Ray Tracing op GPU en CPU

Yannick Francken

6 juni 2005

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van codefragmenten

Samenvatting

In deze thesis wordt beschreven hoe een klassieke ray tracer herschreven kan worden voor uitvoering op de laatste generatie grafische hardware. Hiervan is een voorbeeldimplementatie gerealiseerd in Cg alsook een analoge versie in C++, die gebruikt wordt voor een vergelijkende studie.

De Cg versie blijkt tot meerdere malen sneller te zijn dan de hieruit afgeleide C++ versie. Vooral bij het renderen van hogere resoluties is er een beduidende snelheidswinst op de grafische kaart. Dit is naar alle waarschijnlijkheid te wijten aan een meer coherente program flow van de in parallel lopende programma's (fragment shaders). Wat ook een niet te verwaarlozen rol speelt in het voordeel van de Cg versie, is het feit dat de ray tracer geschreven en geoptimaliseerd is in Cg, en daarna pas vertaald naar C++.

Indien de grafische kaarten blijven evolueren zoals dat nu het geval is, zal het in de toekomst mogelijk worden om een aanzienlijke snelheidswinst te boeken inzake ray tracing. Het verwezenlijken van een ray tracer op de GPU zal ook eenvoudiger worden vanwege de versoepeling van de huidige beperkingen van enerzijds de grafische kaart, en anderzijds de huidige compilers.

Dankwoord

Dit werk zou niet tot stand gekomen zijn zonder de hulp van bepaalde mensen. Deze zou ik hiervoor dan ook graag dank toe zeggen.

Phillipe Bekaert bedank ik voor het promoten van deze thesis. Door hem heb ik interessante wegen weten te vinden doorheen het doolhof van de literatuur. Ik heb vrij mijn weg kunnen bepalen, en dit op basis van nuttige hints van de promotor, zonder dat er mij meteen een bepaalde uitgang gewezen werd. Hartelijk dank hiervoor.

Bert De Decker en Erik Hubo, de copromotors van deze thesis, ben ik ook uitermate dankbaar voor hun hulp. Bij hen kon ik steeds terecht voor het stellen van vragen, het voeren van opbouwende discussies en het verkrijgen van allerlei "tips and tricks" voor het maken van een thesis. Heel erg bedankt hiervoor.

Hoewel Tom Mertens geen rechstreekse begeleider was voor deze thesis, ben ik meermaals met vragen bij hem terecht gekunnen, waarvoor ook mijn dank.

Verder zijn er nog verscheidene collega's die mij geholpen hebben bij het tot stand brengen van deze thesis. Hierbij denk ik meer in het bijzonder aan Nicolas Barbier, Johan Huysmans, Bart Cornelissen, Rob Van Dyck en Dirk Vanden Boer. Door de gevoerde gesprekken, ben ik vaak tot nieuwe inzichten gekomen, waarvoor ik hen zeer dankbaar ben.

Mijn vriendin, Tanja, ben ik ook zeer dankbaar voor de geleverde steun. Op de moeilijkere momenten kon ze me steeds weer motiveren en stimuleren om met volle moed verder te zetten.

Uiteindelijk zou ik graag nog alle niet vernoemde mensen, die in meer of mindere maten hebben bijgedragen tot deze thesis, bedanken. Ik denk hierbij aan mijn ouders, familie, vrienden, docenten, en nog vele anderen.

Hoofdstuk 1

Inleiding

De laatste jaren heeft de grafische hardware een gigantische evolutie gekend. Een belangrijke stuwende kracht hierbij is de game-industrie. Deze hardware is niet meer enkel geschikt voor het rasteriseren van driehoeken, maar wordt meer en meer gebruikt als algemeen programmeerbare hardware. Dit is onder andere te wijten aan de toegenomen (en nog steeds toenemende) performantie, programmeerbaarheid en precisie van deze hardware.

Het computer graphics domein is ook zeker niet onveranderd gebleven. Er bestaan zulke volledige globale illuminatie modellen dat dit domein zelfs niet meer voor verbetering vatbaar lijkt te zijn. Merk op dat verbetering hier geïnterpreteerd moet worden als grafisch van betere kwaliteit zijnde. De tijd nodig om een fotorealistisch beeld te genereren ligt namelijk nog steeds aan de hoge kant. Waar er realtime interactiviteit vereist is in 3D applicaties, is er nog steeds een opzettelijke verlaging van de kwaliteit vereist om de rendertijd te beperken.

In deze thesis wordt er getracht door middel van de grafische hardware een implementatie te maken die interactieve rendering toelaat door middel van klassieke ray tracing.

Per hoofdstuk zal er nu een kort overzicht gegeven worden van de materie die er in behandeld wordt:

Hoofdstuk 1: Inleiding Dit hoofdstuk.

Hoofdstuk 2: Achtergrond In dit hoofdstuk zullen enkele belangrijke, zaken waarop dit werk steunt, besproken worden. Eerst zal de rendering vergelijking uit de doeken worden gedaan. Met dit als achtergrond wordt vervolgens de ray tracing techniek uitgelegd.

Hoofdstuk 3: Grafische hardware Dit hoofdstuk geeft een inleiding tot grafische hardware. Eerst wordt er geschetst hoe deze hardware de laatste 10 jaar geëvolueerd is. Vervolgens zal de huidige rendering pipeline nader worden toegelicht. Uiteindelijk worden de GPU en CPU naast elkaar gezet, en worden de belangrijkste voor- en nadelen besproken.

Hoofdstuk 4: GPU programmeren Dit hoofdstuk handelt over hoe de GPU geprogrammeerd kan worden. Vooraleer we daar toe komen, zullen we de GPU abstraheren tot het stream programming model. Vervolgens zullen

we dit mappen op een concrete taal, namelijk Cg[1]. Er zal aangehaald worden hoe typische shaders, alsook general purpose shaders, eruitzien in Cg. Uiteindelijk worden de moeilijkheden van het huidige GPU programmeren in kaart gebracht.

Hoofdstuk 5: Implementatie In dit hoofdstuk zal de GPU en CPU implementatie besproken worden. Eerst worden de gekozen gegevensstructuren behandeld, daarna de algoritmen die erop opereren.

Hoofdstuk 6: Resultaten Dit hoofdstuk bevat een beschrijving van de bekomen resultaten van de GPU en CPU implementatie, met mogelijke verklaringen hiervoor. Aangezien veel van de bekomen resultaten systeemspecifiek zijn, zullen eerst de systeemparamaters gegeven worden. Een belangrijk deel van de verklaringen heeft betrekking tot de coherentie tussen de geschoten stralen doorheen de acceleratiestructuur. Een onderzoek van de padencoherentie zal bijgevolg een belangrijk aandeel vormen van dit hoofdstuk. Verder zal er nog een model geconstrueerd worden voor het evalueren van de kost van verschillende if-structuren. Dit zal gebruikt worden voor een vergelijkende studie tussen algoritmen uit de literatuur, en het algoritme dat voor deze thesis geïmplementeerd werd.

Hoofdstuk 7: Toekomstig werk Zaken die niet binnen het bestek van deze thesis pasten, maar toch mooi zouden aansluiten, zullen hier aangehaald worden.

Hoofdstuk 8: Conclusie Conclusies die getrokken kunnen worden uit dit werk, worden in dit hoofdstuk besproken.

[1] C for graphics [nVi]

Hoofdstuk 2

Achtergrond

In dit hoofdstuk zullen enkele belangrijke, zaken waarop dit werk steunt, besproken worden. Eerst zal de rendering vergelijking uit de doeken worden gedaan. Met dit als achtergrond wordt vervolgens de ray tracing techniek uitgelegd.

2.1 Rendering vergelijking

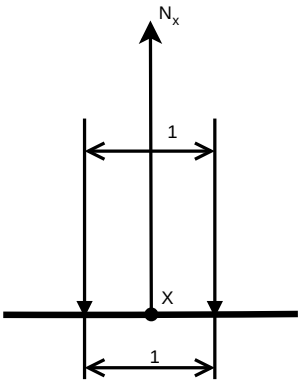
Vooraleer we ons kunnen toespitsen op ray tracing, is het belangrijk om eerst een notie te hebben van het gedrag van licht in de reële wereld. Lichtbronnen zenden energie uit onder de vorm van fotonen. Deze fotonen bewegen zich voort in de ruimte en kaatsen af (of door) fysische objecten totdat ze geabsorbeerd zijn door de omgeving. Merk op dat zo een fysisch object een camera of een oog kan zijn, dat een deel van de fotonen van een omgeving absorbeert, en deze energie omzet in een beeld.

Een wiskundige vergelijking die dit beschrijft, is de bekende rendering vergelijking. Deze vergelijking onderstelt dat lichtenergie zich ogenblikkelijk propageert (dus bv. geen fluorescentie), en dat er geen participerende media aanwezig zijn. Ze ziet er als volgt uit:

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Theta_x} f_r(x, \Psi \to \Theta) L(x \leftarrow \Psi) cos(N_x, \Psi) d\omega_\Psi$$

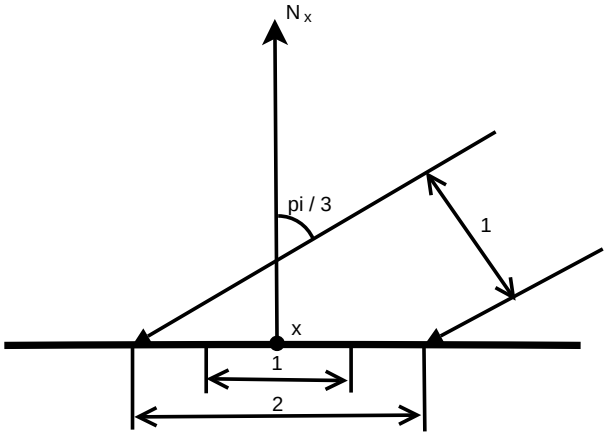
$L(x \to \Theta)$ stelt de hoeveelheid uitgaande radiantie (energie per oppervlakte) voor op positie x in de richting Θ. Analoog is $L(x \leftarrow \Psi)$ de inkomende radiantie uit de richting Ψ. Ook $L_e(x \to \Theta)$ is analoog hieraan, maar betreft enkel de zelf uitgestraalde radiantie. Indien x zich niet op een lichtbron bevindt is bijgevolg $L_e = 0$. $f_r(x, \Psi \to \Theta)$ is de BRDF[1]. Dit geeft aan "hoeveel" van het inkomende licht in een punt x uit richting Ψ er gekaatst wordt in de richting Θ. De factor $cos(N_x, \Psi)$, waarbij N_x de normaal op x voorstelt, zorgt ervoor dat het uitsmeren van het licht over het oppervlak rond x in rekening wordt gebracht. Hoe schuiner het licht binnenvalt, hoe meer de hoek tussen N_x en Ψ nadert naar $\pi/2$ rad, dus hoe meer de $cos(N_x, \Psi)$ nadert naar 0. Dit strookt met de intuïtie, aangezien de radiantie dan inderdaad verdeeld dient te worden over een oneindige oppervlakte. Dit zal verduidelijkt worden aan de hand van situatieschetsen 2.1 en 2.2.

[1] bidirectional reflectance distribution function



Figuur 2.1: *Verdeling van invallende licht, onder een hoek van 0 radiaal t.o.v. de normaal N_x, zodat $cos(N_x, \Psi) = cos(0) = 1$.*

In figuur 2.1 komen de invallende lichtstralen evenwijdig aan de normaal N_x op het oppervlak. De cosinus tussen beide vectoren is duidelijk gelijk aan 1. In figuur 2.2 komen de invallende lichtstralen onder een hoek van $\pi/3$ rad met de normaal N_x op het oppervlak, dus de cosinus ervan is gelijk aan $1/2$. De energie die binnenkomt wordt als het ware uitgesmeerd over een twee maal zo groot oppervlak.



Figuur 2.2: *Verdeling van invallende licht, onder een hoek van $\pi/3$ radiaal t.o.v. de normaal N_x, zodat $cos(N_x, \Psi) = cos(\pi/3) = 1/2$.*

Nu elk afzonderlijk deel van de vergelijking toegelicht is, gaan we deze als een geheel trachten te interpreteren. De hoeveelheid uitgaande radiantie in een richting Θ vanuit een punt x, is de som van de radiantie die het materiaal zelf uitzendt in die richting ($L_e(x \rightarrow \Theta)$), en het weerkaatste deel van de inkomende radiantie volgens de richting Θ. Laat ons het gereflecteerde deel noteren als Θ

$L_r(x \rightarrow \Theta)$. Dit geeft volgende eenvoudigere vorm:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta)$$

waarbij

$$L_r(x \rightarrow \Theta) = \int_{\Theta_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) cos(N_x, \Psi) d\omega_\Psi$$

Het gereflecteerde deel $L_r(x \rightarrow \Theta)$ kan gezien worden als de "som" van de inkomende radiantie uit alle richtingen Ψ, uitgesmeerd over het raakoppervlak, en daarvan enkel het weerkaatste deel in de richting Θ.

Voor de mensen die een meer formele en volledigere beschrijving wensen inzake rendering vergelijking ed., worden doorverwezen naar het boek "Advanced Global Illumination" [DBB03].

2.2 Ray tracing

Ray tracing is een techniek om beelden te genereren van virtuele scènes. In deze sectie zal het ray tracing algoritme, de eigenschappen ervan, en mogelijke acceleratiestructuren uit de doeken worden gedaan.

De (GPU) ray tracer implementatie voor deze thesis is vrij samenlopend met de (CPU) versie die voorgesteld wordt in het boek "Realistic Ray Tracing" van Peter Shirley [Shi00]. Hierin wordt stapsgewijs een klassieke ray tracer gebouwd, met telkens een verantwoording waarom er voor bepaalde algoritmen, datastructuren, formules, e.a. geopteerd werd. Dikwijls werden bepaalde keuzen gemaakt, puur voor de eenvoud, ten koste van efficiëntie. Een voorbeeld hiervan is het gebruik van een regular grid in plaats van bijvoorbeeld k-d tree. Dit was in vele gevallen welgekomen om de GPU-code beperkt te houden qua complexheid en aantal instructies.

In de doctoraatsthesis "Realtime Ray Tracing and Interactive Global Illumination" van Ingo Wald [Wal04] komen allerlei technieken en afwegingen naar voren om realtime ray tracing mogelijk te maken. Bepaalde optimalisaties, zoals gebruik van SSE[2], geheugenindeling en cache alignment zijn niet zonder meer over te nemen voor een programma dat uitgevoerd wordt op de GPU.

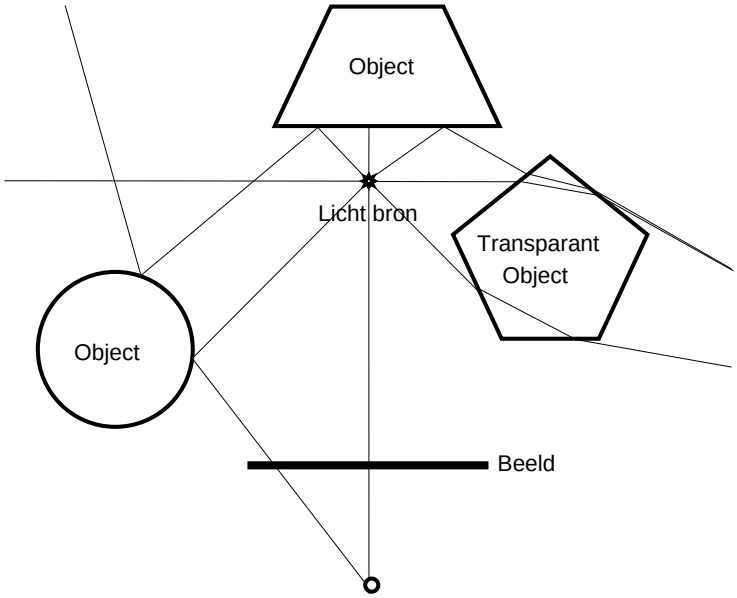
2.2.1 Algoritme

Als we een waarheidsgetrouw beeld wensen te bepalen van een gegeven virtuele scène, zouden we als volgt te werk kunnen gaan. Eerst en vooral plaatsen we onze fictieve camera in de wereld. We itereren over alle lichtbronnen en per lichtbron schieten we lichtdeeltjes (fotonen) de wereld in. We volgen deze doorheen de scène volgens de wetten van de fysica, totdat allen geabsorbeerd zijn of zich niet meer in de beschouwde scène bevinden. Diegene die op de onze camera zijn terechtgekomen en door het brandpunt gaan (zie figuur 2.3), geven we weer op de corresponderende plaats op ons beeldscherm.

[2]Streaming SIMD Extensions (Intel). Door hiervan gebruik te maken is het mogelijk 4 floats te verwerken in slechts één klok cyclus. Een ray tracer kan hiervoor herschreven worden zodat 4 stralen tegelijkertijd worden verwerkt. [WBWS01]

Duidelijk vergt dit een enorm groot aantal geschoten fotonen alvorens er sprake is van een bruikbaar beeld. Gelukkig heeft licht de reciprociteitseigenschap dat het mogelijk maakt om de fotonen in omgekeerde volgorde te traceren. Ray tracing maakt hier uitbundig gebruik van. Voor elke pixel wordt de straal bepaald die door pixel en oogpunt gaat (oogstraal). Deze straal wordt dan gevolgd vanuit het oog, door de betreffende pixel, totdat deze een object uit de scène intersecteert. De kleur van de pixel wordt nu bepaald aan de hand van de materiaaleigenschappen van het geïntersecteerde object. Indien het object reflecterend is, zal er recursief verder een weerkaatste straal geschoten worden om zo verder de kleur te bepalen. Analoog geldt voor refractieve objecten dat er een straal wordt gebroken in plaats van weerkaatst. Meer hierover verder in deze sectie. Pseudocode 2.1 beschrijft het klassieke ray tracing algoritme in meer detail.

Als we figuur 2.3 met figuur 2.4 vergelijken zien we duidelijk dat er sneller een beeld gevormd wordt door te ray tracen in plaats van het naïeve algoritme. Dit is te wijten aan het feit dat er bij het naïeve algoritme enorm veel stralen worden gevolgd die niet bijdragen tot het uiteindelijke beeld. Bij ray tracing is dit niet het geval.



Figuur 2.3: *Naïef algoritme om beelden te vormen. Vanuit elke lichtbron worden,volgens de wetten van de fysica, fotonen geschoten en gevolgd tot deze door de camera geabsorbeerd worden.*

Primaire stralen

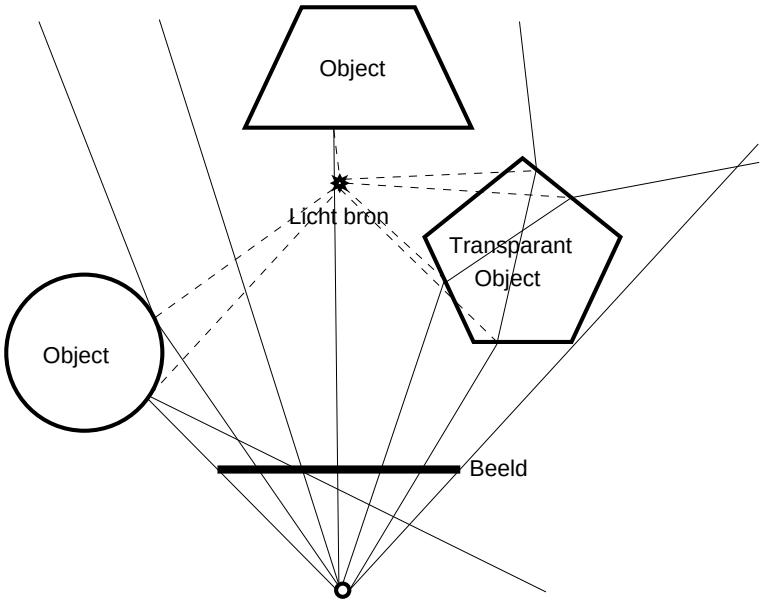
We beschouwen een doelbeeld van n_x op n_y pixels. De camera (pinhole) bevindt zich op positie e, gericht volgens g, loodrecht op het view-plane op een afstand s. v_{up} is de view-up vector die naar de "bovenkant" van de scène wijst. Laten we de grootte van het view-plane noteren als s_x en s_y. Figuur 2.5 geeft hier een

```
Voor elke pixel in beeld {
  straal = pixel - oogpunt
  tDichtsteObject = oneindig
  dichtsteObject = achtergond

  Voor elk object in scene {
     Als straal object intersecteert {
        Als t kleiner is als tDichtsteObject {
           tDichtsteObject = t
           dichtsteObject = object
        }
     }
  }

  Als dichtsteObject is achtergond {
     pixel = achtergrondKleur
  } anders {
     Schiet straal naar elke puntlichtbron
         en controleer of oorspong in schaduw
     Indien reflecterend, genereer reflectiestraal en
         herhaal deze stappen voor bekomen straal
     Indien transparant, genereer refractiestraal
         herhaal deze stappen voor bekomen straal
     Bepaal kleur adhv de shading functie
  }
}
```

Codefragment 2.1: *Pseudocode ray tracing*



Figuur 2.4: *Ray tracing algoritme.*

grafische representatie van.

Eerst bepalen we een assenstelsel (u, v, w) voor de camera.

$$w = \frac{g}{\|g\|}$$

$$u = -\frac{v_{up} \times w}{\|v_{up} \times w\|}$$

$$v = w \times u$$

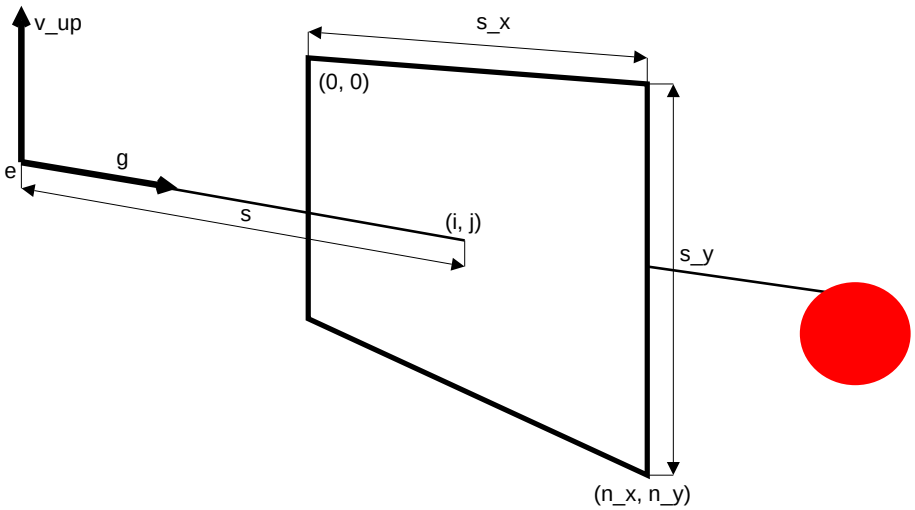
De primaire stralen, ook wel oogstralen genoemd, kunnen nu als volgt bepaald worden. Per pixel i, j is de bijbehorende straal gelijk aan

$$p(t) = e + t\left(s_x\left(\frac{i}{n_x - 1} - \frac{1}{2}\right)u + s_y\left(\frac{j}{n_y - 1} - \frac{1}{2}\right)v + sw\right)$$

Deze formule is een aangepaste versie van de formule gegeven in [Shi00].

Objectintersectie

Om te kunnen bepalen welk object geraakt wordt door de betreffende lichtstraal, moet er voor elke soort gebruikte primitieve een een intersectie methode aanwezig zijn. Indien er gebruik gemaakt wordt van een acceleratiestructuur, dienen er voor het doorlopen ervan ook de juiste intersectie methoden aanwezig te zijn. Veelgebruikte primitieven zijn driehoeken, bollen, vlakken en axis aligned boxes. Een intersectiealgoritme geeft typisch aan of er daadwerkelijk sprake was van een intersectie, de positie van het intersectiepunt op de straal, en de normaal op het inslagpunt.



Figuur 2.5: *Parameters gebruikt voor het genereren van de primaire stralen.*

Een concreet voorbeeld dat gegeven wordt, is de intersectie van een straal met een bol. Dit is een typisch voorbeeld vanwege de eenvoudige wiskundige oplossing en de mooie resultaten. Beschouw een bol met centrum c en straal R, dan geldt:

$$(p - c) \cdot (p - c) - R^2 = 0$$

voor alle punten p die zich op de bol bevinden. Een straal met oorsprong o en richting d kunnen we beschrijven als een rechte, met volgende vergelijking:

$$p(t) = o + td$$

We zoeken nu de kleinst mogelijk t zodanig dat $(p(t) - c) \cdot (p(t) - c) - R^2 = 0$. Met een minimum aan rekenwerk komen we aan:

$$t = \frac{-d \cdot (o - c) \pm \sqrt{(d \cdot (o - c))^2 - (d \cdot d)\left((o - c) \cdot (o - c) - R^2\right)}}{d \cdot d}$$

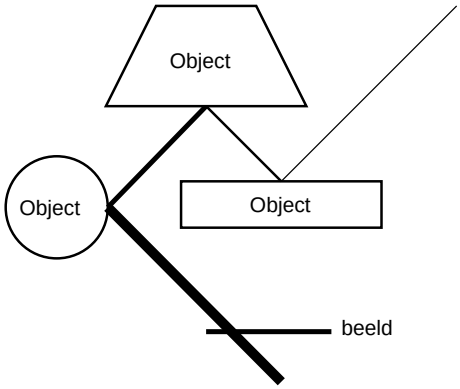
Indien de discriminant (onder de vierkanstwortel) negatief is, is er geen intersectie met dit object. Anders wordt de kleinste positieve t genomen aangezien dit de eerste (dus dichtstbijzijnde) intersectie is met de bol. Indien deze waarde niet bestaat, is er geen sprake van een intersectie. De normaal n is ook triviaal te bepalen, namelijk $n = (p - c)/R$.

Het intersecteren van vlakken of axis aligned boxes is min of meer van dezelfde complexiteit. Zie voor meer info [Shi00]. Driehoeken daarentegen zijn al iets geavanceerder om op effeciënte wijze af te handelen. Een veel gebruikt algoritme hiervoor is Moller-Trumbore [MT97], wat in detail behandeld wordt in hoofdstuk 5.2. Verder zie [Wal04].

Reflectie

Bij het ray tracen van een reflecterend oppervlak is de kleur niet enkel afhankelijk van het materiaal van het geïntersecteerde object, maar ook van objecten

geraakt door de afgekaatste stralen. In de situatieschets 2.6 is de bijdrage van de straal aangegeven door de dikte ervan. Duidelijk is de primaire straal de dikste. Aangezien elk object hier 50 % reflecterend is, halveert de dikte van straal na elke bosting met een object, alsook de bijdrage tot de kleur.

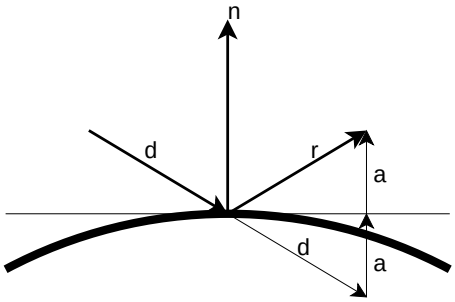


Figuur 2.6: *Reflecteren van stralen. De dikte van de straal geeft aan voor hoeveel procent deze straal de kleur beïnvloedt, gaande van 100% voor de dikste straal, tot 12,5% voor de dunste.*

De uitgaande reflectiestraal r is afhankelijk van de normaal n en invallende straalrichting d als volgt:

$$r = d + 2\frac{d \cdot n}{\|n\|^2}n$$

Hoe deze formule tot stand komt is duidelijk af te lezen van de bijbehorende schets 2.7. Merk op dat $\frac{d \cdot n}{\|n\|^2}n$ voorgesteld wordt als a.



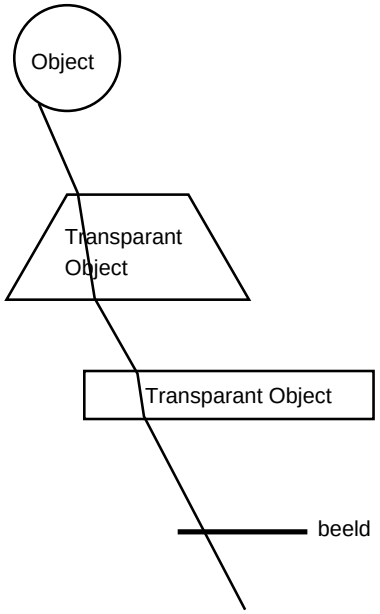
Figuur 2.7: *Parameters voor de reflectieformule, waarbij r de uitgaande reflectiestraal is, n de normaal en $a = \frac{d \cdot n}{\|n\|^2}n$*

Refractie

Wanneer licht botst op bijvoorbeeld glas, water, diamant, ... zal het gerefracteerd worden. Licht dat zich van een optisch ijlere, naar een optisch dichtere

stof of omgekeerd begeeft, zal afgebogen worden en zich verder door de materie begeven volgens de wet van Snell. Dit fenomeen heet refractie. Aangezien in de GPU ray tracer implementatie geen refractie aanwezig is, zal er hier niet in detail op ingegaan worden. Zie alweer [Shi00] voor formules en meer details.

In figuur 2.8 wordt één lichtstraal gevolgd, die twee maal gerefracteerd wordt en vervolgens botst op een niet transparant object (bol).



Figuur 2.8: *Gerefracteerde stralen.*

Schaduw

Het bepalen of een intersectiepunt zich in een schaduw gebied van een bepaalde lichtbron bevindt, is zeer recht-toe-recht-aan. Creëer een straal met oorsprong het intersectiepunt en eindpunt het beschouwde puntlichtbron. Controleer vervolgens of lijnstuk één of meerdere objecten in de scène kruist. Indien ja, bevindt dit punt zich in een schaduw gebied, anders in een belicht gebied. Merk op dat deze stappen voor elke lichtbron dienen ondernomen te worden.

Spatiale subdivisie

Het ray tracing algoritme dat we tot nog toe hebben voorgesteld, is weinig effeciënt voor grote scènes. Met grote scène wordt bedoeld: bestaande uit een groot aantal primitieven. Voor het bepalen van een straal-scène intersectie dienen *alle* primitieven getest te worden, om vervolgens te bepalen welke het dichtstbijzijnde intersectiepunt oplevert. Dikwijls maakt men gebruik van driehoeken als primitieven. Ook gekromde oppervlakken dienen dan benaderd te worden door voldoende (meestal veel) driehoeken, om er smooth uit te zien. Het renderen van een gemiddeld 3D model dat iets of wat realistisch oogt, zou al snel leiden

tot gigantische rendertijden. Dit probleem is eenvoudig op te lossen door onze ruimte op te delen in deelruimten (spatiale subdivisie). Per deelruimte houden we dan een lijst bij van welke primitieven zich er al dan niet volledig in bevinden. Dit maakt dat we enkel die primitieven moeten controleren, die zich in de geïntersecteerde deelruimten bevinden. Indien een gepaste acceleratiestructuur zorgvuldig geconstrueerd wordt, en op een slimme manier doorlopen, kan dit een enorme snelheidswinst opleveren. Een kd-tree en BSP-tree kunnen bijvoorbeeld doorlopen worden in $O(\log n)$ t.o.v. $O(n)$ voor een regular grid, waarbij n het aantal primitieven is.

Hieronder volgt een lijst van veelgebruikte spatiale subdivisie structuren, met telkens een korte omschrijving hoe deze de ruimte opdelen. In figuur 2.9 wordt elk van deze structuren, toegepast op een voorbeeldscène, geïllustreerd aan de hand van een schematische voorstelling.

Regular grid (Uniform grid) Dit is een axis aligned bounding box waarbij elke as verdeeld wordt in een aantal gelijke delen. Hierdoor wordt de ruimte opgesplitst in allemaal gelijke voxels, die axis aligned zijn.

Bounding volume hierarchy Rond groepen van primitieven worden bounding volumes gekozen, en deze volumes kunnen dan weer ondergebracht worden in grotere volumes enz., zodat er sprake is van een hiërarchie van bounding volumes.

Octree Dit is een axis aligned bounding box dat in acht gelijke deelruimten verdeeld is door een x-, y- en z-vlak gaande door het middelpunt van deze box. Dit kan per verkregen ruimte recursief verder gebeuren, tot er aan een gegeven stopvoorwaarde voldaan is.

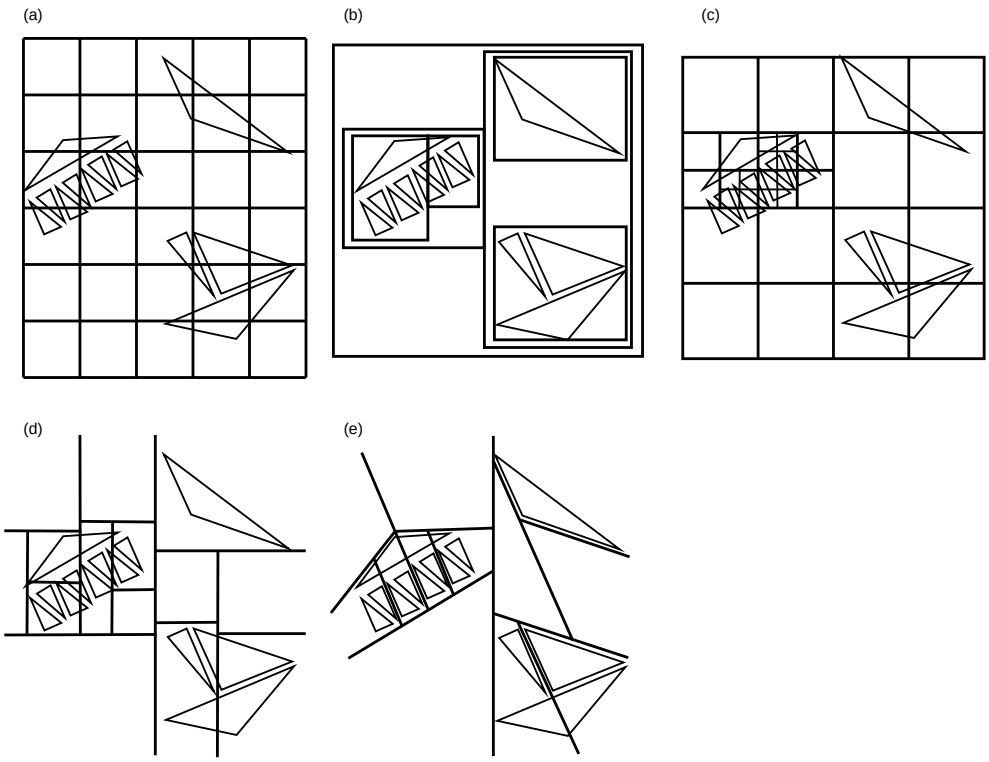
Kd-tree Dit is een boomstructuur waarbij de ruimte telkens in twee delen gedeeld wordt, afwisselend door een x-, y- of z-vlak. Dit gebeurt recursief verder tot er aan een gegeven stopvoorwaarde voldaan is.

BSP-tree Dit is een veralgemeende kd-tree aangezien elke ruimte opgedeeld kan worden door een willekeurig vlak, in plaats van enkel door een x-, y- of z-vlak.

Voor een gedetailleerde uitwerking en vergelijkende studie van spatiale datastructuren, zie [Hav00]. Voor een zeer geoptimaliseerde kd-tree implementatie, zie [Wal04]

2.2.2 Eigenschappen

Een pittig detail aan het klassieke ray tracing algoritme is dat het *niet* strookt met de rendering vergelijking. Om de kleur te bepalen op een bepaald intersectiepunt x dient er rekening gehouden te worden met het invallende licht vanuit de omgeving van x, en niet enkel met het licht vanuit de ideale reflectie- (of refractie-) richting. Een klassieke ray tracer is wel eenvoudig aan te passen naar een zogenaamde "stochastic ray tracer", die wel degelijk voldoet aan de rendering vergelijking [Shi00, DBB03]. Eigenlijk is dit een redelijk rechtstreekse implementatie van de rendering vergelijking waarbij er door middel van Monte Carlo methoden het aantal samples per intersectiepunt aanzienlijk verlaagd kan worden. Voor meer details, zie de gegeven referenties.



Figuur 2.9: *Spatiale subdivisie structuren. (a) Regular grid (b) Bounding volume hierarchy (c) Octree (d) Kd tree (e) Bsp tree*

Effecten zoals "soft shadows", "caustics" en "color bleeding" zijn met een klassieke ray tracer onmogelijk te verkrijgen. Omdat een stochastic ray tracer veel samples per pixel (en dus rekentijd) nodig heeft om te convergeren naar een correct beeld, wordt er voor het verkrijgen van caustics en color bleeding meestal gebruik gemaakt van photon mapping. Hierbij worden fotonen geschoten vanuit de lichtbronnen, zoals bij het naïeve algoritme, maar worden opgeslagen in een zogenaamde photon map. De combinatie van een stochastic ray tracer en een photon mapper (voor de indirecte belichting) is dus efficiënter om een fotorealistisch beeld te genereren dan slechts één van beide technieken te verkiezen. Meer details hierover zijn te vinden in [Jen01, DBB03].

Hoofdstuk 3

Grafische hardware

Dit hoofdstuk geeft een inleiding tot grafische hardware. Eerst wordt er geschetst hoe deze hardware de laatste 10 jaar geëvolueerd is. Vervolgens zal de huidige rendering pipeline nader worden toegelicht. Uiteindelijk worden de GPU en CPU naast elkaar gezet, en worden de belangrijkste voor- en nadelen besproken. De informatie waarop dit hoofdstuk gebaseerd is, is afkomstig van [nVi, ATI].

3.1 Geschiedenis

Ruim 10 jaar geleden was de grafische hardware, onder de vorm van beeldkaarten, enkel voorzien van 2D rasterisatie functionaliteit. Ontwerpers van 3D applicaties (CAD, games, ...) dienden zelf over software te beschikken die in staat was om een virtuele scène te kunnen renderen, zonder hardware ondersteuning. Vanzelfsprekend was het programmeren van interactieve 3D applicaties een enorm complexe zaak vanwege de beperkte resources. Low-level programmeren met de nodige voorzichtigheid, al dan niet gedeeltelijk in machinetaal, was dus de boodschap voor het renderen van de meest eenvoudige 3D wereld. Gelukkig is hier de laatste 10 jaar verandering in gebracht.

In 1995 kwam 3dfx met de Voodoo kaart op de markt, die het mogelijk maakte getextureerde 3D driehoeken correct op het scherm te tonen. Door middel van hardwarematige Z-buffering werden deze driehoeken in de juiste volgorde op het scherm getekend. Deze kaart werkte via de PCI[1] interface.

In 1998 kwamen de TNT (van nVidia) en Rage (van ATI) kaarten met ingebouwde multitexturing op de markt. Deze kaarten communiceerden via de AGP[2] poort.

Vanaf 1999 verschenen er kaarten die een zogenaamde TnL unit bevatten (GeForce 256, GeForce 2, Radeon 7500, Savage3D). De naam TnL staat voor "Transform and Lighting". Standaard belichtingen en transformaties (projecties, rotaties, ...) konden vanaf toen overgenomen worden door de beeldkaart. Ook bump mapping, cube texture mapping en projective texture mapping werd ingebakken in de hardware.

[1] Peripheral Component Interconnect
[2] Accelerated Graphics Port

In het jaar 2001 kende de grafische hardware een belangrijke vooruitgang, namelijk kaarten met programmeerbare vertex shaders met static flow control[3] doken op (GeForce 3 en 4, Radeon 8500). Ook volume texture mapping, shadow mapping en antialiasing werd allemaal mogelijk door de hardware. Er is natuurlijk niet enkel *extra* functionaliteit toegevoegd op de grafische kaart, maar ook optimalisaties (zoals Z-culling) zijn gerealiseerd.

Een jaar later, in 2002, werden de beeldkaarten nog meer programmeerbaar. Eerst en vooral kwamen er programmeerbare fragment shaders bij (met static flow control). Ook niet onbelangrijk is het feit dat vanaf toen de vertex shaders static *en* dynamic flow control ondersteunden! MRT[4] deed ook zijn intrede. De toenmalige laatste nieuwe 3D-kaarten waren de GeForce FX, Radeon 9600, 9800, X600 tot X800.

Dynamic branching op de fragment shader werd pas in 2004 een feit. Ondertussen had ook PCIe[5] zijn intrede gedaan, is de GeForce 6 reeks verschenen, is er 64 bit kleuren ondersteuning, ... Kaarten van de GeForce 6 reeks, bezitten tot 16 pixel pipelines en 6 vertex engines, die parallel draaien op een kloksnelheid van 450 Mhz.

Het is zonder meer duidelijk dat de ontwikkeling van grafische (3D) kaarten een enorme evolutie heeft gekend in de laatste 10 jaar. De hardware is ondertussen zo algemeen programmeerbaar geworden dat het zelfs interessant wordt om deze hardware te "mis"bruiken voor totaal andere doeleinden, zoals het oplossen van stelsels vergelijkingen, videocompressie, ray tracing, e.a.

3.2 Graphics pipeline

De graphics pipeline is een pipeline[6] waar langs de ene kant door de applicatie een scène beschrijving wordt ingestopt, en langs de andere kant een gerenderd beeld uit komt. Zoals met een reële pijpleiding, zal de doorvoer volledig afhangen van de traagste stage (dunste deel van de buis). Er bestaat een waaier aan verschillende schematische voorstellingen van de graphics pipeline (veel / weinig detail, wel / niet platformspecifiek, ...). Hier is er geopteerd voor een eenvoudig model, met als doel de vertex en fragment shaders hierin te kunnen plaatsen.

Beschouw figuur 3.1. Opvallend aan deze voorstelling is dat ze iets te high-level is om praktisch nuttig te zijn voor het beoogde doel. Daarom zal stage per stage meer in detail getreden worden door telkens een opsomming te geven van de zaken die plaatsvinden in de desbetreffende stage. De laatste stage verzorgt enkel de uiteindelijke uitvoer, dus wordt hieronder niet verder uitgewerkt.

3.2.1 Applicatie

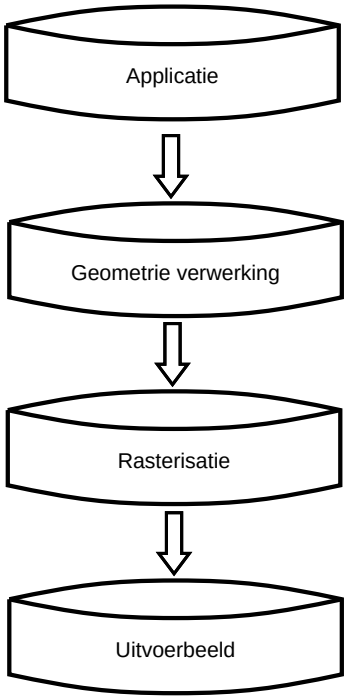
De applicatie stuurt een beschrijving van de scène (geometrie, materialen, positie, lichtbronnen, ...) door naar de grafische kaart. Een eerste vorm van

[3] verder in de tekst zal duidelijk worden wat hiermee bedoeld wordt

[4] Multiple Render Target: dit maakt het mogelijk om meerdere waarden terug te geven na het uitvoeren van een fragment shader

[5] Peripheral Component Interconnect Express

[6] reeks aan elkaar gesloten gegevensverwerkende elementen (dikwijls in parallel uitvoerbaar), waarbij de output van een element wordt doorgegeven als input voor het volgende element



Figuur 3.1: *Graphics pipeline.*

softwarematige geometrie verwerking kan hier reeds plaatsvinden. Denk hierbij aan frustum culling van complexe modellen, LOD[7] selectie, enkel zichtbare sectoren van de wereld renderen, ...

3.2.2 Geometrie verwerking

De geometrie verwerking is gedeeltelijk programmeerbaar door middel van vertex shaders. Laat ons "gedeeltelijk" nader specifiëren. Enkel de transformatie van coördinaten, normalen en textuur coördinaten, en de standaard belichting, kan geherprogrammeerd worden.

Hier volgt een overzicht van de toegewezen taken:

- Transformatie van coördinaten en normalen (model → wereld, wereld → oog)
- Vertex belichting
- Transformatie van texture coördinaten
- Backface culling
- Transformatie naar clip space
- Assembleren van vertices in primitieven
- Clippen over viewing frustum

[7] Level Of Detail

3.2.3 Rasterisatie

In de rasterisatiefase vindt de scanconversie plaats, waarna per pixel een kleurwaarde bepaald wordt. Het bepalen van deze kleurwaarde gebeurt door middel van de fragment shader, die tegenwoordig herprogrammeerbaar is. De fragment shader krijgt de waarden zoals geïnterpoleerde texture coördinaten en kleuren binnen, en de programmeur gebruikt deze dan om een kleur te bepalen op de gewenste manier. Voorheen, zijnde voor 2002, was de fragment shader ingebakken in hardware waardoor deze niet aangepast kon worden.

3.3 GPU versus CPU

In deze sectie zullen de voor- en nadelen van de GPU ten opzichte van de CPU overlopen worden. Hierdoor zal duidelijk worden waarom er interesse is in het gebruik van de grafische hardware voor algemene doeleinden, hoewel deze hardware daar oorsponkelijk niet voor ontworpen was.

Het parallellisme is de belangrijkste aanzet geweest tot het ontstaan van GPGPU[8] programmeren. Op de grafische kaart bevinden zich namelijk meerdere processoren die in parallel instructies kunnen uitvoeren. Aangezien in grafische toepassingen relatief veel 3- en 4-tupel operaties aanwezig zijn, is de tegenwoordige grafische kaart hiervoor geoptimaliseerd. Denk hierbij aan vertices (x, y, z, w), kleuren (R, G, B, A), texture coördinaten (S, T, R, Q), ... We kunnen dus spreken van een vector processor, die de standaard 4-tupel operaties in slechts één clock cycle kan voorzien. De typische "grafische" operaties zoals dot product, cross product, matrix product, ... kunnen dus aan een lage kost uitgevoerd worden.

De huidige generatie grafische kaarten (bv. GeForce 6800 GT) zijn voorzien van 16 pixel pipelines en 6 vertex pipelines. De hedendaagse vertex processoren zijn reeds MIMD[9], wat wil zeggen dat deze volledig in parallel, verschillende instructies kunnen uitvoeren. Spijtig genoeg zijn de fragment shaders nog SIMD[10]. Het lijkt voor de programmeur dat hij software schrijft voor een MIMD processor, hoewel dit geëmuleerd wordt op een SIMD processor. Het is dus van belang dat de fragment shaders die in parallel uitgevoerd worden, een coherente program flow bezitten. Dit is echter geen verplichting, maar zal dikwijls leiden tot een kortere uitvoeringstijd. Merk op dat dit bij vertex shaders niet of nauwelijks (door instructiecaching?) snelheidswinst zal opleveren, aangezien er meerdere *aparte* processoren aanwezig zijn.

De beschikbare bandbreedte voor on-board geheugentoegang is ook een stuk hoger (grote orde $\times 10$) dan die tussen CPU en systeemgeheugen. Maar toch is het zeker niet allemaal rozengeur en maneschijn. De bandbreedte tussen GPU en CPU, is relatief zeer laag. Het terughalen van data van de grafische kaart is makkelijk 100 $\times$ trager dan de GPU $\leftrightarrow$ on-board geheugen communicatie, via een AGP poort. Het uploaden van data is ongeveer 10 keer trager (ipv 100). PCIe heeft hierin gelukkig verandering gebracht. Upload en download snelheden zijn gelijk, en verhoogd tot (theoretisch) ± 8 GB/s, t.o.v. ± 2 GB/s upload en ± 200 MB/s download bij AGP.

[8] General Purpose GPU programmeren
[9] Multiple Instruction Multiple Data
[10] Single Instruction Multiple Data

De tegenwoordige CPU's zijn nochtans ook gepipelined, hebben streaming instructies (SSE[11]), ondersteunen hyperthreading[12], ... Toch verkiest men meer en meer de GPU voor parallelliseerbare algoritmen. De reden hiervoor is dat deze hardware veel "meer parallel" is dan de huidige verkrijgbare CPU's.

[11] Streaming SIMD Extensions (Intel)
[12] simultane multithreading (Intel)

Hoofdstuk 4

GPU programmeren

Dit hoofdstuk handelt over hoe de GPU geprogrammeerd kan worden. Vooraleer we daar toe komen, zullen we de GPU abstraheren tot het stream programming model. Vervolgens zullen we dit mappen op een concrete taal, namelijk Cg[1]. Er zal aangehaald worden hoe typische shaders, alsook general purpose shaders, eruitzien in Cg. Uiteindelijk worden de moeilijkheden van het huidige GPU programmeren in kaart gebracht.

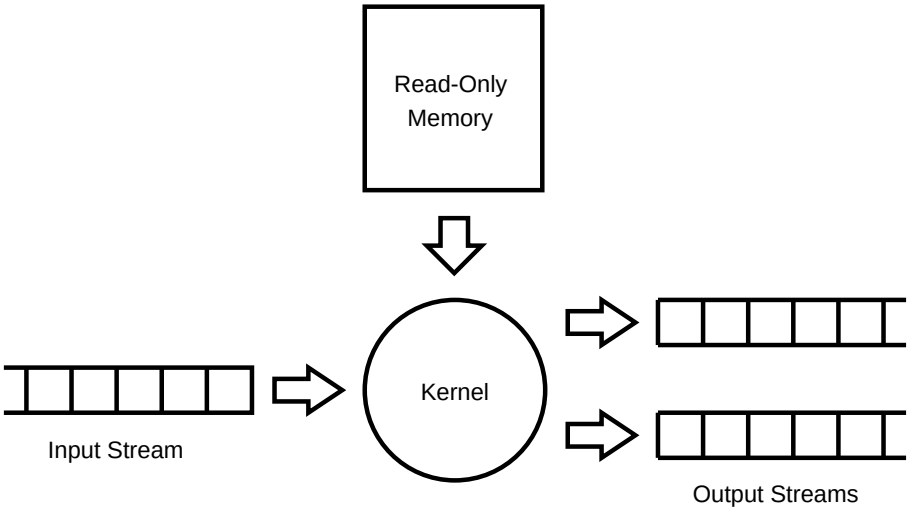
4.1 Stream programming model

Het stream programming model is een model dat programma's oplegt op een bepaalde manier geschreven te zijn. Localiteit in de inputdata en parallellisme dienen hiervoor expliciet aangegeven te zijn in het programma. Dit maakt dat het gecompileerd en geoptimaliseerd kan worden voor stream processoren.

Het stream programming model bestaat enkel uit streams, kernels en een read-only geheugen. Een stream is een verzameling van records, waarop een kernel operaties kan uitvoeren. Een kernel is een proces dat gegeven een record van de input stream, een output stream record genereert. Meerdere output streams zijn eventueel mogelijk (bv. handig voor het sorteren van data). Zie figuur 4.1 voor een schematische voorstelling.

De GPU kunnen we beschouwen als een stream processor, wat natuurlijk de reden is waarom we dit model bespreken. Een vertex shader krijgt als input stream een verzameling vertices (met normalen, kleuren, ...). De shader zelf is dus de voorgenoemde kernel. De output stream bevat opnieuw vertices (en dergelijke), die dan opgebouwd zijn op basis van een input record, en het read-only geheugen. Het read-only geheugen kan bijvoorbeeld bestaan uit 3D vectoren, een array van floats, ... Dit geheugen is dus constant voor alle records. Daarom dat er ook wel gesproken wordt van *globaal* read-only geheugen. Een concrete GPU versie van figuur 4.1 wordt gegeven in figuur 4.2.

[1] C for graphics



Figuur 4.1: *Stream Programming Model schema.*

4.2 Voorbeeldtaal: Cg

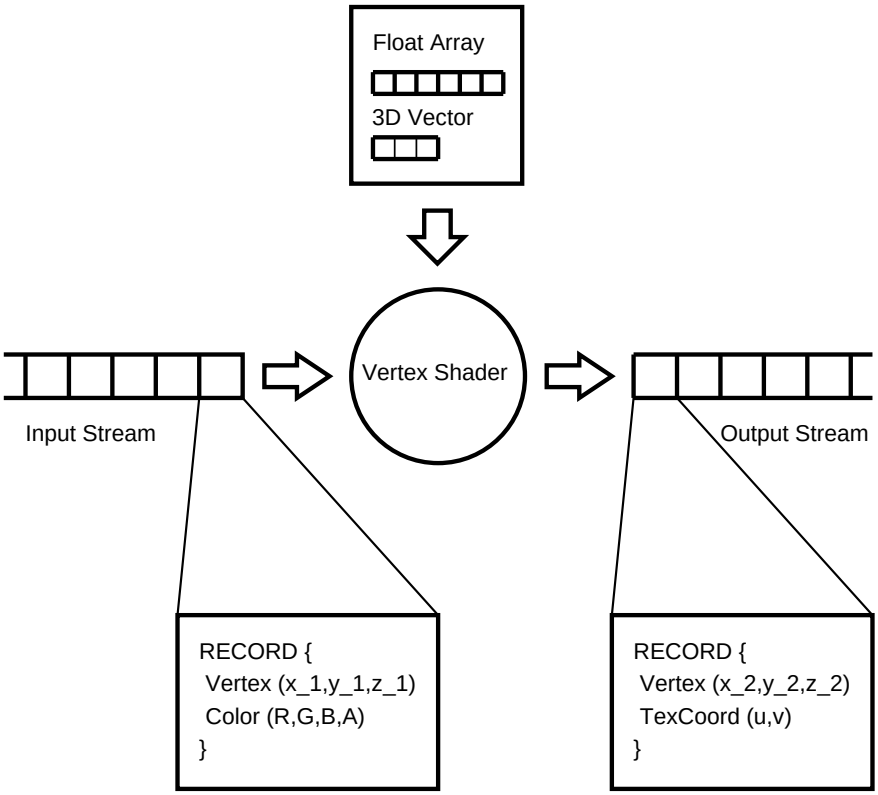
Er bestaan verscheidene programmeertalen die machinetaal code voor de grafische kaart genereren. Voorbeelden hiervan zijn onder andere Cg [nVi], HLSL, GLSL [Ope], BrookGPU [Bro] en Sh [Sh]. De eerste wijdverspreide GPGPU taal, namelijk Cg, zal hier besproken worden. De implementatie voor deze thesis is hierin geprogrammeerd.

In Cg is het mogelijk om GPGPU programma's te schrijven, maar toch dient men zich nog bezig te houden met grafische details. Zo wordt er doorgaans een rechthoek op het scherm getekend om per pixel de fragment shader aan te roepen, ook al hebben de resultaten (output stream) totaal geen grafische betekenis. Ook het expliciet in textures plaatsen van data die doorgeven dienen te worden, is voor vele toepassingen iets dat onder de motorkap zou mogen gebeuren. In BrookGPU, dat nog in ontwikkeling is, worden deze grafische details weg geabstraheerd.

4.2.1 Kernels

Met de kernels in Cg bedoelen we concreet de programmeerbare vertex en fragment shaders. Bij het schrijven van een typisch imperatief of objectgeoriënteerd programma zijn we gewoon dat de hoofdfunctie (bv. `main` in Java, C en C++) éénmaal doorlopen wordt. In Cg werkt dit anders. De hoofdfunctie (willekeurige naam) wordt éénmaal aangeroepen per input stream record, zijnde een vertex of fragment. Typisch kunnen er verscheidene records gelijktijdig verwerkt worden. Merk op dat de volgorde van records in de stream ongedefiniëerd is, en dat er geen enkele vorm van onderlinge communicatie bestaat tussen de in parallel lopende vertex of fragment shaders.

In codefragment 4.1 is voorbeeldcode gegeven die de structuur aangeeft van Cg programma's. De inputstream bestaat hier uit records van het type



Figuur 4.2: *Concreet Stream Programming Model schema voor de GPU.*

`inputRecord`. Deze worden door de kernel, zijnde de `cgMain`, verwerkt zodat een outputstream verkregen wordt. Deze outputstream bestaat uit elementen van het type `outputRecord`. In volgende sectie zal er besproken worden welke data zich in deze records kunnen bevinden.

Merk op dat deze code evengoed voor een vertex als fragment shader kan fungeren. In de volgende paragraaf zal er wel degelijk een opsplitsing gemaakt dienen te worden, aangezien de vertex en fragment shaders op verschillende datastream types werken.

4.2.2 Input en output stream

De vertex en fragment shaders krijgen twee soorten input, enerzijds *variërende*, anderzijds *uniforme*. De variërende is de input die per vertex (resp. fragment) verschillend is, en de uniforme is de input die constant is voor een gehele vertex (resp. fragment) stream. Merk het verband op met het stream programming model. De variërende input komt overeen met de input stream, en de uniforme input is het globaal read-only geheugen dat ter beschikking is.

Logischerwijs is het nodig om afspraken te maken voor de communicatie tussen de applicatie, vertex en fragment shaders. De input van de vertex shader dient op de juiste manier opgevangen te worden van de applicatie, en de output ervan moet ook correct worden doorgestuurd van de vertex naar de

```
struct inputRecord {
    /*
        ...
     */
};

struct outputRecord {
    /*
        ...
     */
};

/*
    Eventuele extra data- en / of functiedefinities
 */

outputRecord cgMain(inputRecord input) {
   outputRecord output;

    /*
        ...
     */

    return output;
}
```

Codefragment 4.1: *Cg kernel code structuur*

fragment shader. Merk op dat er zich tussen de vertex en fragment shader nog de scanconversiefase bevindt, die doorgegeven waarden kan aanpassen, zoals het interpoleren van de texture coördinaten.

Er zijn dus *binding semantics* voorzien die het mogelijk maken om een variabele aan een bepaalde unit te koppelen (bv. kleur, texture coördinaten, e.a.). Deze zijn verschillend voor de vertex en fragment shaders, dus zullen afzonderlijk behandeld worden.

Vertex shader

De volgende input binding semantics voor een vertex shader zullen steeds beschikbaar zijn, hoewel er bij specifieke profielen[2] extra mogelijkheden bestaan (bv. `COLOR`):

`POSITION`	`BLENDWEIGHT`	`TANGENT`	`BINORMAL`
`NORMAL`	`TEXCOORD0-TEXCOORD7`	`PSIZE`	`BLENDINDICES`

In OpenGL [Ope] kunnen variabelen door de applicatie doorgegeven worden door `glVertex`, `glNormal`, `glTexCoord`, ... aan te roepen, die respectievelijk `POSITION`, `NORMAL`, `TEXCOORD0`, ... een waarde toekennen.

De minimum vereiste output binding semantics zijn de volgende:

`POSITION`	`PSIZE`	`FOG`
`COLOR0-COLOR1`	`TEXCOORD0-TEXCOORD7`	

Fragment shader

De per fragment variërende input zijn de al dan niet geïnterpoleerde waarden geproduceerd door de vertex shader. De output binding semantics zijn `COLOR` en soms ook `DEPTH`, afhankelijk van het gebruikte profiel. Sinds het bestaan van MRT[3] is het mogelijk geworden om in één enkele fragment shader, meerdere resultaatwaarden te bepalen.

4.2.3 Read-only geheugen

De constante gegevens voor de hele input stream kunnen meegegeven worden door middel van `uniform` parameters. Deze moeten vooraf in de C/C++ code gekoppeld en ingevuld worden door de gepaste functies aan te roepen. Ook de gebruikte textures zijn read-only geheugen. Het is dus onmogelijk om rechtstreeks veranderingen aan te brengen in een texture. Dit is echter wel mogelijk door de output van de fragment shader om te vormen tot een texture, en deze als input mee te geven aan een volgende fragment shader (CTT[4]/RTT[5]).

[2] Het profiel geeft aan waar de machinetaal, gecompileerd uit de Cg-code, aan moet voldoen. Concreet houdt dit in of de shader data gaat ontvangen vanuit OpenGL of DirectX, en met welk shader model er gewerkt wordt, bijvoorbeeld Shader Model 3.0. De hardware dient dan te voldoen aan een bepaald shader model. Een shader model specifiëert de minumum te ondersteunen shader lengte, of dynamic branching ondersteund wordt, de floating point precisie, e.a. Voor meer info, zie [nVi, sha]

[3] Multiple Render Target

[4] Copy to texture

[5] Render to texture

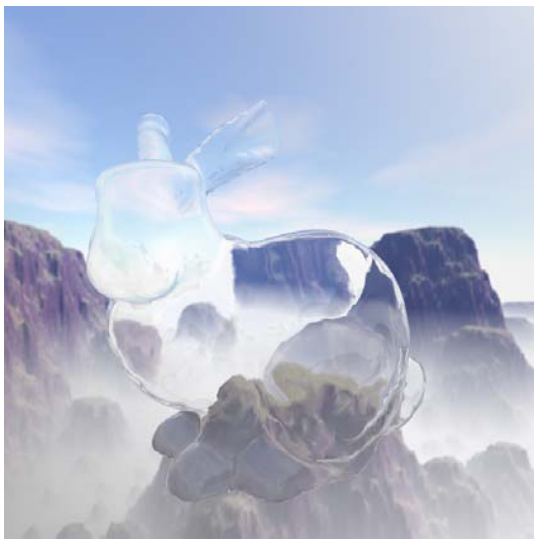
4.2.4 Voorbeeld vertex en fragment shader

Voorbeeldcode van een vertex en fragment shader, geschreven in Cg, zijn te vinden in codefragment 4.2 en 4.3. Het betreft een realtime benadering van per pixel refractie. Deze code is overgenomen van "Cg Toolkit" [nVi] en wordt hier gebruikt als voorbeeld van een uitgewerkte vertex en fragment shader.

Het vermelden van deze code heeft als enige doel een concreet voorbeeld te geven van een shader. Het is dus niet noodzakelijk de code volledig te begrijpen, maar het volstaat om de algemene structuur en de in vorige paragrafen besproken zaken, te herkennen. Zo is het duidelijk dat de vertex kernel, `main`, gegeven data elementen van het type `inputs`, data elementen van het type `outputs` oplevert. Dit gebeurt aan de hand van enkele uniforme variabelen, en een extra geschreven functie, `fast_fresnel`. De mapping van de data, die zich bevinden in `inputs` en `outputs`, naar de gebruikte units (`POSITION`, `TEXCOORD0` e.a) is ook duidelijk zichtbaar in deze code.

De fragment shader ontvangt dan uiteindelijk drie van de vier waarden, uitgevoerd door de vertex shader, zijnde `refractVec`, `reflectVec` en `fresnelTerm`. Merk dus op dat `hPosition` niet rechtstreeks gebruikt wordt door de fragment shader. Deze waarde wordt enkel uitgevoerd door de vertex shader, zodat de texture coördinaten kunnen geïnterpoleerd worden tijdens de rasterisatiefase.

Voor meer details in verband met programmeren in Cg, zie [FK03]. Een screenshot van een gerenderd beeld door middel van deze shadercombinatie is zichtbaar in figuur 4.3.



Figuur 4.3: *Screenshot van refractie benadering in Cg. (Bron: "Cg Toolkit").*

```
struct inputs {
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};
```

```
struct outputs {
    float4 hPosition : POSITION;
    float4 fresnelTerm : COLOR0;
    float4 refractVec : TEXCOORD0;
    float4 reflectVec : TEXCOORD1;
};

// fresnel approximation
fixed fast_fresnel(float3 I, float3 N, float3 fresnelValues) {
    fixed power = fresnelValues.x;
    fixed scale = fresnelValues.y;
    fixed bias = fresnelValues.z;
    return bias + pow(1.0 - dot(I, N), power) * scale;
}

outputs main(
    inputs IN,
    uniform float4x4 ModelViewProj,
    uniform float4x4 ModelView,
    uniform float4x4 ModelViewIT,
    uniform float theta) {
    outputs OUT;
    OUT.hPosition = mul(ModelViewProj, IN.Position);
    // convert position and normal into appropriate spaces
    float3 eyeToVert = mul(ModelView, IN.Position).xyz;
    eyeToVert = normalize(eyeToVert);
    float3 normal = mul(ModelViewIT, IN.Normal).xyz;
    normal = normalize(normal);
    OUT.refractVec.xyz = refract(eyeToVert, normal, theta);
    OUT.refractVec.w = 1;
    OUT.reflectVec.xyz = reflect(eyeToVert, normal);
    OUT.reflectVec.w = 1;
    // calculate the fresnel reflection
    OUT.fresnelTerm = fast_fresnel(-eyeToVert, normal,
        float3(5.0, 1.0, 0.0));
   return OUT;
}
```

Codefragment 4.2: *Cg vertex shader voor refractie*

```
float4 main(in float3 refractVec : TEXCOORD0,
    in float3 reflectVec : TEXCOORD1,
    in float3 fresnelTerm : COLOR0,
    uniform samplerCUBE environmentMaps[2],
    uniform float enableRefraction,
    uniform float enableFresnel) : COLOR {
    float3 refractColor = texCUBE(environmentMaps[0],
        refractVec).rgb;
    float3 reflectColor = texCUBE(environmentMaps[1],
        reflectVec).rgb;
```

```
    float3 reflectRefract = lerp(refractColor, reflectColor,
        fresnelTerm);
    float3 finalColor = enableRefraction ?
        (enableFresnel ? reflectRefract : refractColor) :
        (enableFresnel ? reflectColor : fresnelTerm);
    return float4(finalColor, 1.0);
}
```

Codefragment 4.3: *Cg fragment shader voor refractie*

4.3 General Purpose GPU programmeren

Met de kennis die we tot nu toe opgebouwd hebben, is het niet moeilijk om in te zien dat de GPU ook toepassingen heeft buiten het computer graphics domein. Essentiëel komt het erop neer dat we de "grafische" informatie (bv textures, vertices, kleuren, ...) zullen interpreteren als iets totaal anders (bv. krachtveld, matrix, ...). Zo worden textures typisch gebruikt om datastructuren in op te slaan, en door te geven aan de fragment shader. Pointers, iets wat niet beschikbaar is in Cg, kunnen dan geëmuleerd worden door middel van indices (bv. `int`). Merk wel op dat het herschrijven van een general purpose programma voor de GPU pas nuttig is als het over een groot aantal operaties van beperkte complexiteit gaat, op steeds dezelfde data. Met andere woorden, als het op een niet artificiële manier herschrijfbaar is zodat het voldoet aan het voorgenoemde stream programming model. Meer over GPGPU programmeren is te vinden op [gpg].

4.4 Moeilijkheden

In de praktijk is het GPU programmeren (in Cg) niet zo evident als dat het op het eerst zicht lijkt. Aangezien het nog allemaal in volle ontwikkeling is, zijn er nog redelijk wat kinderziekten die opduiken. Hierbij denk ik aan driver problemen zoals het nog niet ondersteunen van bepaalde functionaliteit, of erger nog, het foutief ondersteunen ervan. Ook zijn er voor Cg nog niet bijzonder veel tools verkrijgbaar, zoals een debugger en profiler.

Het opslaan en doorgeven van datastructuren in textures zorgt ook voor extra last. Niet alleen is het dikwijls omslachtig om de datastructuur te doorlopen, maar ook is elke texture gelimiteerd qua grootte en zijn deze read-only. Ook is het aantal van deze textures nog eens beperkt. Gelukkig zijn er tegenwoordig wel extensies beschikbaar zodat er "non power of 2" textures toegelaten worden, om de pijn te verzachten.

Recursie is ook iets wat onmogelijk is, en dient geëmuleerd te worden (wat op zich geen probleem is, maar wat het programmeren gewoonlijk minder recht-toe-recht-aan maakt).

Verder is er de last met de trage overdrachtsnelheid van data van de GPU naar de CPU, vooral bij een AGP poort. Zie hiervoor hoofdstuk 3.3.

Shaders zijn beperkt in aantal instructies. Enerzijds is er een maximum opgelegd voor het aantal instructies waaruit de assemblercode bestaat, anderszijds is er ook een maximum voor het aantal instructies dat daadwerkelijk uitgevoerd

kan worden. Dit wordt respectievelijk het statisch en dynamisch aantal instructies genoemd. Het dynamisch aantal instructies is typisch meerdere malen hoger dan het statisch aantal. Indien een shaders toch dit aantal instructies probeert te overschrijden, wordt deze afgebroken en spreken we van een time-out. Dit resulteert in pixels met een ongedefiniëerde en dus onbruikbare resultaten. Tijdens het programmeren dient ook hiermee rekening gehouden te worden. Meer over deze topic is te vinden in [PF05], de FAQ sectie van [nVi] en op het forum van [gpg].

Aangezien de fragment shader een SIMD processor is, is het ook niet aangewezen om deze te beschouwen als een MIMD, tijdens het schrijven ervan. Dit zal in vele gevallen leiden tot zeer trage shaders, wat vaak ongewenst is. Een onschuldige if-then-else constructie moet dus met voorzichtigheid gebruikt worden, om efficiëntie te garanderen. Ten eerste neemt if-then constructie 4 klok cycli in beslag, en een if-then-else 6 klok cycli. Dit is relatief veel als we weten dat normaliter meerdere wiskundige operaties tegelijkertijd (op 4 tupels) in één klok cyclus kunnen uitgevoerd worden. Ten tweede kunnen we best het gebruik van if-then(-else) structuren beperken vanwege het feit dat én de if, én de else uitgevoerd worden indien er slechts 1 parallelle shader de if keuze neemt, en een andere de else keuze. Beide keuzemogelijkheden zullen uitgevoerd worden maar slechts de werkelijk uitgevoerde zal permanent gemaakt worden. Het is dus van belang om het aantal en de omvang van deze if-then(-else) constructies in te perken. Nauwgezet programmeren is dus de boodschap. Meer details over de grafische hardware architectuur, zie [PF05].

Duidelijk staat het GPGPU programmeren nog in kinderschoenen, maar gelukkig is het wel een “hot topic”. Een positief gevolg van de grote belangstelling hierin is dat er een stuwkracht aanwezig is die de vorige problemen zal oplossen, of minstens verzachten. Verzachtende omstandigheden zouden onder andere meerdere tools, betere programmeertalen, en de aanwezigheid van meerdere, grotere textures kunnen zijn.

Hoofdstuk 5

Implementatie

In dit hoofdstuk zal de GPU en CPU implementatie besproken worden. Eerst worden de gekozen gegevensstructuren behandeld, daarna de algoritmen die erop opereren.

5.1 Gegevensstructuren

Vooraleer we door middel van een fragment shader (in Cg), een beeld kunnen berekenen van een virturele scène, zijn er gegevens nodig die de scène beschrijven. Voor deze implementatie komt het erop neer dat we een verzameling driehoeken wensen (meshes), alsook een lichtbron. De driehoeken bezitten per vertex een normaal (gebruikt voor Gouraud shading), en bestaan uit één kleur. De grotere datastructuren dienen we telkens door te geven door middel van textures, de andere eventueel door uniforme variabelen[1].

In deze sectie worden de gebruikte gegevensstructuren besproken, met telkens de bijbehorende texturelayout.

5.1.1 Mesh

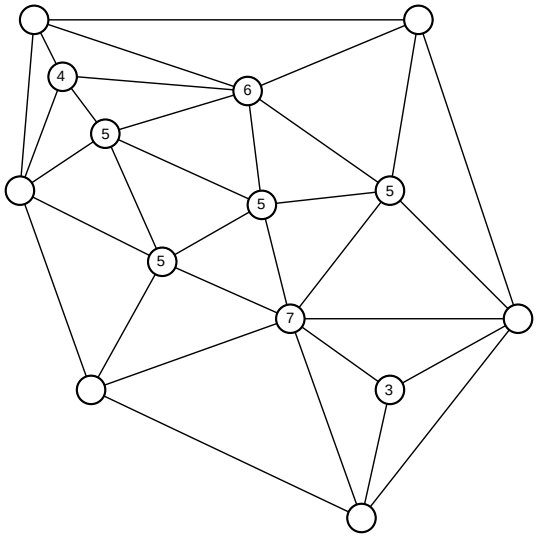
In deze sectie wordt toegelicht op welke manier de meshes zullen worden opgeslagen, en waarom hiervoor geopteerd wordt.

Structuur

Gewoonlijk worden driehoeken opgeslagen als een lijst van vertices en een lijst van indices. Per driehoek zijn er dan drie indices, die verwijzen naar de vertices uit de vertex lijst. De belangrijkste reden waarvoor dit doorgaans zo gedaan wordt, is beperking van geheugengebruik.

In de GPU implementatie heb ik geopteerd om *geen* indexering te gebruiken. De opgeslagen driehoeken kunnen gewoon beschouwd worden als een lijst van vertices, waarbij elke drie op elkaar volgende vertices een driehoek vormen. De reden hiervoor is dat we het aantal texture lookups voor het opvragen van de geometrische informatie, met 25% kunnen laten dalen. Hiervoor betalen we dus

[1] gebruik best textures voor constante data over de verschillende frames, ipv uniforme variabelen



Figuur 5.1: *Voorbeeld van een deel van een mesh. De cijfers in de cirkels geven aan door hoeveel driehoeken de betreffende vertex gebruikt wordt.*

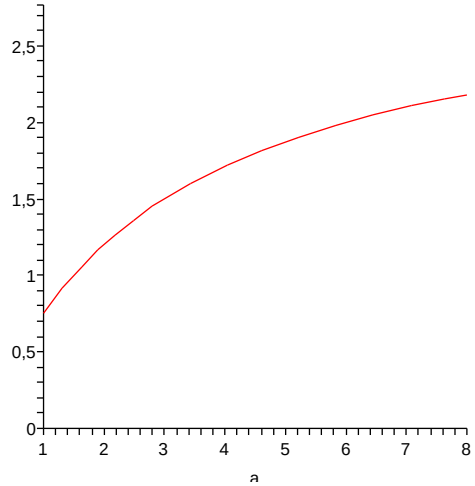
een tol, die extra texture geheugen inhoudt. Voor een gemiddeld model is er typisch minder dan twee keer zo veel geheugenruimte nodig. Dit wordt verder in deze paragraaf toegelicht. Deze keuze is gemaakt om de eenvoudige reden dat modellen die de helft van de maximale texturegrootte vullen, niet meer tot interactieve resultaten zullen leiden, vanuit interessante camera standpunten.

Onderstel dat er n_f driehoeken gegeven zijn, gedefiniëerd over n_v vertices, en de opslag van één vertex neemt één data element in beslag (nl. een drie-tupel (x, y, z)). De opslag van drie indices kunnen we duidelijk ook bijhouden in eenzelfde soort data element, namelijk als $(index_1, index_2, index_3)$. Het aantal data elementen dat nodig is indien er met indexering gewerkt wordt, noemen we m_i en is dus gelijk aan $n_v + n_f$. Als er geen sprake is van indexering, is er $m = 3n_f$ geheugen nodig.

Beschouw de mesh weergegeven in figuur 5.1. Elke vertex heeft als label een getal gekregen, dat het aantal aanliggende driehoeken voorstelt, voor de betreffende vertex. In dit voorbeeld blijkt het dat een vertex gemiddeld door 5 driehoeken "gebruikt" wordt. Laat ons het gemiddeld aantal driehoeken waartoe een vertex behoort, noteren als a. We kunnen op basis van a de geheugentoename van de voorgestelde t.o.v. de typische datastructuur, in kaart brengen. Beschouw volgende afleiding, waarbij we de verhouding van beide geheugenbehoeften $\frac{m}{m_i}$ onderzoeken.

$$\frac{m}{m_i} = \frac{3n_f}{n_v + n_f} = \frac{3n_f}{\frac{3n_f}{a} + n_f} = \frac{3n_f}{\frac{3n_f + an_f}{a}} = \frac{3an_f}{(3+a)n_f} = \frac{3a}{3+a}$$

Als we nu kijken hoeveel geheugentoename er nodig is om de mesh van figuur 5.1 op te slaan, komen we op $\frac{3 \cdot 5}{3+5} = \frac{15}{8} = 1,875$. Er is dus 87,5% meer geheugen nodig. Het is natuurlijk ook belangrijk om te weten hoeveel geheugen er in de extreemste gevallen verloren gaat of gewonnen wordt. Stel dat elke vertex



Figuur 5.2: *Functieverloop van de functie* $f(a) = \frac{3a}{3+a}$

gemiddeld maar één keer gebruikt wordt, dan krijgen we $\frac{3 \cdot 1}{3+1} = \frac{3}{4} = 0,75$, of maw een geheugenwinst van van 25%.

Beschouw volgende functie:

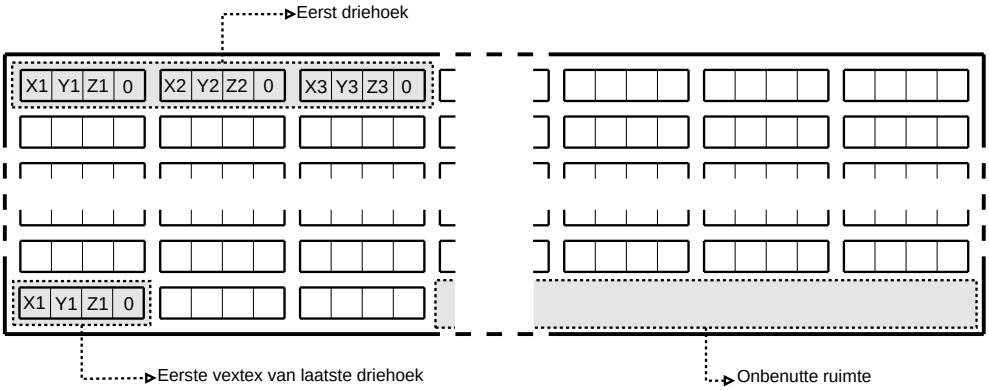
$$f : [1, +\infty[\rightarrow \mathbb{R}^+ : a \mapsto \frac{3a}{3+a}$$

Zie figuur 5.2 voor het functie verloop van f. Om de maximale hoeveelheid verloren ruimte te bepalen, zullen we eerst aantonen dat deze functie strikt monotoon stijgend is, en vervolgens naar het convergentiegedrag kijken. De afgeleide van f is $\frac{9}{(3+a)^2}$, die duidelijk positief is in haar domein. We hebben dus inderdaad te maken met een strikt monotoon stijgende functie. Het maximum is dus gewoon de waarde waarnaar deze functie convergeert, zijnde:

$$\lim_{a \rightarrow +\infty} f(a) = \lim_{a \rightarrow +\infty} \frac{3a}{3+a} = 3$$

Nu hebben we dus aangetoond dat we maximaal drie keer zoveel geheugen nodig hebben. Echter, dit zal in de praktijk niet gauw voor komen. Als bevestiging van voorgaande afleiding, zijn er ook nog 100 voorbeeld modellen bij elkaar gesprokkeld, en de parameters ervan geanalyseerd (tabel A.1). Hieruit blijkt dat voor een gemiddeld model (uit deze lijst), het gemiddelde aantal driehoeken per vertex, gelijk is aan drie. Dit wil zeggen dat er slechts 50% extra geheugen vereist is. Wat ook opvalt is het feit dat er slechts één van de geselecteerde[2] modellen meer dan twee keer zoveel geheugen vraagt.

[2]Deze modellen werden willekeurig geselecteerd van `http://www.gamasutraexchange.com`, `http://www.galiciacad.com` en `http://pascal.leynaud.free.fr/3ds/`



Figuur 5.3: *Texturelayout voor de driehoeken.*

Texturelayout

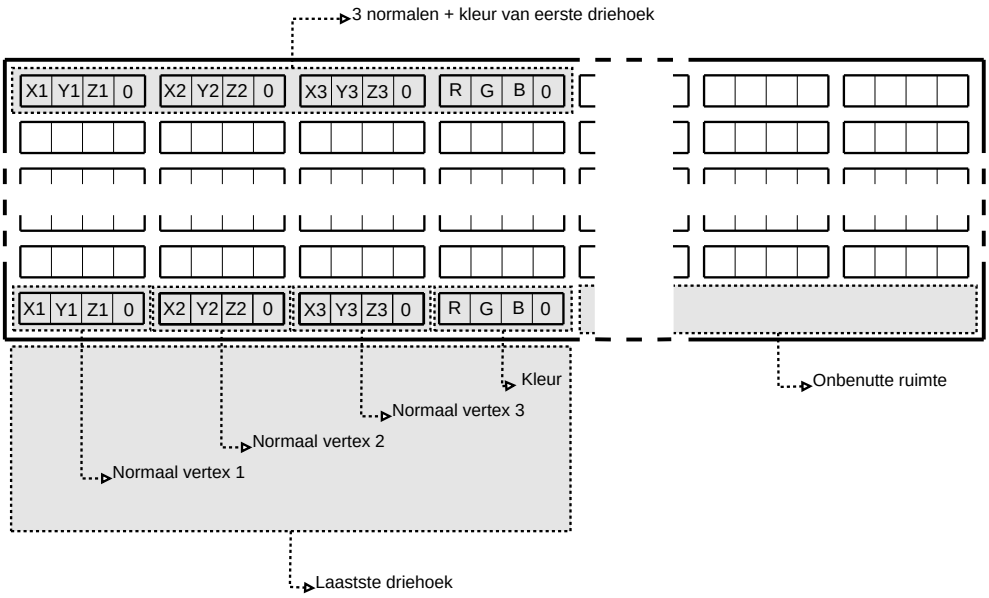
In vorige paragraaf werd besproken hoe we de meshes zullen opslaan. Hoe we deze structuur nu uiteindelijk zullen voorstellen in een texture is echter nog niet besproken. Deze sectie is hier dan ook aan gewijd.

Het is inmiddels duidelijk, en verantwoord, dat we de driehoeken zullen opslaan als een soort lijst van vertices. Aangezien we niet zomaar een type lijst voor handen hebben, maar enkel textures om data in op te slaan, zullen we zo een vertex lijst omvormen voor opslag in een texture. Een eerste zaak waar we rekening mee dienen te houden is dat er slechts een beperkte ruimte beschikbaar is per texture. Dit levert voor deze implementatie geen problematische beperkingen op. Waar wel omheen gewerkt moet worden, is het feit dat het niet mogelijk is om de gehele beschikbare texture ruimte te adresseren als een 1D texture. Dit is eenvoudig op te lossen door de data te plaatsen in een 2D texture [PDC$^+$03].

De data elementen, waaruit de texture bestaat, zijn RGBA tupels. Eigenlijk zouden RGB tupels volstaan, maar in de praktijk blijken RGBA tupels efficiënter verwerkt te worden. Het gevolg daarvan is dat we de "A", zijnde de alpha waarde, negeren. De mapping van een vertex (x, y, z) naar RGBA is gekozen als $R \leftarrow x, G \leftarrow y, B \leftarrow z$ en $A \leftarrow 0$. Deze 0 waarde is natuurlijk facultatief, maar is handig voor eventuele debugging (t.o.v. een ongedefiniëerde default waarde). Zie figuur 5.3 voor een grafische weergave.

Laat ons voor de gemakkelijkheid een texture bekijken als een (grote) matrix. Zij t een m bij n texture (en dus ook matrix), dan is $t_{i,j}$ het (in dit geval RGBA) tupel op rij i en kolom j, waarbij $i \in \{1, \ldots, m\}$ en $j \in \{1, \ldots, n\}$. Voor het omvormen van een 1D array lookup op positie p (`array[p]`), naar een 2D texture t, kunnen we een eenvoudige conversieformule gebruiken, namelijk $i = p \mod n$ en $j = \lfloor \frac{p}{n} \rfloor$. Vervolgens kan er verder gewerkt worden met $t_{i,j}$ in plaats van `array[p]`. Onbenutte tupels zullen nooit gelezen worden door de bijbehorende algoritmen, dus kunnen ook een waarde naar keuze krijgen, bv $(0, 0, 0, 0)$. Deze paragraaf geldt analoog voor tupels van andere typen (RGB, R, G, ...).

Dit gezegd zijnde, is het vrij evident om een mapping te vinden van de vertex lijst, naar een texture. In de implementatie is dit gebeurd door driehoek 1 op



Figuur 5.4: *Texturelayout voor de normalen en kleuren.*

slaan in $t_{1,1}$, driehoek 2 in $t_{1,2}$, ... Indien $j > n$, dan verhogen we i met 1, en stellen j gelijk aan 1, dit rij per rij.

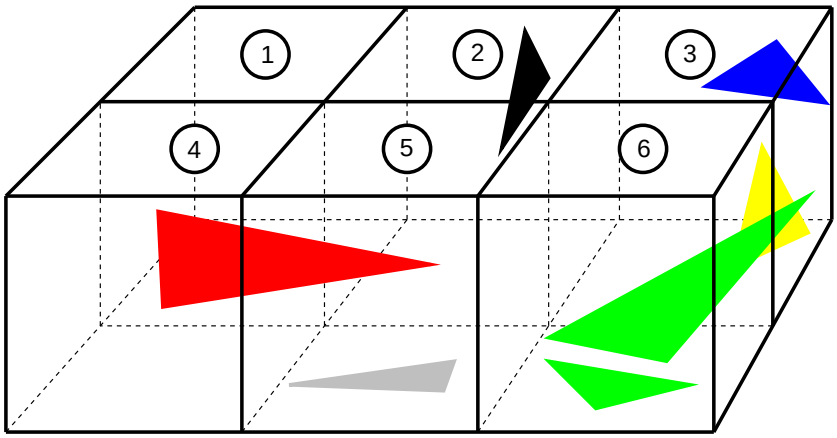
Let op, door een kleine extra moeite is het mogelijk de rechstreekste implementatie van deze uiteenzetting te optimaliseren. Als we zonder meer de breedte (n) kiezen, dienen we per texture lookup te controleren in welke rij en kolom deze valt, gegeven een volgnummer in de lijst. Indien we er nu voor zorgen dat n deelbaar is door 3, moet dit slechts eenmalig per driehoek gebeuren, ipv per vertex. Inderdaad, als we de plaats in de texture van de eerste vertex van een specifieke driehoek gevonden hebben, weten we met zekerheid dat de volgende twee vertices zich op dezelfde rij bevinden, vlak achter deze vertex.

5.1.2 Normalen en kleuren

De normalen (één per vertex) en de kleur (één per driehoek), worden samen in één texture ondergebracht. Dit gebeurd op totaal analoge wijze als in sectie 5.1.1. Voor de volledigheid is er een grafische voorstelling gegeven van de texturelayout, in figuur 5.4.

5.1.3 Regular grid

De gegevensstructuur die we in deze sectie bespreken heeft als enige doel het ray tracen van een scène te versnellen. Hoewel het zeker niet de meest optimale structuur is voor een doorsnee scène, is de gekozen acceleratiestructuur een regular grid. Deze spatiale subdivisie is makkelijker te debuggen dan bv. een k-d tree, en volstaat voor in aantal driehoeken beperkte modellen. Voor complexe modellen is een k-d tree zeker te overwegen, aangezien deze in $O(\log n)$ doorlopen



Figuur 5.5: *Regular grid bestaande uit 6 voxels. Elke driehoek bevindt zich in één of meerdere voxels.Voxel 1 bevat geen driehoeken, voxel 2 en 4 bevatten 1 driehoek, voxel 5 en 6 bevatten 2 driehoeken en voxel 3 bevat 3 driehoeken.*

kan worden, t.o.v. $O(n)$, waarbij n het aantal driehoeken is. Indien n voldoende klein is, zorgt een regular grid niet voor een bottleneck. Aangezien de focus van deze thesis eerder ligt op het vergelijken van de rendertijden tussen een zeer analoge CPU en GPU implementatie, lijkt een regular grid mij een adequate keuze.

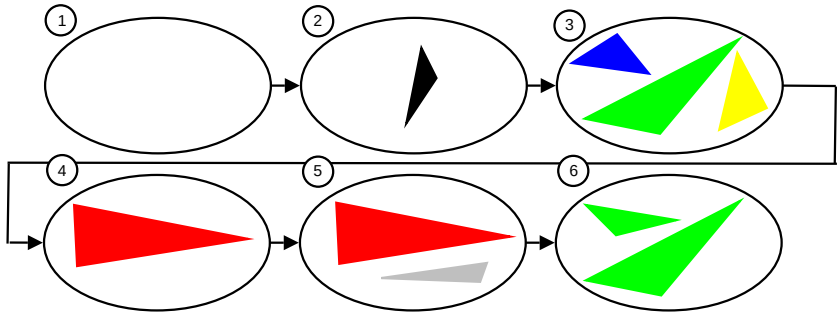
Voor de opslag van het grid maken we gebruik van twee textures. Eén texture zullen we gebruiken om per gridcel de lijst van driehoeken die hierin bevat zijn, bij te houden. De tweede texture houdt dan per grid cel een index bij waar de bijbehorende driehoek lijst zich in de eerste texture bevindt. We zullen naar deze textures verwijzen door ze respectievelijk t_{data} en t_{index} te noemen.

Laat t_{data} een matrix zijn van RGBA tupels, waarvan we enkel de R nuttig zullen gebruiken. Ook dit is een beslissing die genomen is uit effeciëntie overwegingen. De gegevens die we in t_{data} wensen te stoppen, zullen verzamelingen van driehoeken zijn, of maw een lijst van verzamelingen van driehoek. We beschouwen een eenvoudig voorbeeld, ter ondersteuning van de verdere uiteenzetting.

Figuur 5.5 geeft een virtuele scène weer, waarbij de primitieven (in dit geval driehoeken), ondergebracht zijn in een gridstructuur. De voxels, ook wel cellen genoemd, zijn genummerd van 1 tot 6. Hieruit wordt dan de lijst van driehoekverzamelingen gegenereerd. In figuur 5.6 is dit schematisch weergegeven.

Nu zijn we aangekomen op het punt dat we deze symbolische lijst gaan omzetten naar texture data. Dit kunnen we eenvoudig door verzameling per verzameling om te zetten, en die achter elkaar te plaatsen. Merk op dat we met achter elkaar plaatsen bedoelen dat de texture kolom per kolom, rij per rij gevuld wordt. We moeten eerst nog wel definiëren hoe we een verzameling driehoeken gaan voorstellen. Dit doen we door de volgnummers van de betreffende driehoeken achter elkaar te plaatsen, beëindigt door een -1. Wanneer we alle verzamelingen achter elkaar plaatsen, krijgen iets van de vorm 3 6 7 8 -1 2 9 11 -1 2 -1 -1 10 13 ... (voor een willekeurige scène). Plaats nu elk van deze getallen in de R component van de texture, en t_{data} is klaar.

Nu rest er ons nog t_{index} te construeren, hoewel het strikt genomen volstaat



Figuur 5.6: *Regular grid voorgesteld door een lijst van driehoekenverzamelingen.*

om enkel te beschikken over t_{data}. De enige reden waarvoor we t_{index} zullen construeren, is het beperken van de tijd nodig om de driehoeken van een gegeven voxel te vinden. Indien t_{index} niet zou bestaan, dienen we gemiddeld ongeveer de helft van de t_{data}texture te lezen, alvorens we de juiste voxel gevonden hebben. t_{index} biedt hier dus een oplossing voor. Voor elke voxel staat er in de texture een index, die aangeeft waar de corresponderende verzameling van driehoeken zich bevindt in t_{data}. Deze verzameling is dan beëindigt door een -1, zoals vroeger aangegeven. Zie figuur 5.7 voor een grafische voorstelling van t_{data} in combinatie met t_{index}.

De indices dienen ook in een gekende volgorde opgeslagen te zijn in een texture. Dit doen we op de gebruikelijke manier, kolom per kolom en rij per rij. Vooraleer we daartoe in staat zijn, moeten we kunnen beschikken over een volgnummer per voxel (x, y, z) in het grid met breedte s_x, hoogte s_y en diepte s_z. Dit volgnummer, noem het p, kunnen we bepalen door volgende formule: $p = x + y.s_x + z.s_x.s_y$. De indices worden weerom opgeslagen in RGBA tupels, waarvan enkel de R component gebruikt wordt.

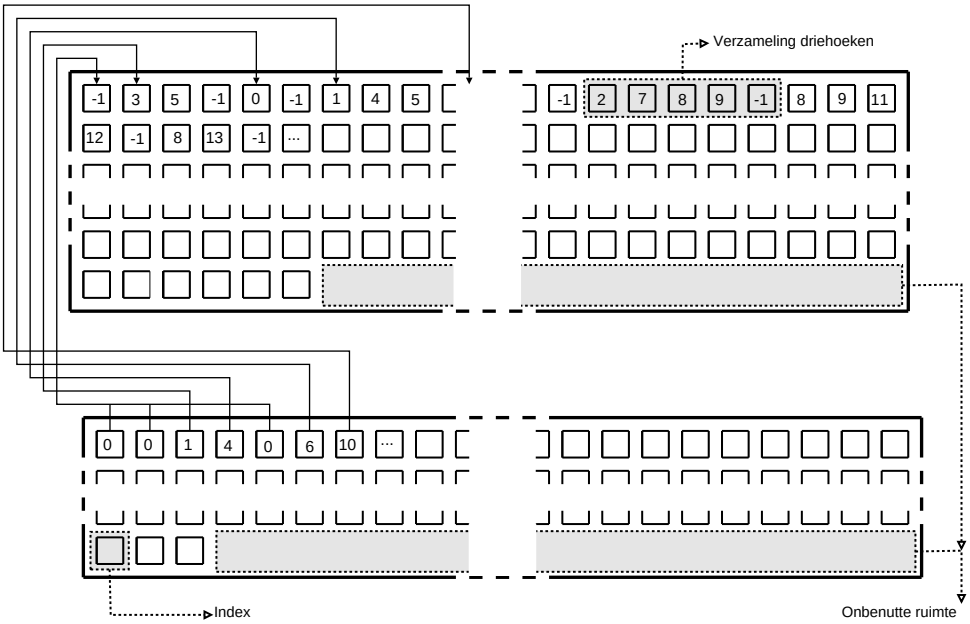
Merk op dat er in t_{data} dikwijls verscheidene opeenvolgende getallen -1 zullen zijn, indien er meerdere aanelkaar gesloten lege voxels zijn. Deze kunnen telkens vervangen worden door slechts één -1, mits de nodige aanpassingen in t_{index}. Voor de lege cellen laten we deze gewoon steeds verwijzen naar dezelfde -1 in t_{data}, die altijd aanwezig zal zijn.

5.2 Algoritmen

In deze sectie worden de gebruikte algoritmen in de GPU/CPU implementatie besproken. Deze algoritmen maken gebruik van de gegevensstructuren beschreven in sectie 5.1.

5.2.1 CPU

De algoritmen die uitgevoerd worden op de CPU zijn geschreven in de programmeertaal C++. Vergeleken met de hoeveelheid Cg-code, betreft dit het overwicht van de code. Doch, dit zijn hoofdzakelijk oninteressante algoritmen om te bespreken. Enkel het opbouwen van de gridstructuur zal in meer detail



Figuur 5.7: *Regular grid voorgesteld aan de hand van twee textures, t_{data} (bovenaan) en t_{index} (onderaan). t_{data} bevat de driehoekenverzamelingen, en t_{index} verwijzingen naar de startplaatsen van de driehoekenverzamelingen. Het einde van een driehoekenverzamelingen in t_{data} wordt aangegeven met een -1.*

uit de doeken gedaan worden.

Om de camera op een intuïtieve manier in de scène te kunnen positioneren, is er een navigatiesysteem gebouwd. Dit systeem geeft interactief beelden van de scène weer, vanuit het huidige standpunt en kijkrichting. Er is de mogelijkheid voorzien om dit beeld te bepalen door middel van een standaard geometrie engine, of door middel van ray tracing. De gerenderde beelden worden weergegeven via OpenGL [Ope], in een Qt [Tro] venster.

De modellen die geladen worden om vervolgens 2D beelden van te genereren, zijn van het type 3DS [3DS]. Deze worden ingelezen met behulp van de bibliotheek lib3ds [lib]. Vervolgens worden de ingelezen data opgeslagen in de textures, zoals in meer detail besproken in hoofdstuk 5.1.

Het opbouwen van de regular grid structuur bestaat essentiëel uit twee stappen, zijnde het bepalen van het aantal voxels, en het vullen van de voxels. Laat ons het aantal voxels noteren als n_x, n_y en n_z, waarbij de index bij de n telkens de betreffende as aangeeft. Het aantal voxels die "nodig" zijn, hangt sterk samen met het totaal aantal driehoeken, en de lengte (w_x), breedte (w_y) en diepte (w_z) van het grid. In de praktijk blijken min of meer kubische voxels, in een grid waarbij het aantal voxels in de grootte-orde ligt van het aantal driehoeken, optimaal te zijn. [Shi00]

Een formule die voldoet aan bovenvermelde eisen, ziet er als volgt uit [Shi00]:

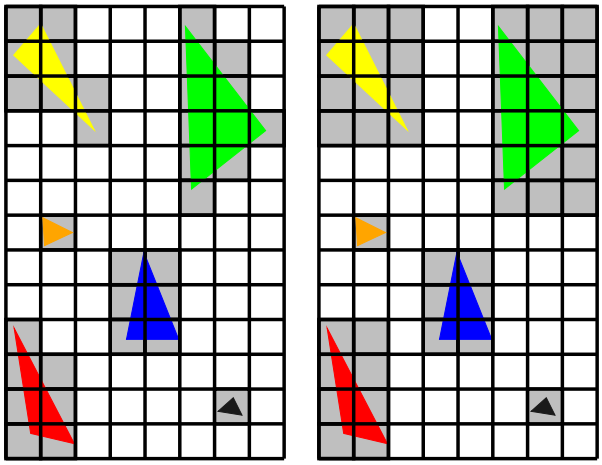
$$\begin{aligned} s &= \sqrt[3]{\frac{w_x w_y w_z}{n}} \\ n_x &= \lfloor \frac{w_x}{s} + \frac{1}{2} \rfloor \\ n_y &= \lfloor \frac{w_y}{s} + \frac{1}{2} \rfloor \\ n_z &= \lfloor \frac{w_z}{s} + \frac{1}{2} \rfloor \end{aligned}$$

Nu het aantal voxels in de drie dimensie gekend zijn, kunnen de voxels gevuld worden met driehoeken. Aangezien dit zeer recht-toe-recht-aan kan gebeuren, zal dit hier niet volledig uitgeschreven worden. Er is echter wel één belangrijke zaak die minder triviaal is, namelijk het bepalen van welke voxels er geïntersecteerd worden door een gegeven driehoek. Dit kan gebeuren door het schrijven van een 3D scanconversie algoritme voor driehoeken.

Aangezien voor deze thesis de focus niet ligt op het opstellen van een "perfecte" gridstructuur, wordt er geopteerd voor een heuristiek, die makkelijker te implementeren is. In plaats van een 3D scanconversie van de driehoek zelf, verkiezen we een scanconversie van de kleinst mogelijke bounding box rond deze driehoek. Merk op dat er zich bijgevolg soms overbodige driehoeken in bepaalde voxels bevinden, maar er zal geen enkele driehoek ontbreken. Figuur 5.8 geeft een 2D grafische voorstelling van de resultaten van beide scanconversie algoritmen.

5.2.2 GPU

In deze sectie zullen de algoritmen die uitgevoerd worden op grafische kaart, besproken worden. Er zal uitgelegd worden hoe de eerste generatie stralen worden gegenereerd, hoe deze de gridstructuur doorlopen, en op welke manier intersecties ontdekt worden met de driehoeken die zich in de voxels bevinden.



Figuur 5.8: *3D scanconversie. Links: driehoeken, rechts: bounding boxes*

Oogstralen genereren

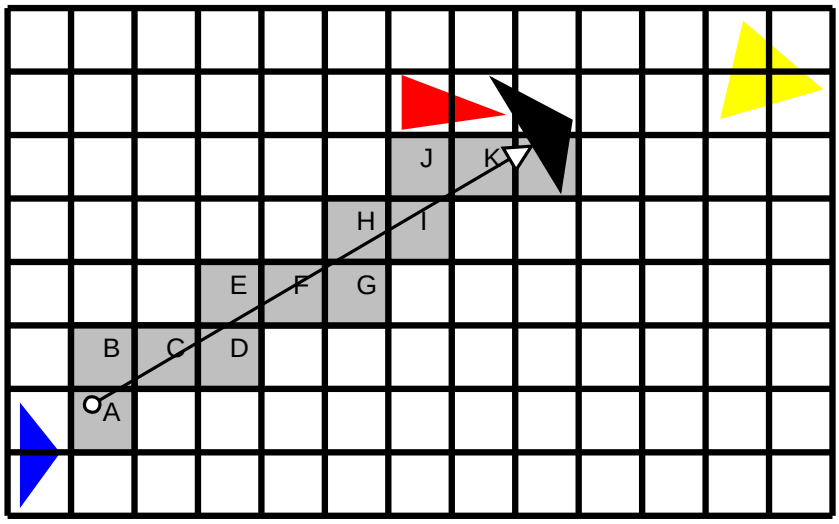
Het genereren van de oogstralen, ook wel primaire of eerste generatie stralen genoemd, kan met behulp van de grafische kaart voorzien worden met een minimum aan programmeerwerk. In de literatuur [Pur, Chr05, KL04] worden deze stralen telkens in een texture meegegeven, die al dan niet berekend wordt door een shader. Met de manier die in deze thesis voorzien wordt, is het niet nodig om al deze oogstralen via een texture door te geven, en uit te lezen met de fragment shader. Hierdoor wordt er een aanzienlijke hoeveelheid geheugen bespaard, en zal ook de texture cache minder belast worden.

Eerst wordt er in OpenGL een rechthoek getekend met als grootte de venstergrootte. Aan elk van 4 vertices kennen we een 3D texture coördinaat toe. Deze vier 3-tupels hebben telkens als waarde de positie van de corresponderende vertex van het virtuele scherm in de 3D scène. Indien we in OpenGL texture mapping aangezet hebben, zal er per pixel in de fragment shader een 3D geïnterpoleerde texture coördinaat ter beschikking zijn. Deze waarde geeft dan de 3D positie aan waarop de betreffende pixel zich bevindt in de scène coördinaten. Het enige wat er nu nog ontbreekt is één 3-tupel dat het camerastandpunt aangeeft. Dit kan makkelijk via een uniforme variabele doorgegeven worden, of mee in een andere data texture geplaatst worden.

Alle data nodig om een straal te construeren zijn dus verworven. Duidelijk is er geen nood aan het schrijven van een extra shader, of een C++ routine, die zich bezig houden met het genereren en doorgeven van de oogstralen.

Grid doorlopen

Om zinvol gebruik te maken van het opgebouwde regular grid, dienen we dit correct en op een efficiënte wijze te doorlopen. Hiervoor gebruiken we het voorgestelde algoritme door Amanatides en Woo [AW87]. Het is een snel en eenvoudig algoritme en het leent zich goed om geconverteerd te worden naar de GPU. We zullen stap voor stap het uiteindelijke GPU algoritme opbouwen.



Figuur 5.9: *Doorlopen van het regular grid. De geïntersecteerde voxels zijn aangegeven door een letter op een grijze achtergrond. Ze worden doorlopen in alfabetische volgorde, tot de intersectie gevonden is.*

Voor de duidelijkheid maken we gebruik van een situatieschets in twee dimensies, zie hiervoor figuur 5.9. Laten we een straal r beschouwen, gegeven door $r(t) = o + td$, waarbij o de oorsprong is, d de richtingsvector en t een getal groter of gelijk aan 0. Uit figuur 5.9 blijkt dat o zich in voxel A bevindt. Om de intersectie met de juiste driehoek te ontdekken, dienen de voxels in de juiste volgorde doorlopen te worden. In het geval van figuur 5.9 is dit de alfabetische lijst van voxels A, B, C, D, E, F, G, H, I, J, K, L. Dit algoritme zal telkens de waarde van t verhogen zodat er naar de volgende voxel wordt overgegaan.

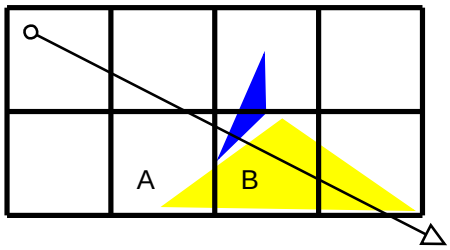
Het algoritme om de voxels te doorlopen splitsen we in twee belangrijke delen, zijnde de initialisatie en het incrementeel doorlopen van het grid. In de initialisatiefase wordt er bepaald in welke voxel de straal start. Laten we deze voxel (x, y, z) noemen. Indien de oorsprong o zich binnen het grid bevindt, dient er enkel opgezocht te worden in welke voxel dit punt zich bevindt. Het is echter ook mogelijk dat o buiten het grid geplaatst is, in dit geval moeten we bepalen in welke voxel de straal het grid binnenkomt. In beide gevallen ligt (x, y, z) dus vast. Ook tot deze fase behoort de invulling van de variabelen $stepX$, $stepY$ en $stepZ$, waarbij $stepX, stepY, stepZ \in \{-1, 1\}$. $stepX$ geeft aan of bij het doorkruisen van de straal en een X-vlak van de huidige voxel, x dient verhoogd of verlaagd te worden. Analoog geldt dit voor $stepY$ en $stepZ$. Dit wil zeggen dat we de waarden van $stepX$, $stepY$ en $stepZ$ kunnen bepalen door te kijken naar de tekens van x, y en z. Concreet geeft dit:

$$x > 0 \Rightarrow stepX = 1, \text{ anders } stepX = -1$$

$$y > 0 \Rightarrow stepY = 1, \text{ anders } stepY = -1$$

$$z > 0 \Rightarrow stepZ = 1, \text{ anders } stepZ = -1$$

Vervolgens zullen we de t-waarden bepalen die aangeven op welke plaats op de straal we het volgende x, y en z vlak intersecteren. Naar deze t-waarden



Figuur 5.10: *Driehoekintersectie buiten de voxel. In voxel A wordt er een intersectiepunt gevonden met de grote driehoek. Dit punt bevindt zich buiten voxel A. Het algoritme stopt, en bekomt bijgevolg een fout intersectiepunt.*

zullen we respectievelijk verwijzen als $tMaxX$, $tMaxY$ en $tMaxZ$. Het mininum van deze drie waarden geeft dus aan hoever we ons maximaal op de straal kunnen begeven, alvorens een nieuwe voxel binnen te gaan.

De initialisatiefase kunnen we besluiten na het bepalen van de waarden $tDeltaX$, $tDeltaY$ en $tDeltaZ$. $tDeltaX$ geeft aan hoever we ons, in eenheden van t, moeten voortbewegen op de straal om een voxelbreedte te overbruggen. Analoog geldt dit voor $tDeltaY$ die zorgt voor een overbrugging van de voxelhoogte, en $tDeltaZ$ die zorgt voor een overbrugging van de voxeldiepte.

Nu de initialisatiefase volbracht is, kunnen we starten met het doorlopen van het grid. Om te bepalen welk de volgende voxel zal zijn, zullen we controleren welke van de variabelen $tMaxX$, $tMaxY$ en $tMaxZ$ de kleinste waarde heeft. Stel $tMaxX$ heeft de kleinste waarde, dan krijgt x de waarde $x + stepX$ en passen we $tMaxX$ aan door deze de waarde $tMaxX + tDeltaX$ te geven. Dit gebeurt analoog voor $tMaxY$ en $tMaxZ$, indien één van beiden de kleinste waarde bezit.

Het algoritme zal blijven naar de volgende voxel over te gaan zolang we ons nog binnen het grid zullen bevinden, en zolang er nog geen intersectie met een driehoek gevonden is. Merk op dat het niet volstaat *een* intersectie te vinden, maar deze intersectie moet zich wel degelijk binnen de betreffende voxel bevinden om geldig te zijn. In figuur 5.10 wordt er een situatie weergegeven waar het zonder deze maatregel fout zou lopen. In voxel A bevindt zich namelijk een driehoek die de straal zou intersecteren buiten deze voxel, namelijk in voxel B. Indien deze intersectie als geldig beschouwd zou worden, geeft dit duidelijk een ongewenst resultaat. We dienen deze dus negeren, zodat in voxel B de juiste intersectie met de kleine driehoek gevonden wordt.

Op het eerste zicht lijkt het controleren of een intersectiepunt zich binnen de voxel bevindt, zes floating point vergelijkingen in te houden, namelijk één per zijvlak. Die zes floating point vergelijkingen kunnen makkelijk gereduceerd worden naar één floating point vergelijking. We bezitten namelijk de maximale t-waarde die de straal kan hebben opdat deze zich nog binnen de betreffende voxel zal bevinden. Deze waarde kunnen we vergelijken met de t-waarde van het bekomen intersectiepunt. Indien deze waarde groter is dan de maximale t-waarde, dan hebben we mogelijk te maken met een incorrecte intersectie, en zullen we deze negeren.

Aangezien het mogelijk is dat een driehoek zich in meerdere voxels tegelijker-

tijd bevindt, kan het gebeuren dat een straal meerdere keren een intersectietest uitvoert met dezelfde driehoek. In het algoritme van Amanatides en Woo is er een oplossing voorzien zodat deze meervoudige intersectietests geëlimineerd worden. Hiervoor dient er per driehoek een ID bijgehouden te worden, maar dit behoort niet tot de mogelijkheden van Cg, aangezien deze data read-only zijn. Daarom is de meervoudige intersectietest eliminatie niet opgenomen in de bijbehorende implementatie.

De implementatie van het beschreven algoritme is recht-toe-recht-aan. Het enige relevante waar rekening mee dient gehouden te worden is feit dat if-then(-else) constructies traag zijn, zoals in sectie 4.4 besproken wordt. Amanatides en Woo gebruiken een constructie waarin zeven conditionele operaties, zijnde if-then(-else) constructies, voorkomen. De herwerkte versie voor uitvoering op de GPU, beschreven in [Chr05], bevat er slechts zes. Het algoritme voorgesteld in deze thesis, bevat geen enkele conditionele operatie meer. In hoofdstuk 6 zullen de kosten van deze drie algoritmen bepaald en vergeleken worden.

Driehoeken intersecteren

Het bepalen of een straal een gegeven driehoek intersecteert, gebeurt door middel van het Möller Trumbore algoritme [MT97]. Dit algoritme vereist enkel de drie vertices die de driehoek beschrijven en een beschrijving van de straal. Er zijn dus geen voorberekende waarden nodig, zoals een vergelijking van het vlak waarin de driehoek zich bevindt.

Typische algoritmen zoeken eerst het intersectiepunt met het vlak waarin de driehoek zich bevindt, en controleren vervolgens of dit punt zich binnen de driehoek bevindt. Indien er geen intersectie is met dit vlak, zal er bijgevolg ook geen intersectie met de betreffende driehoek zijn. Dit algoritme werkt namelijk op een manier zodat er geen vlakvergelijking nodig is. Het transformeert de straal door de oorsprong te verplaatsen, en de richtingsvector van basis te veranderen. Hierdoor wordt er een vector (t, u, v) bekomen, waarbij t de afstand tot de intersectie is, en (u, v) de coördinaten van het intersectiepunt op driehoek zijn.

Voor de implementatie is er voor dit algoritme geopteerd vanwege het feit dat dit een minimale hoeveelheid geheugen in beslag neemt, snel is, en zich zeer goed leent tot een conversie naar de GPU. Het algoritme is zeer recht-toe-recht-aan te programmeren, als eenmaal de wiskundige achtergrond gekend is. Daarom zal in deze sectie de wiskundige afleiding gegeven worden.

Beschouw een driehoek met vertices v_0, v_1 en v_2 en een straal $r(t) = o + td$, waarbij o de oorspong is, en d de genormaliseerde richtingsvector. We wensen nu te weten te komen of de straal r en driehoek v_0, v_1, v_2 elkaar snijden.

Een punt p op de driehoek kan door middel van barycentrische coördinaten geschreven worden als $p(u, v) = (1 - u - v)v_0 + uv_1 + vv_2$, waarbij $u \geq 0$, $v \geq 0$ en $u + v \leq 1$. Merk op dat deze (u, v) coördinaat gebruikt kan worden voor het interpoleren van normalen, kleuren en texture coördinaten. Het snijpunt, indien aanwezig, kunnen we vinden door $r(t)$ gelijk te stellen aan $p(u, v)$, en hiervoor de juiste t, u en v waarde voor te zoeken. Indien deze niet bestaan, is er geen sprake van een intersectiepunt. De vergelijking $r(t) = p(u, v)$ wordt

verder uitgewerkt en herschreven als volgt:

$$\begin{aligned} r(t) &= p(u,v) \\ \Rightarrow o + td &= (1-u-v)v_0 + uv_1 + vv_2 \\ \Rightarrow o + td &= v_0 - uv_0 - vv_0 + uv_1 + vv_2 \\ \Rightarrow o - v_0 &= -td + uv_1 - uv_0 + vv_2 - vv_0 \\ \Rightarrow o - v_0 &= \begin{bmatrix} -d & v_1 - v_0 & v_2 - v_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} \end{aligned}$$

Laat ons uit drie hulp variabelen invoeren, om de notaties enigszins te verkorten, namelijk $e_1 = v_1 - v_0$, $e_2 = v_2 - v_0$ en $s = o - v_0$. Dit levert volgende verkorte versie van de vergelijking op:

$$s = \begin{bmatrix} -d & e_1 & e_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix}$$

Door gebruik van de regel van Cramer, kunnen we deze vergelijking oplossen naar (t, u, v):

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det(-d \quad e_1 \quad e_2)} \begin{bmatrix} \det(-d & e_1 & e_2) \\ \det(-d & t & e_2) \\ \det(-d & e_1 & t) \end{bmatrix}$$

Uit de lineaire algebra weten we dat $\det(a \quad b \quad c) = -(a \times c) \cdot b = -(c \times b) \cdot a$, waarmee we volgende vorm kunnen verkrijgen:

$$\begin{aligned} \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{(d \times e_2) \cdot e_1} \begin{bmatrix} (t \times e_1) \cdot e_2 \\ (d \times e_2) \cdot t \\ (t \times e_1) \cdot d \end{bmatrix} \\ &= \frac{1}{p \cdot e_1} \begin{bmatrix} q \cdot e_2 \\ p \cdot t \\ q \cdot d \end{bmatrix} \end{aligned}$$

Merk op dat er hulpvariabelen $p = d \times e_2$ en $q = t \times e_1$ zijn ingevoerd met als doel aan te duiden dat in de implementatie deze waarden slechts eenmalig dienen bepaald te worden. Hiermee is het doel bereikt, we kunnen eenvoudig een formule evalueren die ons een waarde voor t, u en v oplevert.

5.2.3 Converteren GPU naar CPU

In deze sectie zal behandeld worden welke aanpassingen er gemaakt zijn om de Cg code te converteren in C++ code. Er zal blijken dat er slechts weinig fundamentele aanpassingen dienen doorgevoerd te worden. Het herschrijven van het oogstralen generator, het schrijven van een Cg emulator bibliotheek en enkele syntactische aanpassingen volstaan om te kunnen beschikken over een werkende C++ versie, vertrekkende van een Cg versie.

Oogstralen genereren

Het genereren van de oogstralen zal niet meer door de grafische kaart kunnen gebeuren, waardoor we dit zelf volledig dienen te schrijven. Dit zal gebeuren aan de hand van de formule die besproken wordt in paragraaf 2.2.1.

Voor alle pixels die ingekleurd moeten worden, bepalen we gegeven de pixel coördinaten (i, j), de bijbehorende oogstraal. Deze wordt vervolgens meegegeven aan de oorsponkelijke Cg shader, die nu herschreven is als een C++ functie. Meer over de syntactische aanpassingen staat verder in deze sectie beschreven.

Cg emulator

Cg bevat een gamma aan ingebouwde datatypes en functionaliteit, die in C++ onbestaand zijn. In de Cg implementatie worden er bijgevolg verscheidene zaken gebruikt die in C++ niet zonder meer voor handen zijn.

De ontbrekende zaken, die bijgevolg nog geschreven dienden te worden, zijn types zoals `float3`, `int2` en `samplerRECT`, alsook operaties zoals `reflect`, `dot` en `cross`.

Syntactische aanpassingen

Het is spijtig genoeg niet voldoende om enkel de oogstralen generator en Cg emulator bibliotheek te schrijven. Het is onvermijdelijk om toch bepaalde syntactische veranderingen aan te brengen. De aanpassingen die van toepassing zijn op de ray tracer conversie, worden hieronder toegelicht.

uniform Het keyword `uniform` is in C++ ongekend, maar kan gewoon overal weggelaten worden.

texRECT Deze functie krijgt normaliter een `samplerRECT` en een `int2` mee als argumenten, en voert een texture lookup uit op de gegeven positie (`int2`) in de texture (`samplerRECT`). In de C++ code is `samplerRECT` gedefiniëerd als een `float *`. Deze heeft bijgevolg geen notie meer van het aantal `float`s waarnaar verwezen wordt, zijnde 4 in het geval van RGBA, en 3 in het geval van RGB. Dit is opgelost door het doorgeven van een extra paramater, zijnde het aantal `float`s.

: COLOR, : TEXCOORD0 Deze zogenaamde binding semantics bestaan niet in C++, maar kunnen zonder meer verwijderd worden. Meer hierover is te vinden in sectie 4.2.2.

main De naam van de shader, die doorgaans `main` wordt genoemd, dient een andere naam te krijgen. Zo worden er naamconflicten vermeden tussen de hoofdfunctie `main` en de naam van de geëmuleerde shader.

xyz() In Cg kunnen groepen van member variabelen geselecteerd worden via een syntax die lijkt op die van member selectie in C++. Zo levert

```
float4(1.0, 2.0, 3.0, 4.0).xyz
```

de waarde

```
float3(1.0, 2.0, 3.0)
```

op. In de C++ code is de `xyz` member voorzien door een member functie `xyz()`. In de oorspronkelijke Cg code dienen dus telkens haakjes toegevoegd te worden waar er `.xyz` gebruikt wordt.

Hoofdstuk 6

Resultaten

Dit hoofdstuk bevat een beschrijving van de bekomen resultaten van de GPU en CPU implementatie, met mogelijke verklaringen hiervoor. Aangezien veel van de bekomen resultaten systeemspecifiek zijn, zullen eerst de systeemparamaters gegeven worden. Een belangrijk deel van de verklaringen heeft betrekking tot de coherentie tussen de geschoten stralen doorheen de acceleratiestructuur. Een onderzoek van de padencoherentie zal bijgevolg een belangrijk aandeel vormen van dit hoofdstuk. Verder zal er nog een model geconstrueerd worden voor het evalueren van de kost van verschillende if-structuren. Dit zal gebruikt worden voor een vergelijkende studie tussen algoritmen uit de literatuur, en het algoritme dat voor deze thesis geïmplementeerd werd.

6.1 Systeemspecificatie

De systeemspecifieke factoren, die bepalend kunnen zijn voor de besproken resultaten, worden samengevat in tabel 6.1

Grafische Hardware	
Grafische kaart	nVidia GeForce 6800LE
Geheugen	128 MB
Aantal pixel pipelines	8
Aantal vertex pipelines	4
GPU kloksnelheid	300 Mhz
Meest geavanceerde shader model	3.0
Overige hardware	
CPU	AMD Athlon XP 2400+
AGP poort	4×
Software	
Driverversie	6.14.10.7650
Besturingssysteem	Windows XP (Service Pack 2)
Cg versie	1.3
C++	Microsoft Visual Studio .NET 2003
Gebruikte bibliotheken	Qt, OpenGL, GLEW, lib3ds

Tabel 6.1: *Systeemspecificatie.*

	axe	bed	gnome
aantal driehoeken	19392	838	49552
aantal vertices	77568	3352	198208
aantal voxels	(67, 252, 17)	(37, 25, 12)	(157, 37, 135)
percentage lege voxels	92.3561	78.5766	95.6404
gemiddeld aantal driehoeken per voxel	0.547037	1.26757	0.26434
maximum aantal driehoeken per voxel	229	156	77

Tabel 6.2: *Modelinformatie.*

6.2 Model beschrijvingen

De gegevens waarop we ons in deze sectie zullen baseren zijn gegenereerd uit drie verschillende 3DS modellen, telkens uit een bepaald interessant standpunt bekeken. We zullen er naar refereren met de bestandsnaam die ze gekregen hebben, namelijk “axe”, “bed” en “gnome”. Een screenshot van elk, gegenereerd op de GPU, is te vinden in figuur 6.7, 6.9 en 6.8.

Om de complexitieit van een model te modelleren, gebruiken we een aantal parameters van de driehoeken- en gridstructuur die hierin bepalend zijn. Concreet zal dit volgende zaken betreffen:

- aantal driehoeken
- aantal vertices
- aantal voxels
- percentage lege voxels
- gemiddeld aantal driehoeken per voxel
- maximum aantal driehoeken per voxel

Voor de gebruikte modellen in deze sectie, levert de invulling van deze parameters waarden op van tabel 6.2

6.3 Rendertijden

De tijd nodig om een beeld te vormen van een scène is afhankelijk van verscheidene factoren. Een belangrijke factor is het feit of we gebruik maken van de GPU, of CPU. Het model, aanzicht, reflectiediepte, regular grid opbouw en de volgorde waarin de pixels ingekleurd worden, hebben ook een grote invloed op de rendertijden. De hoeveelheid dat een bepaalde factor bijdraagt hierin, zal duidelijk duidelijk worden uit het vergelijkende materiaal dat zal besproken worden.

	320 × 200	**640 × 480**	**800 × 600**	**1024 × 768**
axe r_0	148	975	1227	2010
gnome r_0	420	1929	2906	4609
bed r_0	217	1075	1679	2734
bed r_1	547	2782	4344	7115
bed r_2	901	4617	7218	11840
bed r_3	1262	6484	10125	16578

Tabel 6.3: *Rendertijden op de CPU, gemeten in ms. De x van r_x geeft de maximale reflectiediepte aan.*

	320 × 200	**640 × 480**	**800 × 600**	**1024 × 768**
axe r_0	266	649	818	1133
gnome r_0	367	930	1267	1766
bed r_0	102	300	435	602
bed r_1	383	1086	1555	2219
bed r_2	718	2125	2977	4328
bed r_3	968	3063	4297	6141

Tabel 6.4: *Rendertijden op de GPU, gemeten in ms. De x van r_x geeft de maximale reflectiediepte aan.*

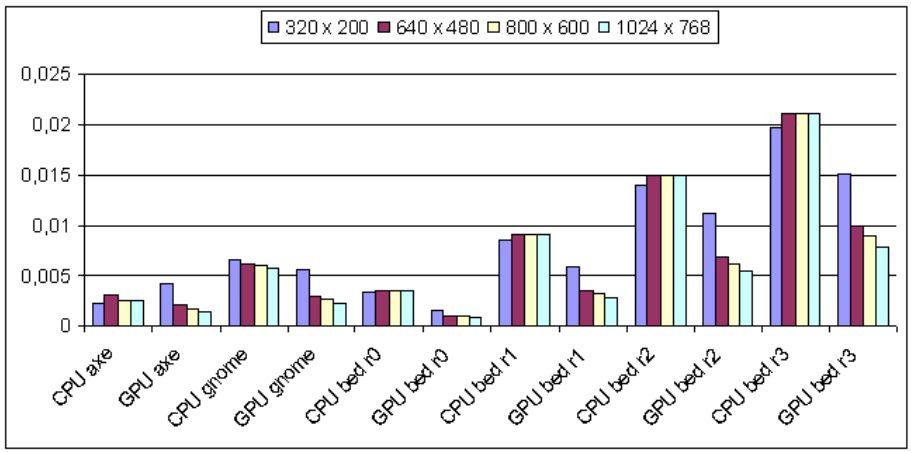
6.3.1 Verschillende configuraties

Tabellen 6.3 en 6.4 geven per model, per resolutie en per reflectiediepte, de rendertijden aan op de GPU en CPU. De reflectiediepte is telkens aangegeven door r_{diepte}. De rendertijden zijn voorgesteld in milliseconden (ms).

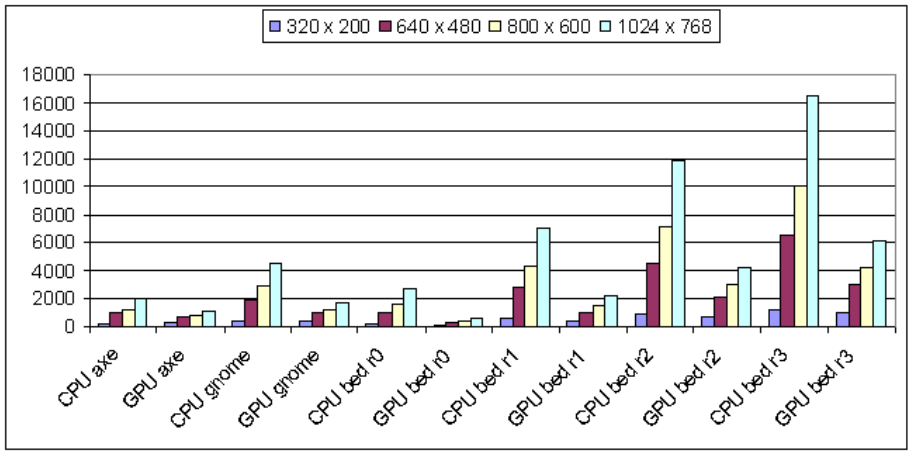
Om een betere feeling te krijgen met deze data, is deze gevisualiseerd in een grafiek (figuur 6.2). Elk element op de x-as stelt een configuratie van een model. In volgorde (1 tot 14) zijn deze: axe CPU, axe GPU, gnome CPU, gnome GPU, bed CPU, bed GPU, ..., bed r_3 CPU, bed r_3 GPU. De oneven data elementen zijn dus steeds CPU resultaten, en de even die van de GPU. Elk element op de x-as heeft vier corresponderende waarden, zijnde de rendertijden voor de 4 resoluties.

Duidelijk zijn de rendertijden op de GPU beduidend lager dan die op de CPU. Merk op dat zeker voor grotere resoluties, de GPU 3 tot 4 maal sneller is. In de grafiek weergegeven in figuur 6.1 zien we dat hoe groter de resolutie is, hoe lager de gemiddelde tijd is, nodig om een pixel in te kleuren. Op de CPU doet dit fenomeen zich niet voor.

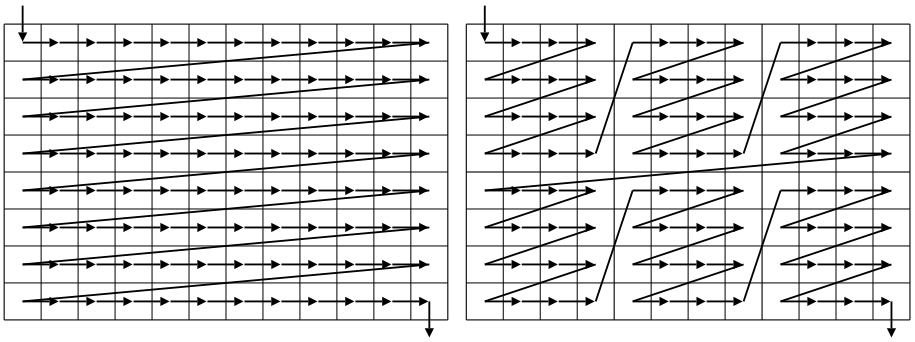
Ook uit de reflectiediepte merken we belangrijke zaken op. Hoewel deze op de GPU nog steeds enkele malen sneller uitgevoerd kunnen worden, is het wel opvallend dat er per diepte verhoging (van r_0 naar r_1, r_1 naar r_2, ...) op de GPU een groter percentage extra tijd nodig is, t.o.v. op de CPU. Op de CPU hebben we volgende lijst van rendertijden per reflectiediepte voor de bed scène: 2734, 7115, 11840, 16578. Dit wil zeggen, een toename van 160%, 66% en 40%. Op de GPU zijn de tijden voor deze scène 602, 2219, 4328, 6141. Dit levert een toename van 287%, 95% en 42%. In sectie 6.4 worden mogelijke verklaringen voor deze fenomenen besproken.



Figuur 6.1: *Rendertijden per pixel in ms.*



Figuur 6.2: *Rendertijden per frame in ms.*



Figuur 6.3: *Volgorde waarin de pixels ingekleurd worden. Links: scanlijn na scanlijn, rechts: vierkant na vierkant.*

6.3.2 Tiled rendering

In de bijbehorende implementatie kiest de GPU zelf de volgorde waarin de pixels van de gerenderde driehoek ingekleurd worden. In de praktijk is het echter wel mogelijk om deze volgorde te veranderen door de betreffende rechthoek te splitsen in kleinere primitieven, en deze in de gewenste volgorde door te sturen naar de grafische kaart. Op de CPU daarentegen dient de pixelvolgorde expliciet door de programmeur gekozen te worden.

Twee interessante pixelvolgordes die hier vergeleken worden op de GPU en CPU, zijn voorgesteld in figuur 6.3. Bij de linkse afbeelding wordt er van links naar rechts, van boven naar onder scanlijn per scanlijn gerenderd. Bij de rechtse afbeelding wordt het doelbeeld gesplitst in vierkanten van gelijke grootte, en deze deelbeelden worden vervolgens van links naar rechts en van boven naar onder gerenderd op dezelfde manier als in de rechtse afbeelding.

Voor zowel CPU en GPU is deze tiled rendering techniek geïmplementeerd. Op de GPU levert dit, zoals verwacht, geen snelheidswinst op. Integendeel zelfs, de GPU implementatie werkt daardoor minder efficiënt. Op de CPU zijn er wel voordelen aan verbonden, zoals uit tabel 6.5 blijkt. In deze tabel worden de drie gebruikte scènes gerenderd op de GPU met verschillende groottes voor het vierkant, en op de verschillende standaard resoluties. Telkens wordt het percentage snelheidswinst aangegeven ten op zichte van de normale rendering, scanlijn per scanlijn. Merk op dat er dikwijls een aanzienlijke snelheidswinst merkbaar is, maar in bepaalde gevallen is er sprake van een klein snelheidsverlies.

6.4 Mogelijke verklaringen

In deze sectie zullen er mogelijke verklaringen gegeven worden van de bekomen metingen uit sectie 6.3.

6.4.1 Configuraties

De GPU implementatie blijkt beduidend sneller beelden te kunnen genereren. Waarom zou dit zo zijn? Het betreft tenslotte, op kleine irrelevante details

model	resolutie	4×4	8×8	16×16	32×32	48×48	64×64
axe	**320 × 200**	-5	0	-2	-2	-2	-2
	640 × 480	0	0	0	0	0	0
	800 × 600	1	1	1	1	0	-1
	1024 × 768	-1	0	0	0	0	0
gnome	**320 × 200**	22	27	28	28	28	28
	640 × 480	14	14	18	18	17	8
	800 × 600	12	14	15	15	13	11
	1024 × 768	9	12	12	13	12	13
bed	**320 × 200**	1	0	5	5	16	5
	640 × 480	0	0	0	0	3	-3
	800 × 600	0	-2	2	5	9	6
	1024 × 768	-1	-1	-3	-2	1	0

Tabel 6.5: *Snelheidswinst op de CPU door het gebruik van tiled rendering ten opzichte van scanlijn rendering. De resultaten zijn aangegeven in procenten. Negatieve procenten stellen dus een snelheidsverlies voor.*

na, dezelfde code. Uit enkele case studies, en ervaringen, denk ik dat er twee belangrijke redenen voor zijn.

De eerste belangrijke reden is het feit dat de code geoptimaliseerd is voor Cg, en dan pas omgezet naar C++. Deze code zou dus nog bijgewerkt kunnen worden met als resultaat een performantiewinst. We kunnen dan echter niet meer spreken van "dezelfde" code in Cg en C++, dus de C++ is met opzet onaangepast gelaten.

De tweede belangrijke reden is de coherentie tussen de paden van de geschoten stralen. De CPU put hier enkel voordeel uit doordat er dan vrij veel cache hits zijn. De GPU daarentegen is een SIMD processor, en kan dus meerdere stralen tegelijkertijd tracen. Hoe meer de paden op elkaar gelijken, hoe meer de program flow van de verschillende pixels op elkaar zal gelijken. Dit zal dan leiden tot een efficiëntere uitvoer van de lopende shaders.

Intuïtief spreekt het voor zich dat er voor eenzelfde model, in eenzelfde grid, bekeken vanuit hetzelfde standpunt, meer coherentie is tussen de paden van de stralen indien de beeldresolutie groter wordt. Dit leidt dan weer tot een GPU voordeel, wat daardwerkelijk te zien is in grafiek 6.1. De coherentie tussen de paden van de stralen zullen we illustreren aan de hand van enkele voorbeelden.

Een pad stellen we voor als een lijst van getallen, die telkens een doorkruiste voxel aanduiden. Aan een voxel (x, y, z) kennen we een getal toe zoals beschreven in sectie 5.1.3. Een voorbeeld pad kan er als volgt uitzien:

```
[ 3167 3179 3178 2878 2877 2889 2589 2588 2288
  2300 2299 1999 1998 1698 1710 1709 1409 ]
```

Eerst wordt dus voxel 3167 gecheckt op intersecties, vervolgens 3179, ... en uiteindelijk voxel 1409. Laat ons nu aan elke verschillende lijst, een uniek getal koppelen. Zo krijgen we bijvoorbeeld:

```
0 : [ 3179 3191 2891 2890 2902 2901 2601 2613 2612
```

```
       2312 2311 2011 2023 2022 1722 1734 1733 1433 ]
1 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2011 2023 2022 1722 1721 1733 1433 ]
2 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600 2612
       2312 2311 2011 2023 2022 1722 1721 1733 1433 ]
1 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2011 2023 2022 1722 1721 1733 1433 ]
3 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2011 2023 2022 1722 1721 1733 1433 1432 ]
3 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2011 2023 2022 1722 1721 1733 1433 1432 ]
3 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2011 2023 2022 1722 1721 1733 1433 1432 ]
4 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2323 2023 2022 1722 1721 1733 1732 1432 ]
5 : [ 3479 3491 3191 3190 2890 2902 2901 2601 2600
       2612 2312 2311 2323 2322 2022 2021 1721 1733 1732 1432 ]
...
```

De unieke getallen die toegekend werden aan de paden doorheen gridstructuur, kunnen nu gebruikt worden om per pixel het gevolgde pad aan te duiden. Indien meerdere pixels hetzelfde pad volgden, kan dit eenvoudig opgemerkt worden aangezien deze hetzelfde nummer bevatten.

Merk op dat twee paden niet noodzakelijk totaal verschillend dienen te zijn, om een totaal verschillend padnummer te krijgen. Zo zijn pad 4 en 5 verschillend, hoewel deze slechts één verschillende voxel doorkruisen. Pad 4 gaat namelijk op een gegeven moment door voxel 1722, terwijl pad 5 door voxel 2021 gaat. De overige 19 voxeldoorkruisingen zijn identiek.

Beschouw de scène van screenshot 6.9. In het midden ervan snijden we een vierkantje van 16 bij 16 pixels uit dit beeld, en hiervan bekijken we de gevolgde paden. We doen dit voor de vier resoluties, zijnde 320 $\times$ 200, 640 $\times$ 480, 800 $\times$ 600 en 1024 $\times$ 768, weergegeven in tabel 6.6. Op elke pixelpositie wordt in plaats van de kleur, het padnummer getoond.

De maximum waarde per figuur, met één verhoogd, geeft aan hoeveel verschillende paden er gevolgd werden om het 16 $\times$ 16 vierkant in te kleuren. Dit is respectievelijk 233, 131, 86 en 66 voor de gebruikte scénes. Het is duidelijk dat er steeds meer paden gelijk zijn, als de resoluties hoger worden. In de situatie van figuur 6.6(a) kunnen we bij benadering stellen dat elk pad slechts één keer gebruikt wordt, dus er worden"geen" paden herbruikt. Als we dit vergelijken met 6.6(d) zien we dat elk pad gemiddeld ongeveer vier keer gebruikt wordt, wat een opmerkelijke toename is.

Wanneer we een volledig beeld genereren van de drie gebruikte scènes, in plaats van enkel een 16 $\times$ 16 vierkant, en telkens het aantal verschillende stralen weergeven, verkrijgen we tabel 6.7. Merk op dat de combinatie gnome en 1024 $\times$ 768 niets bevat. Dit is te wijten aan een geheugentekort van het systeem waarop deze resultaten werden bepaald.

Stel dat de grafische kaart een groep van, in dit geval 8, pixels selecteert om in te kleuren. Indien elk van de gegenereerde stralen hetzelfde pad volgt doorheen de gridstructuur, wil dit zeggen dat de program flow in de 8 fragment

	0	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	15	16	17	2	18	19	20	21	22	23	24	25	26	27	28	29
	30	31	32	33	34	19	35	36	37	38	39	40	41	42	43	44
	45	46	32	47	48	49	50	51	51	52	53	54	55	56	57	58
	45	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73
	74	75	75	61	76	77	78	79	79	80	81	82	83	84	85	86
	87	88	89	90	91	92	93	94	95	96	97	98	99	100	85	101
	102	103	103	104	105	106	107	108	109	110	111	98	112	113	114	115
(a)	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
	132	133	133	134	135	136	122	137	137	138	139	140	141	142	143	144
	145	146	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	160	161	161	162	163	164	165	166	166	167	168	169	170	171	172	173
	174	175	175	176	177	178	179	180	180	181	182	183	184	185	186	187
	188	189	189	190	191	192	193	194	194	195	196	197	198	199	200	201
	202	203	203	204	205	206	207	208	209	210	211	212	213	214	215	216
	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232
	0	1	2	3	4	5	5	6	6	7	8	9	9	10	11	12
	13	14	2	3	15	16	16	6	6	6	17	18	19	20	21	22
	13	23	24	25	15	16	16	6	6	26	27	28	29	30	31	22
	32	33	24	25	34	35	16	26	26	26	36	29	29	37	38	38
	32	33	24	25	39	40	40	41	42	43	36	29	29	37	38	38
	32	44	45	46	47	48	48	41	41	41	49	29	29	50	51	51
	52	53	54	55	47	48	48	41	41	41	56	57	58	50	51	51
	59	60	61	55	47	48	48	62	62	62	56	57	57	63	64	65
(b)	59	60	66	67	68	69	69	62	62	62	56	57	70	71	72	72
	59	73	74	75	76	77	78	79	62	62	56	80	80	81	72	72
	82	83	74	75	76	77	77	84	85	85	86	80	80	81	72	72
	87	88	74	75	76	89	89	90	90	90	91	92	80	81	93	93
	94	88	95	96	97	89	89	90	90	90	98	99	99	100	93	101
	102	103	104	105	106	107	107	90	108	108	109	99	99	110	111	112
	113	114	104	105	115	116	117	118	118	108	119	120	121	122	123	123
	113	114	124	125	126	116	116	127	128	128	129	121	121	122	123	130
	0	1	2	3	3	4	5	5	5	5	5	6	7	7	8	9
	10	10	11	12	3	4	5	5	5	13	13	14	15	15	16	16
	10	10	11	17	17	18	19	13	13	13	20	21	15	15	16	16
	10	10	11	22	22	23	24	25	26	20	20	21	15	15	16	16
	27	27	28	22	29	30	25	25	25	25	26	21	15	15	16	31
	27	32	33	29	29	30	25	25	25	25	25	34	35	36	31	31
	32	32	33	29	29	30	25	25	25	37	38	39	40	40	41	31
	42	42	33	29	29	30	25	38	38	38	38	39	40	40	43	43
(c)	44	44	45	46	47	48	38	38	38	38	38	39	40	40	49	49
	44	50	51	52	52	53	54	38	38	38	38	39	55	56	57	57
	50	50	51	52	52	58	59	54	54	38	60	61	56	56	57	57
	50	50	51	52	52	58	59	62	62	63	63	61	56	56	57	57
	64	50	51	52	65	66	62	62	62	62	62	67	68	56	57	69
	64	64	70	65	65	66	62	62	62	62	62	71	72	72	73	69
	74	74	70	75	75	66	62	62	62	76	76	77	72	72	78	79
	74	74	80	81	75	82	83	83	76	76	76	77	84	84	79	85
	0	0	1	1	2	2	3	4	4	4	4	5	5	6	6	7
	0	0	1	1	8	8	9	4	4	10	10	5	5	6	6	7
	0	11	12	12	8	8	13	13	14	10	10	5	5	6	6	7
	11	11	12	15	16	16	13	13	13	13	14	17	5	6	6	7
	18	18	15	15	16	16	13	13	13	13	13	19	20	21	22	23
	18	18	15	15	16	16	13	13	13	13	13	24	25	26	27	28
	18	18	15	15	16	16	13	13	13	29	29	25	25	26	26	30
	31	18	15	15	16	16	29	29	29	29	29	25	25	26	26	30
(d)	32	31	33	34	35	35	29	29	29	29	29	25	25	26	26	30
	36	36	37	38	39	35	29	29	29	29	29	25	25	40	40	41
	36	36	37	37	42	39	43	29	29	29	29	25	25	44	44	45
	36	36	37	37	42	42	46	43	43	43	47	48	48	44	44	45
	36	36	37	37	42	42	46	49	49	50	50	51	48	44	44	45
	36	36	37	37	52	52	49	49	49	49	49	53	51	54	44	45
	55	56	57	57	52	52	49	49	49	49	49	53	53	58	59	60
	61	61	62	57	52	52	49	49	49	49	49	63	63	64	64	65

Tabel 6.6: *Padencoherentie. (a)-(d) geven blokken van 16 × 16 pixels weer, bekomen uit het midden van een gerenderd beeld van de scène "bed", op vier verschillende resoluties. Elk van de 256 getallen stelt een unieke pad voor. De resoluties zijn: (a) 320 × 200, (b) 640 × 480, (c) 800 × 600, (d) 1024 × 768.*

resolutie	axe	bed	gnome	aantal mogelijke paden
320 × 200	15502	25967	34566	64000
640 × 480	87440	52084	183989	307200
800 × 600	134365	58390	280858	480000
1024 × 768	213440	64331		786432

Tabel 6.7: *Aantal verschillende paden voor raycasting, per scène, per resolutie.*

programma's gelijk is. Dit is duidelijk het ideale geval voor de fragment shader, die tenslotte een SIMD processor is.

Gelukkig zijn er voor een gelijke program flow van twee shaders, geen twee volledig identieke paden vereist. Het is niet belangrijk dat de voxels zelf gelijk zijn, maar dat het aantal driehoeken in de overeenstemmende voxels telkens gelijk zijn. Dit kan op analoge wijze gemodelleerd worden als bij de paden, enkel de unieke nummers toegekend aan de voxels, dienen vervangen te worden door het aantal driehoeken dat zich in de desbetreffende voxel bevindt. Paden van dit soort zullen we n-paden noemen. Enkele voorbeelden van typische n-paden zijn:

```
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 8 ]
```

Meestal worden er namelijk eerst een aantal lege cellen doorkruist. Vervolgens zijn er in het "gemiddelde" geval slechts een beperkt aantal voxels waarin zich driehoeken bevinden, tot er zich een intersectie voordoet. Indien we op dezelfde manier als in tabel 6.6 nummers toekennen aan de paden, bekomen we als resultaat tabel 6.8. Als we deze tabellen met elkaar vergelijken, is het opmerkelijk hoe het aantal verschillende stralen in tabel 6.8 daadwerkelijk afgenomen is ten opzichte van het aantal in tabel 6.6.

Indien de volledige beelden gerenderd worden van de drie scènes, en telkens het aantal verschillende n-paden weergegeven worden, verkijgen we tabel 6.9. Let op, deze cijfers dienen met voorzichtigheid geïnterpreteerd te worden. Uit de voorbeelden blijkt duidelijk dat er voor een groot aantal pixels, slechts zeer weinig verschillende n-paden gevolgd dienen te worden, maar enkel dit volstaat niet. Elke pixel uit de groep van pixels, die gelijktijdig ingekleurd worden door de grafische kaart, dient hetzelfde n-pad te volgen voor een optimale performantie. Wetende dat zo een groep tegenwoordig gemiddeld uit 8 tot 16 pixels bestaat, is dit niet vanzelfsprekend het geval. Hierover meer in de sectie 6.4.2.

6.4.2 Tiled rendering

De resulaten van tiled rendering, weergegeven in sectie 6.3.2, blijken op de CPU meestal een snelheidswinst op te leveren, wat niet het geval is op de GPU. In deze sectie wordt besproken waaraan dit te wijten is. De CPU en GPU worden hier afzonderlijk onder de loep genomen.

	0	1	1	2	3	3	3	4	4	5	5	5	5	6	3	4
	7	1	1	2	3	3	3	4	4	5	5	5	5	6	3	4
	7	0	0	2	3	3	3	4	4	5	8	8	8	6	3	4
	7	0	0	9	10	10	3	4	4	8	8	8	8	11	10	4
	7	0	0	9	10	10	10	3	4	8	12	12	5	6	3	4
	7	0	0	9	10	10	10	3	3	5	5	5	5	6	3	4
	7	0	0	9	10	10	3	4	4	5	5	5	5	6	3	4
	13	0	0	9	3	3	3	4	4	5	5	5	8	6	3	4
(a)	13	0	0	2	3	3	3	4	4	8	8	8	8	6	3	4
	7	0	0	9	10	10	3	4	4	8	8	8	8	11	10	3
	7	0	0	9	10	10	10	3	4	8	12	12	12	11	10	3
	7	0	0	9	10	10	10	3	3	12	12	12	12	11	10	3
	7	0	0	9	10	10	10	3	3	12	12	12	12	11	3	4
	14	0	0	9	10	10	10	3	3	12	12	8	8	6	3	4
	14	15	15	9	10	10	10	3	4	8	8	8	8	6	3	4
	15	15	15	16	17	3	3	4	4	8	8	8	8	11	10	3
	0	0	0	0	0	0	0	1	1	2	3	4	4	4	4	5
	0	0	0	0	0	0	0	1	1	1	6	4	5	5	5	5
	0	0	0	0	0	0	0	1	1	2	3	5	5	5	5	5
	0	0	0	0	0	0	0	2	2	2	3	5	5	5	5	5
	0	0	0	0	1	1	1	2	2	2	3	5	5	5	5	5
	0	1	1	1	1	1	1	2	2	2	3	5	5	5	5	5
	1	1	1	1	1	1	1	2	2	2	3	5	5	5	5	5
	1	1	1	1	1	1	1	2	2	2	3	5	5	5	5	7
(b)	1	1	1	1	1	1	1	2	2	2	3	5	7	7	7	7
	1	1	1	1	1	1	1	2	2	2	3	7	7	7	7	7
	1	1	1	1	1	1	1	2	2	2	3	7	7	7	7	7
	0	0	1	1	1	1	1	2	2	2	3	7	7	7	7	7
	0	0	0	0	1	1	1	2	2	2	3	7	7	7	7	7
	0	0	0	0	0	0	0	2	2	2	3	7	7	7	7	4
	0	0	0	0	0	0	0	1	1	2	3	7	4	4	4	4
	0	0	0	0	0	0	0	1	1	1	6	4	4	4	4	4
	0	0	0	0	0	0	1	1	1	1	1	2	3	3	3	3
	0	0	0	0	0	0	1	1	1	4	4	5	3	3	3	3
	0	0	0	0	0	0	4	4	4	4	4	5	3	3	3	3
	0	0	0	1	1	1	4	4	4	4	4	5	3	3	3	3
	1	1	1	1	1	1	4	4	4	4	4	5	3	3	3	3
	1	1	1	1	1	1	4	4	4	4	4	5	3	3	3	3
	1	1	1	1	1	1	4	4	4	4	4	5	3	3	3	3
	1	1	1	1	1	1	4	4	4	4	4	5	3	3	3	3
(c)	1	1	1	1	1	1	4	4	4	4	4	5	3	3	6	6
	1	1	1	1	1	1	4	4	4	4	4	5	6	6	6	6
	1	1	1	1	1	1	4	4	4	4	4	5	6	6	6	6
	1	1	1	1	1	1	4	4	4	4	4	5	6	6	6	6
	0	1	1	1	1	1	4	4	4	4	4	5	6	6	6	6
	0	0	0	1	1	1	4	4	4	4	4	5	6	6	6	6
	0	0	0	0	0	1	4	4	4	4	4	5	6	6	6	6
	0	0	0	0	0	0	1	1	4	4	4	5	6	6	6	7
	0	0	0	0	0	0	1	2	2	2	2	3	3	4	4	4
	0	0	0	0	1	1	2	2	2	2	2	3	3	4	4	4
	0	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
(d)	1	1	1	1	1	1	2	2	2	2	2	3	3	4	4	4
	1	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	1	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	1	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	1	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	1	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	0	1	1	1	1	1	2	2	2	2	2	3	3	5	5	5
	0	0	0	1	1	1	2	2	2	2	2	3	3	5	5	5

Tabel 6.8: *N-padencoherentie. (a)-(d) geven blokken van 16 × 16 pixels weer, bekomen uit het midden van een gerenderd beeld van de scène "bed", op vier verschillende resoluties. Elk van de 256 getallen stelt een uniek n-pad voor. De resoluties zijn: (a) 320 × 200, (b) 640 × 480, (c) 800 × 600, (d) 1024 × 768.*

resolutie	axe	bed	gnome	aantal mogelijke n-paden
320 × 200	2496	1837	3691	64000
640 × 480	9062	2384	13705	307200
800 × 600	12033	2481	18451	480000
1024 × 768	16158	2591	25418	786432

Tabel 6.9: *Aantal verschillende n-paden voor raycasting, per scène, per resolutie.*

CPU

Het implementeren van tiled rendering kan zeer recht-toe-recht-aan gebeuren op de CPU. Doch, de code wordt iets uitgebreider en levert naar alle waarschijnlijkheid een groter aantal dynamisch uitgevoerde instructies op. Samen met de de wijzigende belasting van het systeem waarop de metingen bekomen zijn, kan dit in een ongunstig geval tot een klein snelheidsverlies leiden.

Het is zo dat in de meeste gevallen vrij tot veel performantiewinst vastgesteld kan worden, hoewel de code complexer geworden is. Dit is het gevolg van een beter geheugencache gebruik. Uit de coherentie van de paden, zie tabel 6.6, is het duidelijk dat pixels binnen een beperkte omgeving in het beeld, gelijke of bijna gelijke paden opleveren doorheen de scène. Met andere woorden, het programma bezoekt in die omgeving dikwijls dezelfde gridcellen en driehoeken. Nog anders geformuleerd, het programma leest dikwijls hetzelfde geheugen, of geheugen dat zich regelmatig bevindt in de buurt van geheugen dat net gelezen werd. Dit wil zeggen dat dit geheugen zich hoogstwaarschijnlijk in de CPU-cache bevindt, waardoor er een veel snellere geheugentoegang ter beschikking is.

Bij tiled rendering wordt het geforceerd dat de pixels ingekleurd worden binnen eenzelfde omgeving, namelijk een vierkante omgeving. Indien er scanlijn per scanlijn gerenderd wordt, wordt er meestal sneller in ander geheugen gelezen, waardoor er zich meer cache misses zullen voordoen. Scanlijn na scanlijn wordt er opnieuw min of meer gewerkt in hetzelfde geheugen als de vorige scanlijn, maar vanwege de beperkte cachegrootte wordt deze telkens opnieuw overschreven met andere data. Aangezien de geheugentoegangstijden voor de cache enorm veel lager zijn dan voor het systeemgeheugen, kan dit een behoorlijke snelheidswinst opleveren.

GPU

Op de GPU blijkt het gebruik van tiled rendering enkel tot een performantieverlies te leiden. Dit is naar alle waarschijnlijkheid te wijten aan de extra overhead nodig om expliciet aan te geven in welke volgorde de pixels ingekleurd dienen te worden.

Normaal gezien worden de oogstralen per pixel bepaald, door de grafische hardware. Nu het beeld handmatig opgesplitst wordt in vierkante tegels, dienen er voor de vier hoekpunten van elke vierkant 3D texture coördinaten bepaald te worden. Aangezien deze coördinaten per beeld kunnen verschillen, worden ze telkens verstuurd naar de GPU. Voor een resolutie van 1024 × 768 en tegels van 32 × 32, levert dit bijna 10 Mb *extra* data op, bij een nauwkeurige implemen-

tatie. Zo kunnen de hoekpunten van de tegels op de grafische kaart gehouden worden, door ze op te slaan in een vertex-array.

Hoewel het niet gespecifiëerd wordt in welke volgorde de pixels ingekleurd worden door de grafische kaart, is het zeker niet verwonderlijk dat deze volgorde ook voor ray tracing gunstig is. Denk hierbij aan hardwarematige texturemapping, waarbij het ook nadelig is qua cachegebruik indien er scanlijn per scanlijn gerenderd worden. Doch, hierover wordt blijkbaar niets vrijgegeven [gpg].

Wetende dat er per beeld al snel 10 Mb *extra* data dient verstuurd te worden over de AGP poort, enkel voor het forceren van tiled rendering, is het niet verwonderlijk dat er geen snelheidsvoordeel uit tiled rendering geput kan worden, binnen deze implementatie.

6.5 Vergelijking van conditionele structuren

Zoals in sectie 5.2.2 besproken, gebruikt het algoritme van Amanatides en Woo [AW87] zeven conditionele operaties. In [Chr05] wordt dit algoritme herschreven in een andere vorm, zodat het slechts bestaat uit zes conditionele operaties. In deze thesis wordt dit nog verder aangepast, zodat er geen conditionele operaties meer van toepassing zijn.

Doch, het is niet zonder meer duidelijk welke van de drie structuren de meest efficiënte is. Daarom wordt er een model opgesteld dat per if-structuur een verwachte kost kan bepalen, voor het doorlopen van een gridstructuur, met n SIMD fragment shaders.

In figuur 6.4 worden de drie beschouwde codefragmenten uitgeschreven in een boomstructuur, in pseudocode. Elke knoop bevat code die sequentiëel van boven naar onder kan uitgevoerd worden. Een if-then-else constructie binnen een knoop wordt voorgesteld door twee uitgaande pijlen, die elk verwijzen naar een andere knoop. Indien de conditie naar `true` evalueert, wordt de linkse pijl gevolgd, anders de rechtse. Indien er geen else aanwezig is, is de rechtse pijl met bijbehorende knoop, onbestaand. Voor de drie beschouwde gevallen in figuur 6.4, levert dit telkens een boomstructuur op.

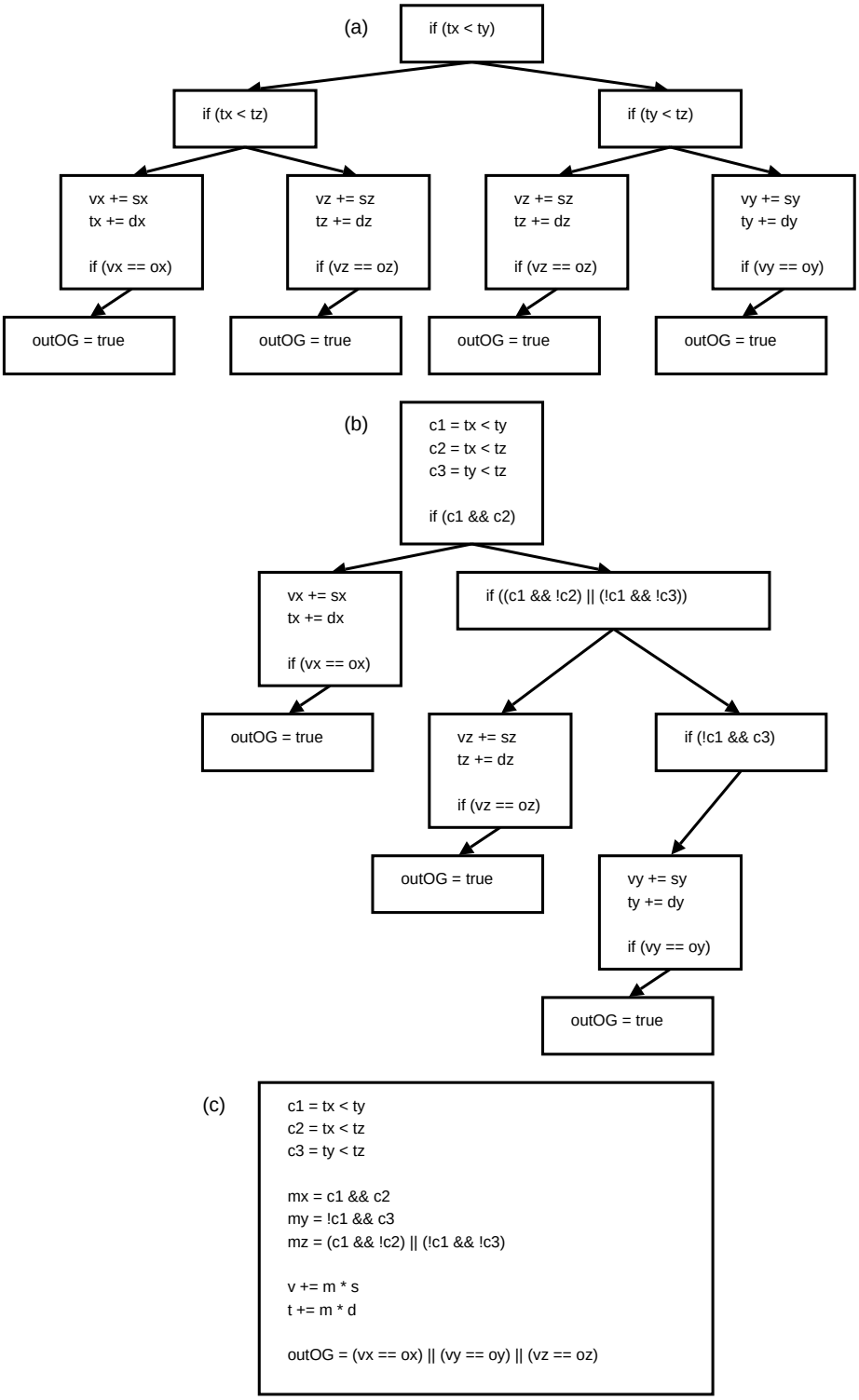
Aangezien we enkel geïnteresseerd zijn in de structuur van de bomen, en het aantal uit te voeren instructies per knoop, herschijven we deze bomen in een meer adequate vorm. Per knoop beschouwen we enkel het *aantal* instructies, in plaats van de instructies zelf. Deze aantallen worden bekomen door de uitgevoerde assemblercode te analyseren, en per knoop de overeenkomstige assemblerinstructies te tellen. Dit levert ons figuur 6.5 op.

Voor de eenvoud zullen we eerst veronderstellen dat we beschikken over een SISD[1] processor. Vervolgens zal deze discussie verder uitgebreid worden zodat deze van toepassing is voor een SIMD processor.

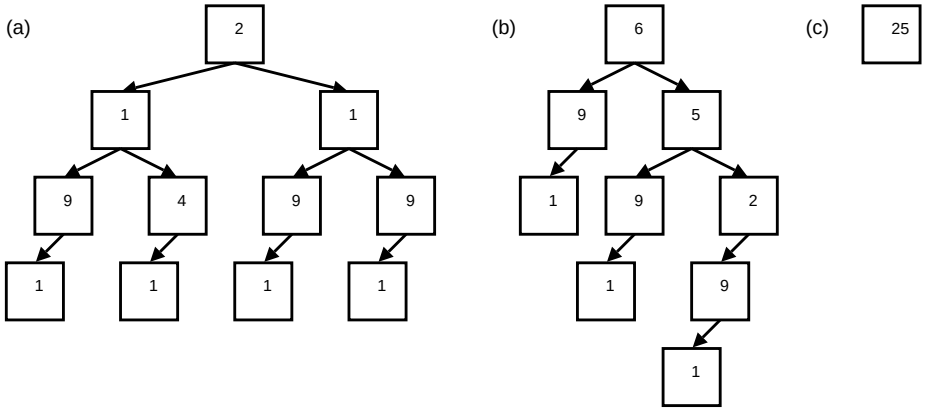
6.5.1 SISD

Een uitvoering van de beschouwde codefragmenten levert een pad op doorheen de boomstructuren van figuur 6.5. Zo een pad kunnen we voorstellen als een verzameling van knopen. De knopen van een boom stellen we voor als een 3-tupel (id, ifs, ops), waarbij:

[1] Single Instruction Single Data



Figuur 6.4: *If-structuren voor de gridtraversie. (a) Amanatides en Woo [AW87], (b) Christen [Chr05], (c) Amanatides en Woo zonder conditionele operaties*



Figuur 6.5: *De kosten van de if-structuren voor de gridtraversie. In elke knoop wordt er aangegeven hoeveel machinetaalinstructies er uitgevoerd worden in deze knoop (a) Amanatides en Woo [AW87], (b) Christen [Chr05], (c) Amanatides en Woo zonder conditionele operaties*

id: een uniek label voor de knoop

ifs: het aantal if-structuren binnen de knoop, zijnde 0 of 1

ops: het aantal instructies (operaties) dat uitgevoerd wordt binnen de knoop

Een volledige boom kunnen we dan voorstellen als de verzameling van alle mogelijke paden waarin de boom kan doorlopen worden. Een boom wordt doorlopen door te vertrekken vanuit de wortel, en telkens de corresponderende pijl te volgen afhankelijk van de conditie, tot er geen pijl meer voorhanden is. Laat ons nu de boomstructuren (a) (b) en (c) uit figuur 6.5 respectievelijk T_1, T_2 en T_3 noemen. Deze bomen kunnen we nu schrijven als volgt:

$$\begin{aligned} T_1 &= \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8\} \\ T_2 &= \{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7\} \\ T_3 &= \{R_1\} \end{aligned}$$

waarbij de paden P_i, Q_i en R_i gelijk zijn aan:

$$\begin{aligned} P_1 &= \{(1,1,2),(2,1,1),(3,1,9)\} \\ P_2 &= \{(1,1,2),(2,1,1),(5,1,4)\} \\ P_3 &= \{(1,1,2),(7,1,1),(8,1,9)\} \\ P_4 &= \{(1,1,2),(7,1,1),(10,1,9)\} \\ P_5 &= \{(1,1,2),(2,1,1),(3,1,9),(4,0,1)\} \\ P_6 &= \{(1,1,2),(2,1,1),(5,1,4),(6,0,1)\} \\ P_7 &= \{(1,1,2),(7,1,1),(8,1,9),(9,0,1)\} \\ P_8 &= \{(1,1,2),(7,1,1),(10,1,9),(11,0,1)\} \end{aligned}$$

$$
\begin{aligned}
Q_1 &= \{(1,1,6),(2,1,9)\} \\
Q_2 &= \{(1,1,6),(4,1,5),(5,1,9)\} \\
Q_3 &= \{(1,1,6),(4,1,5),(7,1,2),(8,1,9)\} \\
Q_4 &= \{(1,1,6),(4,1,5),(7,1,2)\} \\
Q_5 &= \{(1,1,6),(2,1,9),(3,0,1)\} \\
Q_6 &= \{(1,1,6),(4,1,5),(5,1,9),(6,0,1)\} \\
Q_7 &= \{(1,1,6),(4,1,5),(7,1,2),(8,1,9),(9,0,1)\}
\end{aligned}
$$

$$
R_1 = \{(1,0,25)\}
$$

Duidelijk bevat T_1 acht mogelijke paden, T_2 zeven en T_3 slechts één. Hoewel de labels vrij te kiezen zijn, merken we even op dat ze in deze tekst worden toegekend door de boom in preorde te doorlopen, en elke tegengekomen knoop een waarde toe te kennen die één hoger is dan de vorige toegekende waarde. De wortel krijgt als label een 1 toegewezen.

Om de verwachte kost voor het doorlopen van een boom T te bepalen, dient de kans op elk afzonderlijk pad P, gekend te zijn. Aangezien de condities in de knopen telkens gebaseerd zijn op de invoerdata, zijn deze kansen enkel te bepalen indien we iets over de aard van de data weten. In dit geval betreft het een straal, een gridstructuur en een bepaalde voxel binnen het grid. Op basis daarvan dient er beslist te worden welk de volgende voxel is die de straal zal doorkruisen.

De bladeren van de bomen T_1 en T_2 geven telkens aan dat het grid verlaten wordt. Voor een willekeurige straal-voxel-combinatie, en voor een voldoende groot grid, is deze kans zeer klein. Daarom dat deze paden telkens een kans gelijk aan 0 toegewezen krijgen. Ook pad Q_4 krijgt een kans van 0 toegewezen, maar dit is te wijten aan het feit dat de conditie binnen deze knoop steeds naar `true` zal evalueren. Het kan dus aangetoond worden, aan de hand van een waarheidstabel, dat deze conditie gerust weggelaten kan worden.

Voor een willekeurige straal-voxel-combinatie, is de kans dat de volgende voxel zich in de x-, y- of z-richting zal begeven, gelijk. Bij benadering kunnen we dus telkens een kans van $1/3$ toekennen aan de knopen die de voxelpositie aanpassen, in de x-, y- of z-richting. Merk op dat er zich in T_1 twee knopen bevinden, namelijk $(5,1,4)$ en $(8,1,9)$, die de op dezelfde manier de z-richting aanpassen. Beide bijbehorende paden, P_2 en P_3, krijgen bijgevolg een kans van $1/6$ toegewezen. Deze kansen houden duidelijk *geen* rekening met de coherentie tussen de verschillende stralen.

Laat ons de kans dat een pad P gevolgd wordt, noteren als $p(P)$. Samenvattend krijgen we dan volgende kansen:

$$
T_1 \begin{cases} p(P_1) = p(P_4) = \frac{1}{3} \\ p(P_2) = p(P_3) = \frac{1}{6} \\ p(P_5) = p(P_6) = p(P_7) = p(P_8) = 0 \end{cases}
$$

$$
T_2 \begin{cases} p(Q_1) = p(Q_2) = p(Q_3) = \frac{1}{3} \\ p(Q_4) = p(Q_5) = p(Q_6) = p(Q_7) = 0 \end{cases}
$$

$$
T_3 \begin{cases} p(R_1) = 1 \end{cases}
$$

Nu de kansen op elk pad ingevoerd zijn, rest er ons nog een kostenfunctie te definiëren, alvorens we de verwachte waarden van de boomstructuren kunnen bepalen. Laat ons de kost c van een knoop (id, ifs, ops) bepalen als volgt:

$$c((id, ifs, ops)) = (ifs, ops)$$

De kostenfunctie voor het doorlopen van een pad P kunnen we nu vastleggen aan de hand van de voorgaande definitie, als volgt:

$$c(P) = \sum_{n \in P} c(n)$$

De verwachte kost E voor een boom T met als kansfunctie p, kan nu bepaald worden:

$$E(T) = \sum_{P \in T} p(P)c(P)$$

Concreet levert dit volgende verwachte kosten op voor T_1, T_2 en T_3:

$$\begin{aligned}
E(T_1) &= \sum_{P \in T_1} p(P)c(P) \\
&= p(P_1)c(P_1) + p(P_2)c(P_2) + p(P_3)c(P_3) + p(P_4)c(P_4) + \\
&\quad p(P_5)c(P_5) + p(P_6)c(P_6) + p(P_7)c(P_7) + p(P_8)c(P_8) \\
&= \frac{1}{3}c(P_1) + \frac{1}{6}c(P_2) + \frac{1}{6}c(P_3) + \frac{1}{3}c(P_4) \\
&= \frac{1}{3}(3,12) + \frac{1}{6}(3,7) + \frac{1}{6}(3,12) + \frac{1}{3}(3,12) \\
&= (3, \frac{67}{6}) \\
&\approx (3,11) \\
E(T_2) &= \frac{1}{3}(2,15) + \frac{1}{3}(3,20) + \frac{1}{3}(4,22) \\
&= (3,19) \\
E(T_3) &= 1(0,25) \\
&= (0,25)
\end{aligned}$$

Er vanuitgaande dat alle instructies, behalve conditionele, op een SISD processor evenveel klok cycli in beslag nemen, is het duidelijk dat T_1 gemiddeld beter zal presteren dan T_2. T_3 is moeilijker om te vergelijken, aangezien deze meer instructies zal uitvoeren dan T_1 en T_2, maar enkel niet-conditionele. Dit is dus afhankelijk van de onderliggende processor.

6.5.2 SIMD

Het SISD model, beschreven in sectie 6.5.1, wordt in deze sectie verder uitgebreid naar een SIMD model. Laat ons een SIMD processor beschouwen die n dataelementen in parallel kan verwerken. We spreken in dit geval over n fragment shaders. Processoren van deze soort bevatten slechts één instructiewijzer, wat wil zeggen dat de program flow van de n shaders identiek dient te

zijn. Dynamic branching op basis van de invoerdata behoort dus niet tot de mogelijkheden.

Toch is het in Cg geen probleem om dynamic branching, op basis van de invoerdata, te voorzien. Dit is het gevolg van een interessante eigenschap van de GPU. Het is namelijk mogelijk om instructies uit te laten voeren, zonder dat de resultaten ervan permanent gemaakt worden. Het al dan niet permanent maken van de resulaten van een instructie, wordt beslist aan hand van een meegegeven conditie.

Hoe kan deze eigenschap nu gebruikt worden voor het voorzien van dynamic branching? Indien de instructiewijzer beland is bij een conditionele operatie, wordt er voor elk van de n shaders bepaald welk subpad deze dient uit te voeren. Met andere woorden, op basis van de conditie dient er beslist te worden of de linkse of de rechtse pijl gevolgd zal worden. Indien alle n shaders hetzelfde subpad kiezen, wordt enkel dit pad verder doorlopen. Indien bepaalde shaders het linkse, en andere het rechtse subpad kiezen, zullen beide subpaden achtereenvolgens doorlopen worden. Elke afzonderlijke shader maakt dan enkel de resultaten die van toepassing zijn, permanent. Op die manier wordt schijnbaar enkel het daadwerkelijk gevolgde pad uitgevoerd, hoewel het best mogelijk is dat er meerdere paden worden doorlopen, waarvan de resultaten dan genegeerd worden.

Wanneer er knopen aanwezig zijn die zich in meerdere paden bevinden, dienen die door de SIMD processor maar één keer uitgevoerd te worden. Dit kunnen we concluderen uit de werking van dynamic branching, en het feit dat we te maken hebben met een boomstructuur. In een boomstructuur kan er namelijk telkens maar op één enkele wijze in een bepaalde knoop terecht gekomen worden. Dit leidt ons tot de volgende kostenfunctie c, die de kost weergeeft voor het doorlopen van n al dan niet verschillende paden:

$$c(P_1, \ldots, P_n) = c(P_1 \cup \cdots \cup P_n)$$

De kans op het volgen van n gegeven paden, genoteerd als $p(P_1, \ldots, P_n)$, is gelijk aan het product van de kansen op elk afzonderlijk pad:

$$p(P_1, \ldots, P_n) = p(P_1) \ldots p(P_n)$$

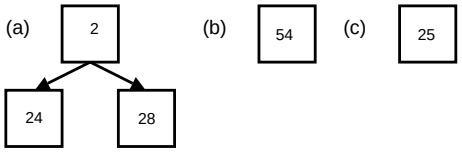
Nu zijn we in staat om de verwachte kost E te bepalen, voor een boomstructuur T, die doorlopen wordt door n shaders. Volgende formule geeft dit weer:

$$E(T, n) = \sum_{P_1, \ldots, P_n \in T} p(P_1, \ldots, P_n) c(P_1, \ldots, P_n)$$

Concreet levert dit voor deze boomstructuren en deze GPU, volgende waarden op:

$$\begin{aligned} E(T_1, 8) &= (6,45; 33,72) \\ E(T_2, 8) &= (5,84; 39,75) \\ E(T_3, 8) &= (0; 25) \end{aligned}$$

Merk op dat $n = 8$ aangezien de gebruikte grafische kaart 8 parallelle pixel pipelines bevat. Voor meer systeemeigenschappen, zie sectie 6.1.



Figuur 6.6: *De kosten van de geoptimaliseerde if-structuren voor de gridtraversie. In elke knoop wordt er aangegeven hoeveel machinetaalinstructies er uitgevoerd worden in deze knoop (a) Amanatides en Woo [AW87], (b) Christen [Chr05], (c) Amanatides en Woo zonder conditionele operaties*

Duidelijk is de laatste methode de meest efficiënte. Voor T_1 en T_2 is het opnieuw niet zonder meer duidelijk, aangezien de kost afhankelijk is van de onderlinge kost tussen conditionele en niet-conditionele operaties.

Het is echter wel belangrijk om op te merken dat Cg, indien niet nader gespecifiëerd, veel van deze if-structuren weg-optimaliseert. Om het aantal instructies in elke knoop te bepalen, werd in dit geval tijdens het compileren expliciet geforceerd om de if-structuren te behouden. Conditionele sprongen op de GPU zijn namelijk duur, waardoor het soms efficiënter is om telkens elk pad te doorlopen, en enkel te werken met het conditioneel permanent maken van bepaalde instructies. De kosten van de geoptimaliseerde boomstructuren zijn weergegeven in figuur 6.6. Laat ons deze bomen respectievelijk T_1', T_2' en T_3' noemen. De verwachte kosten hiervoor zijn:

$$\begin{aligned} E(T_1', 8) &= (1; 56, 80) \\ E(T_2', 8) &= (0; 54) \\ E(T_3', 8) &= (0; 25) \end{aligned}$$

Vanaf dit punt is het duidelijk dat T_1 op de GPU minder effiïent is dan T_2, en dat T_2 dan weer minder efficiënt is dan T_3. Dit wordt ook empirisch bevestigd door een implementatie van deze drie verschillende structuren. Gemiddeld vergt T_1 ongeveer 13% extra tijd om een beeld te genereren, ten opzicht van T_3. T_2 vergt gemiddeld slechts 5% extra rekentijd. De cijfers waarop deze resultaten gebaseerd zijn, worden weergegeven in tabel 6.10.

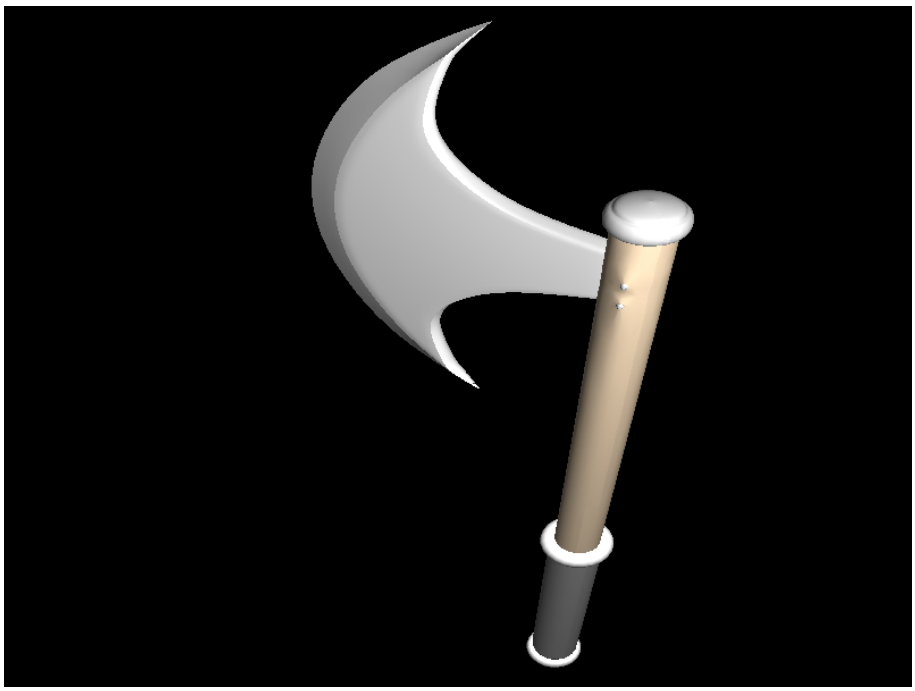
In deze paragraaf is er een duidelijke indicatie gegeven dat er een aanzienlijke snelheidswinst kan geboekt worden, door het herschrijven van het Amanatides en Woo algoritme, zodat het geen gebruik meer maakt van conditionele operaties. Het model dat hier beschreven wordt, kan makkelijk veralgemeend en toegepast worden op andere if-structuren.

Boom	Scène	axe	gnome	bed
T_1	**320 × 200**	305	410	133
	640 × 480	734	1096	367
	800 × 600	1005	1516	516
	1024 × 768	1305	1985	700
T_2	**320 × 200**	305	399	107
	640 × 480	718	1020	336
	800 × 600	968	1399	484
	1024 × 768	1211	1816	637
T_3	**320 × 200**	283	386	114
	640 × 480	672	953	326
	800 × 600	880	1312	450
	1024 × 768	1156	1703	605

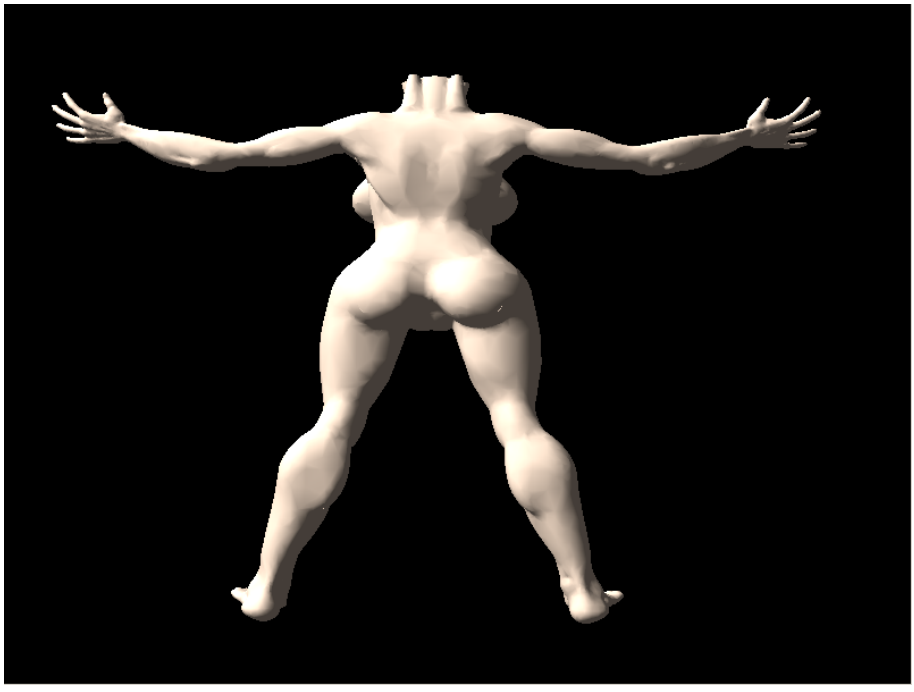
Tabel 6.10: *Rendertijden gebruikmakende van de boomstructuren T_1, T_2 en T_3, gemeten in ms.*

6.6 Screenshots

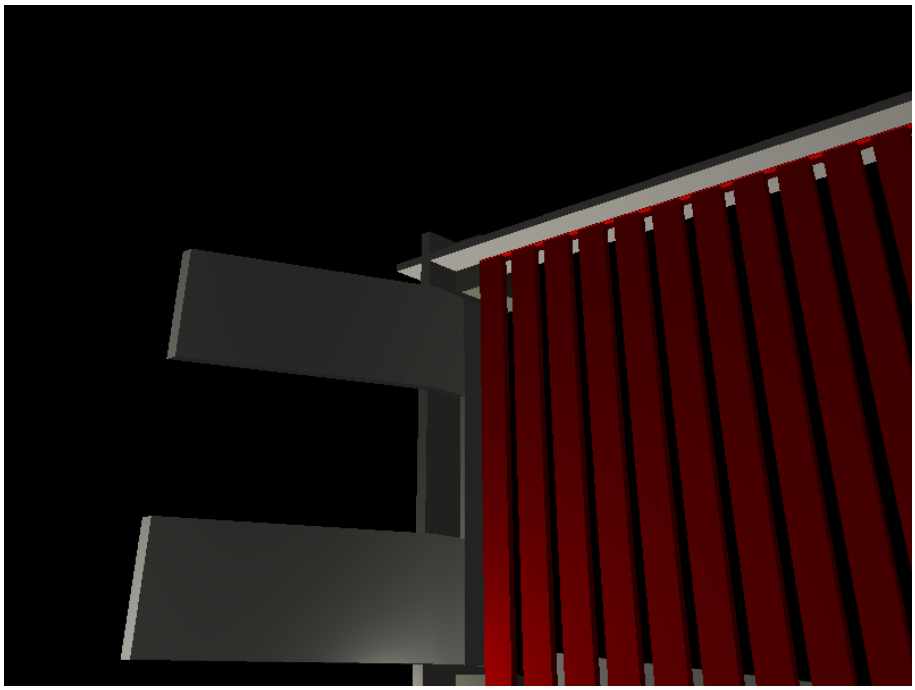
In deze sectie worden enkele screenshots getoond van verscheidene scènes. Deze beelden zijn gegenereerd door de voorziene GPU-implementatie. Telkens wordt de rendertijd, de beeldresolutie, en de hoeveelheid driehoeken waaruit de betreffende scène bestaat, vermeld.



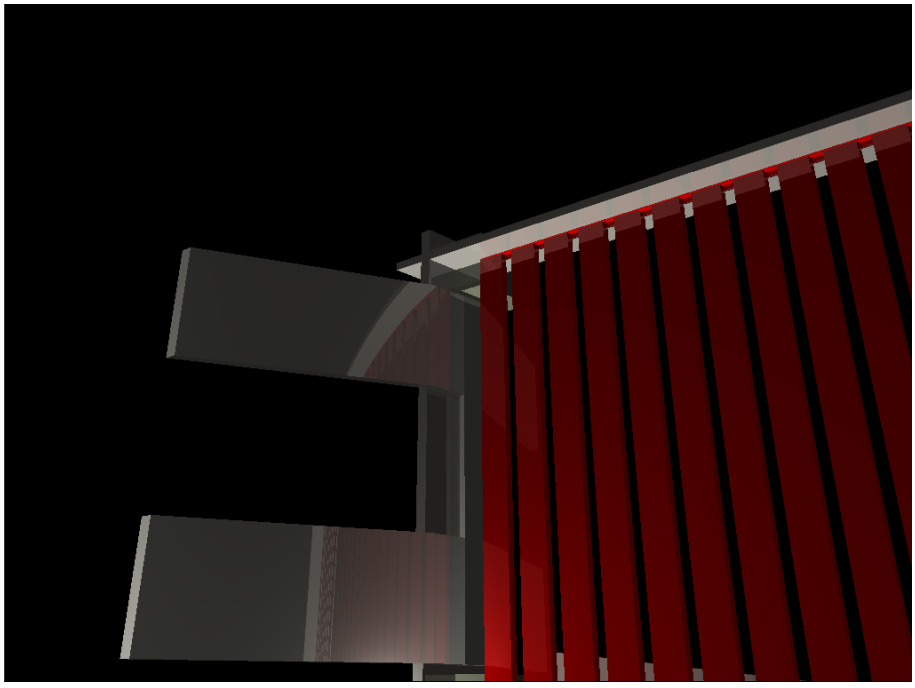
Figuur 6.7: *Axe (800 × 600). 19392 driehoeken. Rendertijd: 818 ms.*



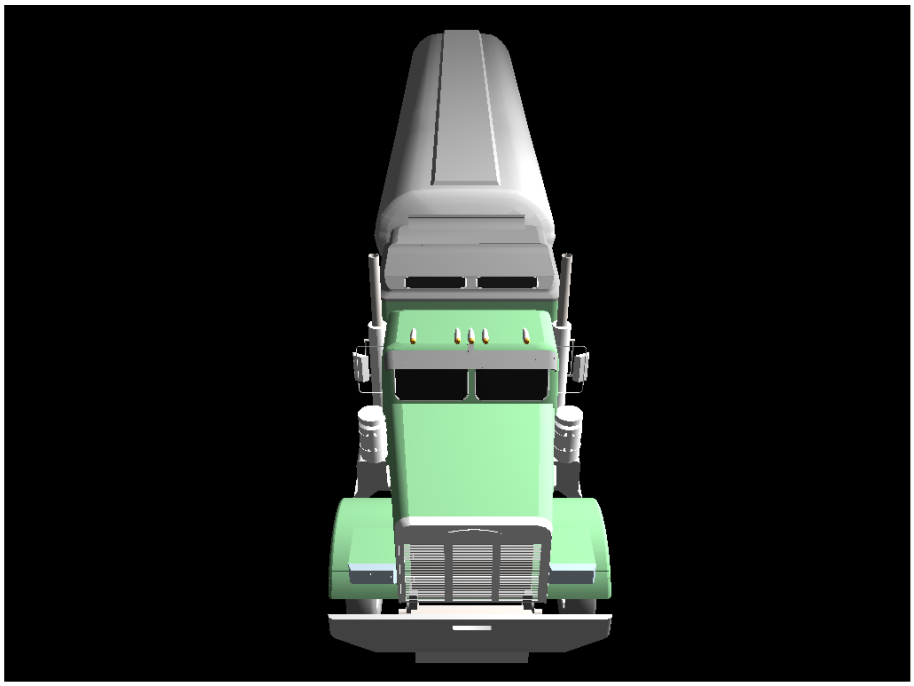
Figuur 6.8: *Gnome (800 × 600). 49552 driehoeken. Rendertijd: 1267 ms.*



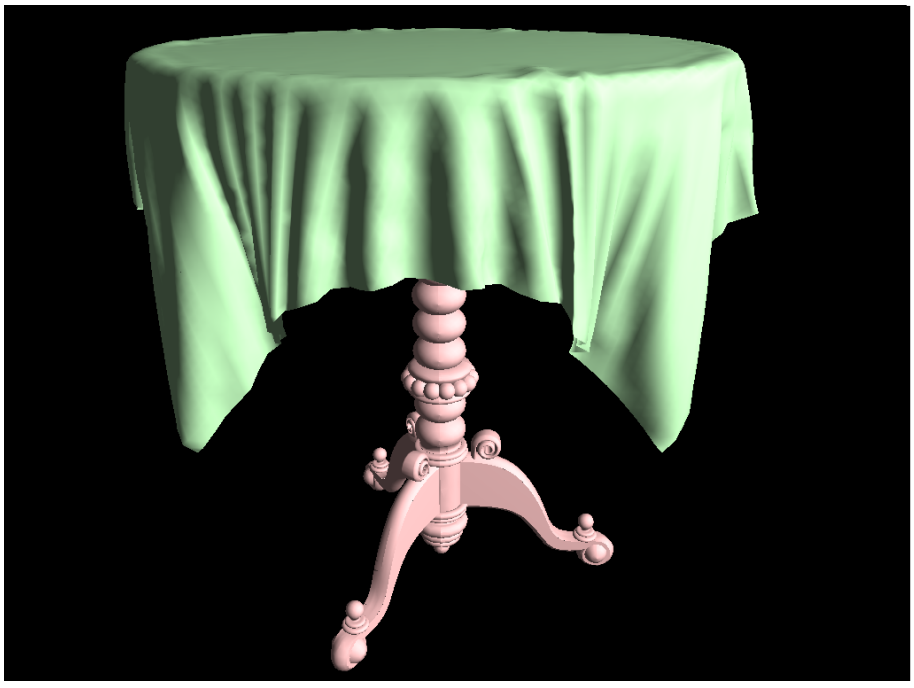
Figuur 6.9: *Bed (800 × 600). 838 driehoeken. Rendertijd: 435 ms.*



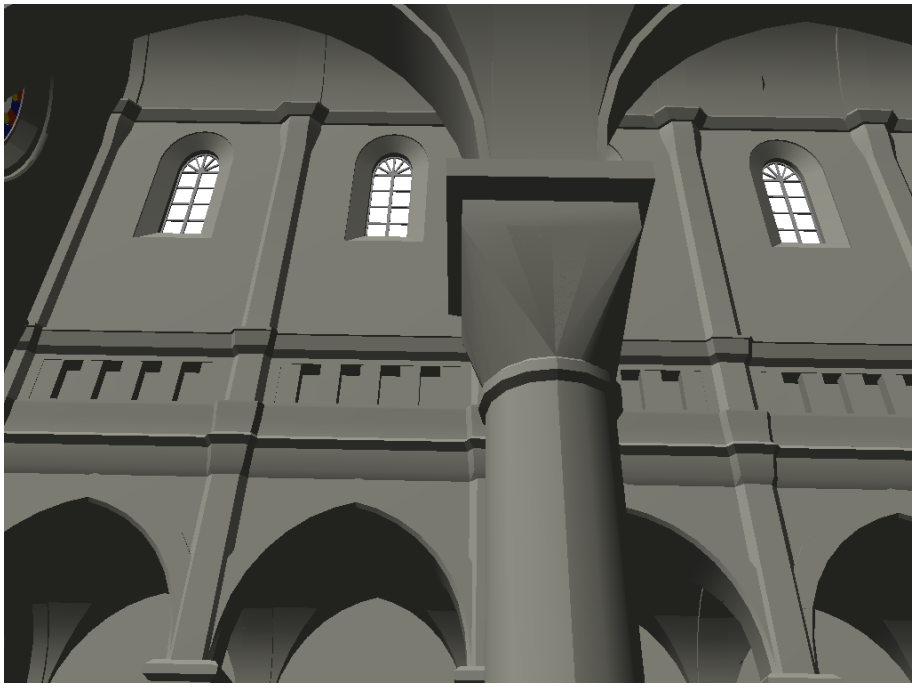
Figuur 6.10: *Bed (1024 × 768). 838 driehoeken. Reflectiediepte: 2. Rendertijd: 4328 ms.*



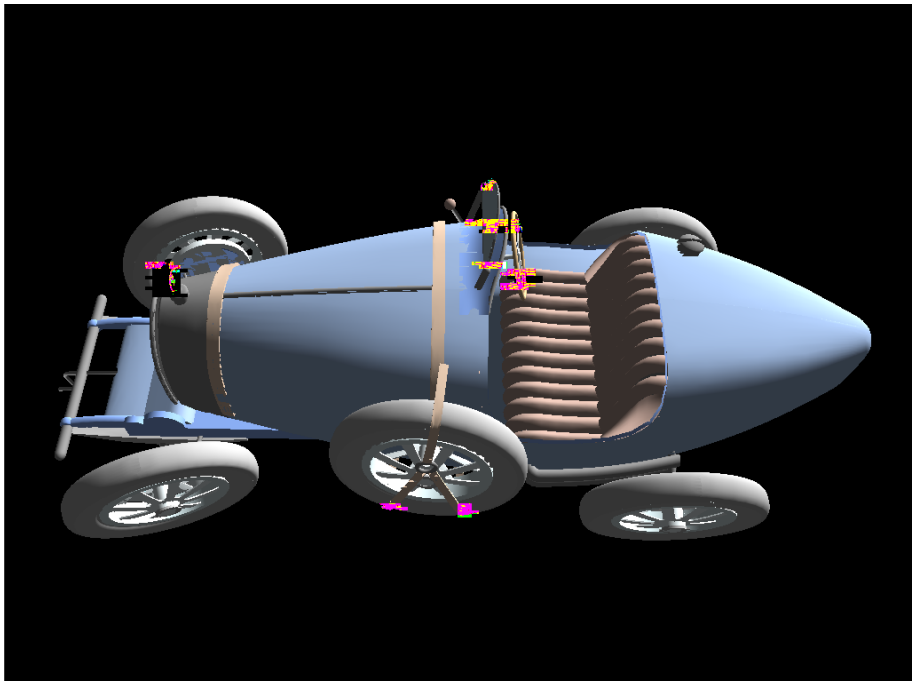
Figuur 6.11: *Truck (1024 × 768). 61169 driehoeken. Rendertijd: 1968 ms.*



Figuur 6.12: *Tafel (1024 × 768). 53046 driehoeken. Rendertijd: 2782 ms.*



Figuur 6.13: *Kerk (1024 × 768). 80479 driehoeken. Rendertijd: 4812 ms.*



Figuur 6.14: *Bugatti met time-outs (1024 × 768). 201036 driehoeken. Rendertijd: 4385 ms.*

Hoofdstuk 7

Toekomstig werk

Zoals is aangehaald in sectie 4.4, is het aantal instructies, die uitgevoerd kunnen worden binnen een fragment shader, beperkt. Indien dit aantal overschreden wordt, doet er zich een time-out voor, en is de kleurwaarde van de betreffende pixel ongeldig. Binnen deze implementatie wordt er niet gegarandeerd of er zich al dan niet time-outs zullen voordoen. Dit zou wel kunnen gebeuren door beperkingen op de scène te leggen, zoals een maximale gridgrootte, en een maximaal aantal driehoeken binnen een voxel. Door de assemblercode van de shader te analyseren, kan er bijgevolg aangetoond worden hoeveel instructies er in het extreemste geval nodig zijn. De beperkingen op de scène en / of de shadercode, kunnen dan aangepast worden tot er voldaan wordt aan het vooropgestelde maximale aantal instructies. Een geautomatiseerde manier om onder een bepaald maximaal aantal instructies te blijven, is dus gewenst.

In de voorziene GPU implementatie is de CPU onbenut tijdens het renderen van een beeld. Dit opent perspectieven, waar er sprake is van een samenwerking tussen de GPU en CPU. Op een naïeve manier zou dit kunnen gebeuren door een deel van de pixels aan de CPU toe te vertrouwen, en het andere deel aan de GPU. Een taakverdeler kan dan bepalen welke pixels aan welke hardware toegewezen worden. Dit zou dan op basis van informatie van voorgaande beelden kunnen worden beslist. Technieken die gekend zijn uit gedistribueerde rendering, zoals beschreven in [WSB01], kunnen hierbij zeker een bijdrage leveren.

Als acceleratiestructuur zou er een kd-tree gebruikt kunnen worden, in plaats van een regular grid. Ondertussen is er een paper verschenen waarin een vergelijkende studie gemaakt is tussen geen acceleratiestructuur, een regular grid, en een kd-tree. Ook op de GPU blijkt de kd-tree de meest effiënt te doorlopen spatiale subdivisie te zijn. Voor meer details, zie [FS05].

Hoofdstuk 8

Conclusie

In deze thesis zijn we vertrokken van een algemene beschrijving van globale illuminatie, aan de hand van de rendering vergelijking. Vanuit snelheidsoverwegingen wordt dit algemene model vereenvoudigd tot ray tracing, dat niet voldoet aan de rendering vergelijking. Doch, de fundamentele concepten, zijnde stralen schieten doorheen een scène, blijven bestaan binnen ray tracing.

Een implementatie van ray tracing op GPU en CPU zijn voorzien, respectievelijk geschreven in Cg en C++. De originele versie is geschreven en geoptimaliseerd voor Cg, en vervolgens geconverteerd naar C++. De GPU versie is beduidend sneller, zelfs drie tot vier maal voor scènes gerenderd op hoge resoluties. Hoe hoger de maximale toegelaten reflectiediepte was, hoe minder voordeel de GPU vertoonde. Beide fenomenen worden verklaard aan de hand van de coherentie tussen de paden die de stralen volgen doorheen de gridstructuur. Hieruit blijkt duidelijk dat de GPU zich goed leent voor ray tracing.

Het schrijven van de ray tracer implementatie op GPU is een delicate zaak. Er dient uiterst voorzichtig geprogrammeerd te worden, bij gebrek aan tools zoals een debugger en profiler. Het gebruik van dynamic branching is wel mogelijk, maar ondermijnt het voordeel van het parallellisme, aangezien er op een SIMD processor gewerkt wordt. Ook de mogelijkheden op zich, zijn vrij beperkt. Alle iets of wat uitgebreide datastructuren dienen geconstrueerd te worden uit één of meerdere textures. Recursieve functies en pointers behoren ook niet tot de mogelijkheden. Het beperkt aantal instructies die uitgevoerd kunnen worden binnen een shader, zorgt ook voor een hele last. Tijdens het programmeren zou er dus rekening gehouden moeten worden met het aantal uitgevoerde instructies op elk moment, om op het juiste tijdstip de shader zichzelf te laten beëindigen. Dit is echter een zeer omslachtige manier van werken. De GPU heeft duidelijk veel potentiëel, maar het is momenteel nog niet evident om dit tenvolle te benutten.

Het GPU programmeren zal naar alle waarschijnlijkheid zodanig evolueren, dat de besproken moeilijkheden verdwijnen ten gevolge van betere hardware en de aanwezigheid handige van tools. Steeds meer systemen zullen in de toekomst een krachtige grafische kaart bezitten, zodat de GPU zal uitgroeien tot een parallelle coprocessor, die op haast elk systeem aanwezig zal zijn.

Bijlage A

Vertex-driehoek coherentie

Tabel A.1: *Vertex-driehoek coherentie, waarbij n_v het aantal vertices is, n_f het aantal driehoeken, m_i het vereiste geheugen met indexering, m het vereiste geheugen zonder indexering, a het gemiddeld aantal keer dat een vertex gebruikt wordt, $f(a)$ de verhouding m/m_i.*

n_v	n_f	m_i	m	a	$f(a)$
56	48	104	144	2,571428571	1,384615385
160	152	312	456	2,85	1,461538462
270	354	624	1062	3,933333333	1,701923077
291	514	805	1542	5,298969072	1,91552795
304	564	868	1692	5,565789474	1,949308756
347	136	483	408	1,175792507	0,844720497
412	580	992	1740	4,223300971	1,754032258
432	804	1236	2412	5,583333333	1,951456311
438	602	1040	1806	4,123287671	1,736538462
488	806	1294	2418	4,954918033	1,86862442
491	644	1135	1932	3,934826884	1,702202643
492	204	696	612	1,243902439	0,879310345
541	846	1387	2538	4,691312384	1,829848594
546	974	1520	2922	5,351648352	1,922368421
560	1048	1608	3144	5,614285714	1,955223881
594	1160	1754	3480	5,858585859	1,984036488
595	592	1187	1776	2,98487395	1,49620893
688	1160	1848	3480	5,058139535	1,883116883
706	1408	2114	4224	5,983002833	1,998107852
716	1138	1854	3414	4,768156425	1,841423948
772	756	1528	2268	2,937823834	1,484293194
835	1376	2211	4128	4,943712575	1,867028494
850	1668	2518	5004	5,887058824	1,987291501
881	1690	2571	5070	5,754824064	1,971995333
888	1772	2660	5316	5,986486486	1,998496241
904	1724	2628	5172	5,721238938	1,96803653
950	864	1814	2592	2,728421053	1,428886439
976	968	1944	2904	2,975409836	1,49382716
1080	1012	2092	3036	2,811111111	1,45124283
1184	1372	2556	4116	3,476351351	1,610328638
1200	1979	3179	5937	4,9475	1,867568418
1220	2406	3626	7218	5,916393443	1,990623276
1243	1070	2313	3210	2,582461786	1,387808042
1254	1208	2462	3624	2,889952153	1,471974005
1413	1654	3067	4962	3,511677282	1,617867623
1464	1366	2830	4098	2,799180328	1,448056537
1540	2247	3787	6741	4,377272727	1,780036969
1598	3136	4734	9408	5,887359199	1,987325729
1599	1535	3134	4605	2,879924953	1,46936822
1606	3150	4756	9450	5,884184309	1,986963835
1614	3216	4830	9648	5,977695167	1,997515528
1681	2254	3935	6762	4,022605592	1,718424396

1786	3564	5350	10692	5,98656215	1,998504673
1941	2638	4579	7914	4,077279753	1,728324962
2080	1650	3730	4950	2,379807692	1,327077748
2121	1194	3315	3582	1,688826025	1,080542986
2185	2776	4961	8328	3,811441648	1,678693812
2194	4348	6542	13044	5,945305378	1,993885662
2220	4428	6648	13284	5,983783784	1,998194946
2406	4800	7206	14400	5,985037406	1,998334721
2416	4776	7192	14328	5,930463576	1,992213571
2646	5148	7794	15444	5,836734694	1,981524249
2757	4725	7482	14175	5,141458107	1,894546913
3093	5908	9001	17724	5,730358875	1,969114543
3288	2600	5888	7800	2,372262774	1,324728261
3300	6580	9880	19740	5,981818182	1,997975709
3526	7032	10558	21096	5,982983551	1,998105702
3664	9307	12971	27921	7,620360262	2,15257112
3768	7461	11229	22383	5,940286624	1,993320866
3900	5924	9824	17772	4,556923077	1,809039088
3974	7392	11366	22176	5,580271766	1,951082175
4536	8478	13014	25434	5,607142857	1,954356846
4572	7112	11684	21336	4,666666667	1,826086957
4860	8146	13006	24438	5,028395062	1,878978933
5137	6741	11878	20223	3,936733502	1,702559353
5254	10234	15488	30702	5,843547773	1,982308884
5534	9588	15122	28764	5,197687026	1,902129348
5535	9708	15243	29124	5,261788618	1,91064751
5984	11165	17149	33495	5,597426471	1,953175112
6306	10474	16780	31422	4,982873454	1,872586412
6646	7210	13856	21630	3,254589227	1,561056582
6732	7199	13931	21597	3,208110517	1,55028354
6969	7419	14388	22257	3,193715024	1,546914095
6978	13720	20698	41160	5,898538263	1,988597932
7259	12831	20090	38493	5,302796528	1,916027875
7337	7646	14983	22938	3,126345918	1,53093506
7337	8964	16301	26892	3,66525828	1,649714741
7424	14132	21556	42396	5,710668103	1,96678419
8071	16031	24102	48093	5,958741172	1,995394573
8412	16707	25119	50121	5,958273894	1,995342171
8567	12822	21389	38466	4,490019844	1,798401047
8606	11366	19972	34098	3,962119452	1,707290206
9130	18209	27339	54627	5,983242059	1,998134533
9374	18514	27888	55542	5,925112012	1,991609294
9424	10960	20384	32880	3,488964346	1,613029827
10086	16802	26888	50406	4,997620464	1,874665278
10429	14470	24899	43410	4,162431681	1,743443512
11220	12840	24060	38520	3,43315508	1,600997506
11541	17541	29082	52623	4,559656875	1,809469775
11698	14628	26326	43884	3,751410498	1,666945225
14554	28792	43346	86376	5,934863268	1,992709823
17132	16152	33284	48456	2,828391314	1,455834635
18096	11098	29194	33294	1,839854111	1,140439816
18418	17436	35854	52308	2,840047779	1,458916718
20496	33264	53760	99792	4,868852459	1,85625
22801	45000	67801	135000	5,920792948	1,991121075
24056	23892	47948	71676	2,979547722	1,494869442
26337	51544	77881	154632	5,871283745	1,985490685
39240	67891	107131	203673	5,190443425	1,901158395
44320	49032	93352	147096	3,318953069	1,575713429

Bibliografie

[3DS] Autodesc website. `http://www.discreet.com/`.

[ATI] Ati developer website. `http://www.ati.com/developer/`.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[Bro] Brookgpu website. `http://graphics.stanford.edu/projects/brookgpu/`.

[Chr05] Martin Christen. Ray tracing on gpu. Master's thesis, University of Applied Sciences Basel, Switzerland, 2005.

[DBB03] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*. A K Peters, Natick, USA, 2003.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. 2005.

[gpg] Gpgpu website. `http://www.gpgpu.org/`.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[Jen01] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[KL04] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers University of Technology, Sweden, 2004.

[lib] lib3ds website. `http://lib3ds.sourceforge.net/`.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *j-J-GRAPHICS-TOOLS*, 2(1):21–28, 1997.

[nVi] nvidia developer website. `http://developer.nvidia.com/`.

[Ope] Opengl website. `http://www.opengl.org/`.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[Pur] Timothy John Purcell. *Ray Tracing on a Stream Processor*. PhD thesis.

[Sh] Sh website. `http://libsh.org/`.

[sha] Microsoft website. `http://www.microsoft.com/whdc/winhec/partners/shadermodel30\_NVIDIA.mspx`.

[Shi00] Peter Shirley. *Realistic ray tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2000.

[Tro] Trolltech website. `http://www.trolltech.com/`.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.

[WSB01] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In S.J.Gortler and K.Myszkowski, editors, *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering)*, pages 277–288. Springer, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.