)

DirectX is ontwikkeld door Microsoft om onder Windows het schrijven van performante toepassingen mogelijk te maken die gebruik maken van geavanceerde hardware (3D-graphics, geluid, netwerk, controllers), vooral gericht naar Games, 3D-graphics, Geluidbewerking enz.

Het gedeelte van DirectX dat rechtstreeks gericht is op 3D-Graphics heet Direct3D.

Het is een zeer krachtige API die zeer weinig beperkingen oplegt aan de programmeur, maar ook zeer uitgebreid is en daarom wel wat studiewerk vraagt.
De originele DirectX-bibliotheken zijn geschreven voor gebruik in C++ en bedoeld voor het ontwikkelen van 'unmanaged' toepassingen (buiten het .Net framework). Voor het ontwikkelen van 3D-toepassingen in bv. C#, is er een namespace beschikbaar gemaakt die de DirectX beschikbaar maakt binnen de .Net omgeving. Er wordt dan gesproken over 'Managed DirectX'. Ten voordele van XNA Game Studio Express, wordt MDX de managed DirectX echter niet meer ondersteund.

OpenGL (Open, Cross-platform)

OpenGL is origineel ontwikkeld oor 'Silicon Graphics Inc (SGI)' en wordt nu beheerd door de Khronos Group (een non-profit industrieconsortium dat tot doel heeft Open API's te beheren).

OpenGL is iets eenvoudiger en abstracter dan Direct3D, maar doordat het iets verder van de hardware staat, zal het niet altijd de laatste technologie van je videokaart ondersteunen. Daarnaast is het de laatste jaren iets minder duidelijk in welke richting de API evolueert.

OpenGL wordt ook in Windows ondersteund wanneer ook de driver van de grafische kaart dit ondersteunt. Om vanuit C# gebruik te maken van de OpenGL API, kan je bv. gebruik maken van de 'OpenTK' bibliotheken (http://opentk.sourceforge.net/)

Ook met DirectX en OpenGL blijft het nog een flinke uitdaging om een serieuze werkende 3D-toepassing te schrijven. Meestal wordt een framework gebruikt dat nog eens bovenop DirectX en .Net of OpenGL draait en dat je heel wat functionaliteit levert die geoptimaliseerd is voor Game-ontwikkeling.

Windows Presentation Foundation (WPF)

WPF is door Microsoft ingevoerd met Vista als een alternatief voor de klassieke Winforms voor het ontwikkelen van native Windows toepassingen. De concepten in WPF zijn moderner dan bij Winforms, zijn gebaseerd op een scheiding van user interface van achterliggende code en het beschrijven van die user interface via een XML-gebaseerde taal (XAML). WPF maakt het veel gemakkelijker om complexe grafisch aantrekkelijke en interactieve interfaces te ontwikkelen. Binnen WPF is ook ondersteuning voor 3D-graphics voorzien (WPF gebruikt trouwens onderliggend DirectX voor de graphics rendering).

Vanuit de bedoeling van WPF is het zeker mogelijkheid en relatief eenvoudig (tenminste zodra je een beetje overweg kan met WPF) om daarbinnen met 3D-graphics te werken, maar vanwege de extra overhead die het WPF raamwerk meebrengt is het zeker niet geschikt voor interactieve rendering van complexe 3D-scenes.

Een typische 3D renderloop.

Het framework dat je gebruikt zal je via een aantal methodes toelaten allerlei bewerkingen te doen, onder andere het tekenen van grafische primitieven en triangles (op basis van hun vertices).

Met die methoden zou je dus vanuit je code een complete en complexe 3D-scene kunnen tonen aan de gebruiker. In praktijk is het echter meestal de bedoeling dat je 3D-toepassing in real time het beeld opnieuw berekent: bv. omdat een CAD-ontwerper een object roteert om het van een andere kant te bekijken, of omdat er in een spel objecten zijn die bewegen ten opzichte van elkaar of de camera.

In die situaties is het dus nodig om in een continue lus telkens opnieuw de nodige parameters te wijzigen (oriëntatie, posities, …) en het hele beeld te herberekenen in een nieuw frame dat aan de gebruiker getoond wordt. Om een stabiel beeld te genereren dat een vloeiende beweging zonder geen flikkering geeft, zal je minimaal zo'n 30 tot 60 frames per seconde moeten berekenen en tonen. Zeker bij complexe scenes zal de efficiëntie en performantie van je code dus belangrijk zijn.

De algemene structuur van je programma is dan ook relatief eenvoudig:

```
Initialisatie();
While (NotTheEnd)  //global game loop
{
     ProcessUserInteraction();
       CalculateNewScene();
              DisplayScene();
}
      CleanupResources();
```

ProcessUserInteraction:

In een interactieve toepassing zal je regelmatig moeten controleren of er input van de gebruiker beschikbaar is en daaraan eventueel de nodige acties koppelen: beweging van de camera, pauzeren of herstarten, schieten, enz.

CalculateNewScene:

Hierin bereken je voor alle objecten in je 3D scene de nieuwe positie, oriëntatie, kleur…
Je houdt daarbij rekening met eventuele invoer van de gebruiker, interactie tussen objecten (collision-detection) en de basiswetten van de fysica (om een realistisch effect te krijgen).

DisplayScene:

Deze methode zorgt voor de rendering van de 3D-scene. Alle objecten worden doorgegeven naar de grafische driver, rekening houdend met de ingestelde belichting en de nodige transformaties en voor een correcte weergave.
Om conflicten met de graphics hardware te vermijden, wordt meestal gewerkt met twee versies van de displaybuffer: de hardware toont de inhoud van één buffer op het scherm, terwijl het volgende frame getekend wordt in de zogenaamde 'backbuffer'. Wanneer het nieuwe frame klaar is, worden beide buffers omgewisseld en kan gestart worden met het renderen van het daarop volgende frame….

Programmavoorbeeld WPF

Zie intro – slides + toegevoegde voorbeelden op toledo

60:

Zie ook:

- http://msdn.microsoft.com/en-us/library/ms754130.aspx

- http://wpftutorial.net/Home.html
- http://wpf-rnv.blogspot.be/2008/01/wpf-3d-basics.html