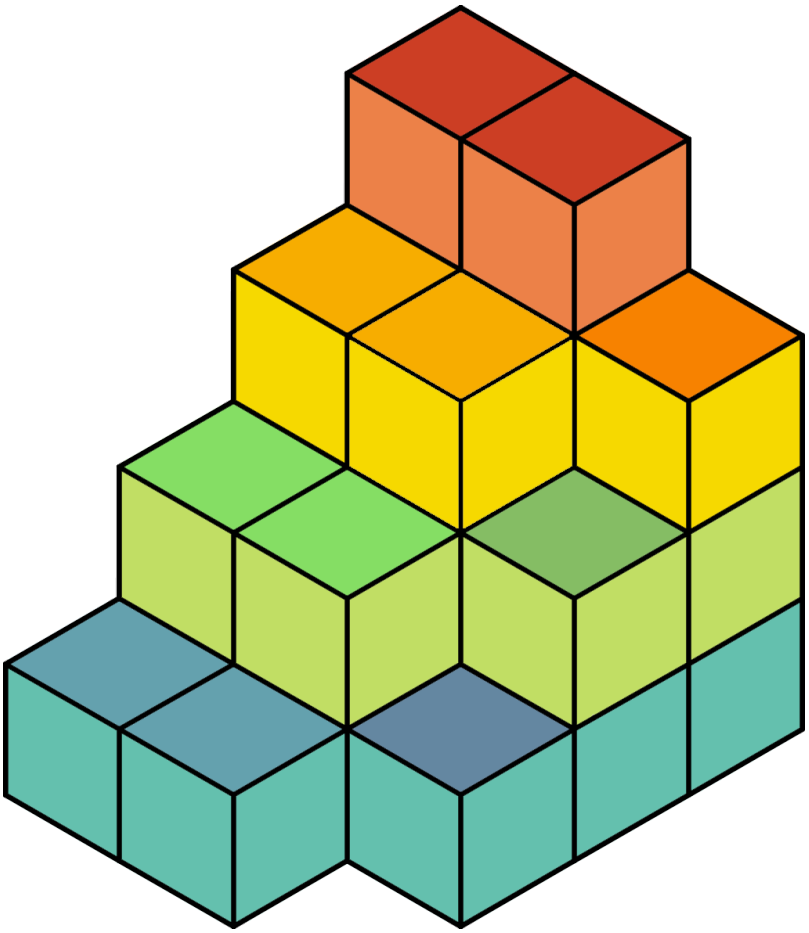
Voxel Pad Reductie

Stein Keijzers

Studentnummer: 3004856

Begeleider: Pieter Koopman

25 juni 2012



Inhoudsopgave

1 Abstract

Medische testresultaten van driedimensionele aard, vaak opgeslagen in 3D-pixels genaamd *voxels*, kunnen gevisualizeerd worden om de evaluatie door professionals te bevorderen. Als deze resultaten sneller op een 2D-scherm getekend kunnen worden zullen ze ook sneller worden geëvalueerd, waardoor er minder tijd verloren raakt en mensen sneller geholpen kunnen worden. In deze scriptie wordt een ontwerp van een algoritme gegeven om een 3D-model bestaande uit voxels te converteren naar een set van *triange strips*; een datastructuur, gebaseerd op een pad, die snel afzonderlijk getekend kan worden. Dit algoritme is geëvalueert door middel van tests op het gebied van geheugengebruik en rendertijd per frame. De tests wijzen uit dat het geheugengebruik met behulp van het geïntroduceerde algoritme sterk daalt. Echter, de rendertijd per frame wordt sterk verhoogd doordat de communicatie met de hardware te veel tijd kost. Hierdoor is het algoritme alleen geschikt in specifieke gevallen.

2 Inleiding

Veel scanresultaten in medische velden, zoals bijvoorbeeld van een fMRI-scan, hebben een driedimensionele aard en worden opgeslagen via een 3D-versie van pixels genaamd *voxels*. Een "afbeelding" van voxels wordt dan ook een *voxel-model* genoemd. Voor evaluatie van deze testresultaten is het nuttig om deze resultaten ook in 3D te kunnen beschouwen, zodat er sneller conclusies uit kunnen worden getrokken dan wanneer ze stap-voor-stap bekeken moeten worden. Hiervoor zijn technieken nodig om deze gegevens, op een efficiënte manier, in 3D te visualizeren. Het doel van dit onderzoek is om een algoritme te ontwerpen, testen en evalueren dat 3D-testresultaten sneller op het scherm kan tekenen. Het idee is om dit te bereiken door minder data naar de grafische kaart te sturen, zodat de hardware hier beter mee om kan gaan. Dit wordt gedaan door het voxel-model te reduceren naar een set van samenhangende stukken: zogenaamde *triangle strips*, datastructuren die afzonderlijk efficiënt gerendered kunnen worden door grafische hardware en minder geheugen in beslag nemen.

Het beschouwen van medische drie-dimensionale testresultaten is belangrijk om kwalen of onderzoeksinformatie te kunnen vaststellen. Behalve 3D-visualisatie zijn er ook andere manieren om dit te doen, zoals door bijvoorbeeld langs alle *slices*, dit zijn individuele lagen van een model, van de resultaten heen te lopen om te zoeken naar relevante zaken. Voordelen van 3D-visualisatie zijn dat de gehele testresultaten meteen weergeven worden en dat er op een meer natuurlijke wijze naar de resultaten gekeken kan worden. Door deze 3D-visualisatie efficiënter te maken kan er waardevolle tijd bespaard blijven, waardoor bijvoorbeeld mensen sneller geholpen kunnen worden.

Het algoritme ontworpen in dit onderzoek gebruikt triangle-strips om een voxel-model om te zetten in data die de grafische hardware van een computer begrijpt. Deze hardware rekent vervolgens uit hoe al deze gegevens op het tweedimensionale computerscherm gezet moeten worden. Door het gebruik van triangle strips zal er hopelijk minder data naar de hardware verzonden moeten worden ten opzichte van algoritmes die geen triangle strips gebruiken, terwijl het resultaat dat op het scherm verschijnt geen kwaliteitsverlies ondervindt.

Om dit onderzoek te voltooien worden de volgende vier stappen uitgevoerd en uitgewerkt:

1. Een kort literatuuronderzoek om bekende technieken te inventariseren.
2. Een beschrijving van een algoritme dat 3D-testresultaten, als een voxel-model, reduceert naar een set van triangle-strips.
3. Aantonen dat dit algoritme hetzelfde render-resultaat oplevert in vergelijking met een basis-algoritme dat geen triangle-strips gebruikt.
4. Onderzoek naar de prestaties van het ontworpen algoritme op het gebied van geheugengebruik van de uiteindelijke data en de tijd die het de hardware kost om het op een scherm te tekeken, in vergelijking tot het basis-algoritme uit stap 3.

Het basis-algoritme, vermeld in stap drie, is de meest simpele vorm van 3D-visualisatie die een volledig resultaat geeft. Dat wil zeggen: het basis-algoritme rendert precies alle zichtbare onderdelen van een voxel-model. Dit wordt gedaan door simpelweg alle zichtbare stukken te verzamelen en om te zetten in gegevens compatibel met grafische hardware. Door een vergelijking van dit basis-algoritme met het in het onderzoek geintroduceerde algoritme kunnen er kwantificeerbare testresultaten behaald worden.

Voor 3D-rendering zijn er veel manieren om het resultaat nog wat te verfraaien, maar dit onderzoek houdt zich daar niet mee bezig. Hier betreft het enkel de basis van 3D-renderen.

3 3D-Rendering

3D-renderen op een computer gebeurt via een aantal stappen. Een programmeur wil graag iets driedimensioneels op het scherm tekenen. Om dit te bereiken moet er eerst precies gedefinieerd worden wat er op het scherm moet komen. Dit is aan de programmeur om op te lossen. Als deze gegevens eenmaal bekend zijn, kunnen ze doorgegeven worden aan een geselecteerde API. Deze API zorgt dat alle data doorgegeven wordt aan het Operating System van de computer. Het OS zorgt dan uiteindelijk dat alles op de hardware komt, waarna het op het scherm getekend kan worden.

De ontwikkelaar van een programma dat gebruik wil maken van 3D-mogelijkheden kiest dus vaak een API en gebruikt dan de mogelijkheden die geleverd worden door dit softwarepakket. De meestgebruikte paketten hiervoor zijn OpenGL en Direct3D. OpenGL is open-source en werkt op vrijwel alle operating systems, terwijl Direct3D door Microsoft ontwikkeld en beheerd wordt en dus standaard alleen op Windows draait. OpenGL en Direct3D zijn qua performance tegenwoordig vrijwel identiek, en de meeste hardwarefabrikanten bieden support voor beide.

Voor dit onderzoek wordt gebruik gemaakt van Direct3D voor de implementaties van de algoritmes. Het onderzoek is echter toepasbaar op beide paketten vanwege de sterk vergelijkbare rendermethodes. De keuze is in dit geval dan ook vanwege bekendheid met implementatiemethodes en niet vanwege efficientie of andere redenen.

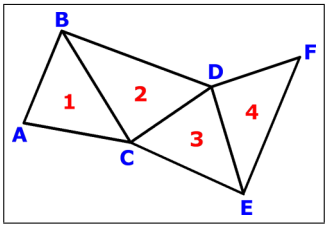
Bij het 3D-renderen worden voornamelijk vertices gebruikt. Dit zijn punten in 3D-ruimte (met x, y en z-coordinaten) waarbij mogelijk nog wat extra informatie, zoals kleur, kan worden opgeslagen. Het tekenen in 3D wordt gedaan via driehoeken, elk bestaande uit drie vertices.

Omdat een model vrijwel nooit van binnen bekeken hoeft te worden wordt er gebruik gemaakt van een techniek die *culling* heet om het efficienter te maken. Deze techniek zorgt ervoor dat alleen driehoeken die met de klok mee richting de gebruiker staan worden getekend; dit heeft als gevolg dat ongeveer de helft van alle driehoeken niet getekent moet worden, iets wat dus een groot voordeel biedt.

Zoals in de inleiding al kort vermeld werd zijn er vele grafische toeters en bellen die toegepast kunnen worden bij het renderen, maar deze zijn niet belangrijk voor medische of andere serieuze toepassingen, omdat functionaliteit daar belangrijker is. In dit onderzoek gaat het enkel over normale driehoeken bestaande uit vertices en hoe deze beter opgeslagen of getekend kunnen worden.

Er zijn verder nog meerdere methodes om de vertices te tekenen. De methode die wordt gebruikt voor het in dit onderzoek geïntroduceerde algoritme werkt via *triangle strips*. Dit is een datastructuur bestaande uit een reeks vertices. Bij het tekenen van een triangle strip (ook wel afgekort als *tristrip*) is de volgorde van de vertices belangrijk. De eerste driehoek wordt getekend via de eerste drie vertices in de tristrip. De tweede driehoek met de tweede tot en met de vierde vertices, enzovoort. Op deze manier wordt er veel geheugen bespaart.

Dit plaatje illustreert hoe een tristrip wordt gebruikt om een stel driehoeken te tekenen:



De tristrip in dit plaatje is $[A, B, C, D, E, F]$. Driehoek 1 wordt beschreven door de vertices $[A, B, C]$, driehoek 2 door $[B, C, D]$, etc. Belangrijk om op te merken is dat de eerste driehoek met de klok mee wordt getekend en de tweede tegen de klok in. Omdat culling wordt toegepast op een van de twee richtingen zal de tweede driehoek dus niet getekend worden. Om dit te voorkomen wordt de culling-richting door Direct3D impliciet omgedraaid na elke driehoek.

Een nadeel van tristips is echter dat er voor elke tristrip een functie moet worden aangeroepen die het tekent via de API, terwijl het met normale vertices in een keer kan. Deze API-functie moet de hardware aanroepen en zorgt dus voor meer rekentijd. Als het aantal paden erg groot wordt kan dit een negatief effect kan hebben op de rendertijd per frame.

Een andere methode om vertices te tekenen maakt gebruik van een lijst vertices en een lijst indices. Het basis-algoritme van dit onderzoek gebruikt deze methode. De lijst van *indices* bevat index-nummers van individuele vertices in de andere lijst. Om een driehoek te tekenen worden er drie indices gelezen, waarna de corresponderende vertices uit de lijst opgeslagen worden. Deze drie vertices worden dan als een driehoek getekend. Het voordeel van deze methode is dat een vertice meerdere keren gebruikt kan worden, zonder dat hij extra ruimte in beslag neemt, door de corresponderende index meerdere keren te gebruiken.

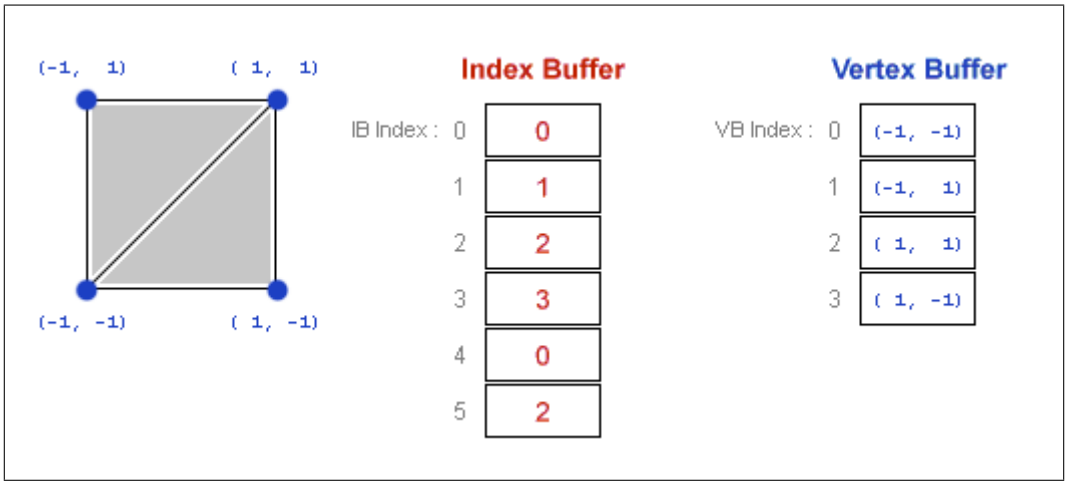
3.1 Voxels

Een voxel is in principe een 3D-pixel. Een "plaatje" van voxels wordt dan ook een model genoemd. Hierbij is de positie van elke voxel bepaald door de structuur van het model in plaats van locatieinformatie in de voxel zelf. Zo'n model heeft een bepaalde resolutie en een grootte in de x, y en z-assen. De simpelste voxel heeft niet meer dan een waarde Ja of Nee in zich, die aangeeft of de voxel wel of niet gevuld is. In de medische wetenschap, zoals bij het resultaat van een fMRI scan, heeft een voxel een bepaalde intensiteit die bijvoorbeeld de hersenactiviteit in dat blok van de hersenen weergeeft. In een andere context kunnen er ook kleurwaardes of rotatiewaardes in worden opgeslagen, alsmede andere dingen afhankelijk van de toepassing.

Bij het renderen van voxels zijn er enkele optimalizaties die standaard worden toegepast. Zo is het natuurlijk niet slim om dingen te tekenen die nooit door de gebruiker gezien zullen worden. Voxels die zich binnenin een model bevinden kunnen dus worden overgeslagen. Zijdes van een voxel die naar de binnenkant gericht zijn hoeven ook niet getekend te worden omdat deze bedekt worden door een andere voxel. Uiteindelijk blijven dan alleen de zijdes van de buitenste voxels die getekend moeten worden over, oftewel de zichtbare zijdes van het model.

Voor het in de inleiding vermeldde basisgeval wordt elke zijde getekend door middel van 4 vertices en een lijstje met 6 indices. Er zijn namelijk twee vertices die twee keer gebruikt kunnen worden, waardoor er in totaal maar vier vertices opgeslagen moeten worden. Het doel van het algoritme beschreven in dit onderzoek is om hetzelfde tekenresultaat als het basisgeval te verkrijgen, maar dan zonder indices en met minder vertices.

Dit plaatje laat zien hoe een zijde van een voxel als vertices en indices opgeslagen kan worden:



De driehoek linksboven in het vierkant wordt beschreven door de indices $[0, 1, 2]$, terwijl de driehoek rechtsonder bestaat uit de indices $[3, 0, 2]$. Op deze manier hoeven er maar vier vertices opgeslagen te worden, in plaats van zes.

4 Gerelateerd Onderzoek

Op het gebied van volume rendering (waaronder voxels) wordt nog steeds veel onderzoek verricht, meer dan mogelijk is om allemaal te bespreken. Daarom volgt hieronder een korte beschrijven van gerelateerd werk en algemene onderwerpen.

Een artikel geschreven door Subramanian et al. in 1990[1] detailleert een methode om een voxel-model efficiënt op te slaan. Hiervoor wordt gebruik gemaakt van een k-d tree, een binaire zoekboom voor k-dimensionele data, om voxels goed verdeeld in 3D-ruimte op te slaan. Deze datastructuur zorgt ervoor dat er sneller informatie over bepaalde voxels kan worden verzameld, wat de render-tijd verbetert. Latere onderzoeken maken gebruik van octrees, een vergelijkbare datastructuur die een ruimte in 3D in acht delen opsplitst, die een nog compactere opslag bieden.[2] Voor dit onderzoek kan deze techniek ook toegepast worden op het resulterende set van paden, maar omdat het betreffende set sequentieel doorlopen wordt voor het renderen, biedt het geen extra mogelijkheden of verbeteringen. Ook is de kans erg groot dat bepaalde paden worden doorbroken door het partitioneren.

Een meer directe link kan worden gelegt tussen dit onderzoek en een artikel geschreven door Grossman et al. over de oppervlakte van een model.[3] Het doel van het algoritme beschreven in dit onderzoek is namelijk om een set van paden te verkrijgen dat precies over het oppervlakte van een voxel-model loopt, om het zo te kunnen renderen. Grossman beschrijft een methode om een 3D-model om te zetten in een 2D representatie, waarbij de afstanden tussen punten op het model zoveel mogelijk behouden blijven. Dit is echter moeilijk toepasbaar op dit onderzoek, omdat het belangrijk is dat verbindingen tussen voxels op het platte vlak niet verbroken worden, zoals bijvoorbeeld het geval is bij het uitrollen van een cilinder. Als er bijvoorbeeld een toren in een voxel-model bestaat, en deze toren vervolgens platgemaakt wordt door de methode van Grossman et al, raakt er een boel 3D-informatie verloren. Vanwege deze eigenschap kan de 2D-representatie niet terug worden vertaald naar 3D-informatie die bruikbaar is voor het algoritme besproken in deze scriptie.

Recent onderzoek in dit gebied wordt gedaan over een renderings-methode genaamd raytracing, zoals de GigaVoxels methode beschreven door Crassin et al.[4] Het onderzoek is vooral geconcentreerd op methodes om grote stukken van een voxel-model te abstraheren of buiten beschouwing te laten, wat de efficientie (vooral op het gebied van geheugen) ten goede komt. Een groep voxels die van grote afstand beschouwd wordt hoeft immers niet volledig gedetailleerd getekend te worden. Voor dit onderzoek kan dit toegepast worden op het uiteindelijke tekenresultaat. Echter, dit gaat verder dan het basisgeval, en is dus een stap verder dan waar deze scriptie zich mee bezighoudt.

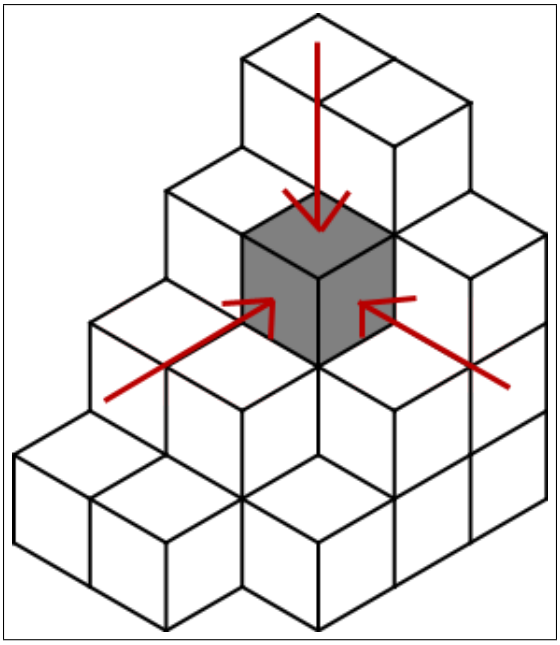
Naast deze onderwerpen is interactiviteit voor voxel-modellen een veelbezocht punt[5][6]: het gaat hier dan voornamelijk om het real-time updaten van voxel-modellen, bijvoorbeeld om een MRI-scan doormidden te snijden zodat beter naar de binnenkant van iets kan worden gekeken. Voor het algoritme besproken in dit artikel zal het nodig zijn om het opnieuw uit te voeren als het model is veranderd, omdat de paden sterk van het volledige model afhangen en omdat ze niet zomaar doorbroken kunnen worden zonder mogelijke problemen. Verscheidende interactiviteitstechnieken zijn hierdoor niet bruikbaar. Een model in twee delen splitsen kan echter wel snel omdat verbindingen op een bepaalde doorsnede snel gevonden en doorbroken kunnen worden.

5 Basis Algoritme

5.1 Beschrijving

Het basisgeval dat in dit onderzoek gebruikt wordt is een simpel algoritme dat niks "slims"doet en gewoon domweg alle zichtbare dingen omzet naar tekenbare vertices en indices. Hiervoor worden alle voxels in een model die boven een bepaalde intensiteitsgrens liggen afzonderlijk beschouwd. Voor elke zijde van een voxel wordt gekeken of deze zichtbaar is of wordt bedekt door een andere voxel. Als hij zichtbaar is wordt deze zijde geconverteerd en toegevoegd aan de vertices en indices die getekend moeten worden.

Dit plaatje illustreert hoe elke voxel afzonderlijk bekeken wordt:



Het algoritme in pseudocode:

Algorithm 1 Basisgeval

Require: $model \wedge (vertices, indices = [])$
Ensure: $vertices, indices = \text{visible}(model)$

```
for all Voxel v in model do
  if v ≥ threshold then
    for all Face f in faces(v) do
      Voxel a ← adjacent(v, f)
      if a ≤ threshold then
        vertices ← vertices + corners(f)
        indices ← indices + indexnums(f)
      end if
    end for
  end if
end for
```

Functies die hier worden gebruikt:

Functie	Werking
$visible(model)$	Alle zichtbare faces van *model*. Alleen gebruikt als definite in de post-conditie.
$faces(v)$	Alle faces van voxel v. Dit zijn er altijd zes omdat een voxel een kubus is.
$adjacent(v, f)$	De voxel die zich naast v bevindt, bij face f.
$corners(f)$	Levert de vier vertices op de hoekpunten van f op.
$indexnums(f)$	Levert de zes indices op die gebruikt kunnen worden om de twee driehoeken van f te tekenen.

5.2 Complexiteit

Het voxel-model is opgeslagen in een 3-dimensionale array. In C++ kunnen elementen uit een array in constante tijd geselecteerd worden, waardoor het bepalen van aangesloten voxels in constante tijd gebeurt. Het algoritme beschouwt, in het slechtste geval waar het hele model boven de threshold zit, elk face van elke voxel. Hierdoor is de rekentijd maximaal $(C + 6)N$, waardoor de bovengrens in $O(N)$ ligt. Hierbij is C een of andere constante en N het aantal voxels in het model. Het algoritme is dus lineair in N.

6 Toren Algoritme

6.1 Beschrijving

Dit is het algoritme dat ontworpen is en getest wordt voor deze scriptie. Het doel van dit algoritme is om een gegeven voxel-model om te zetten in een set van paden. Deze paden kunnen daarna sequentieel op het scherm gezet worden. Een enkel pad is een zogenaamde *tristrip*, wat staat voor *triange strip*. Zo'n strip is een lijst vertices, waaruit de eerste driehoek getekend wordt via de eerste drie vertices. Daarna wordt elke opvolgende vertice gebruikt met de vorige twee om de volgende driehoek te tekenen, zoals in detail besproken in de sectie *3D-Rendering* hierboven.

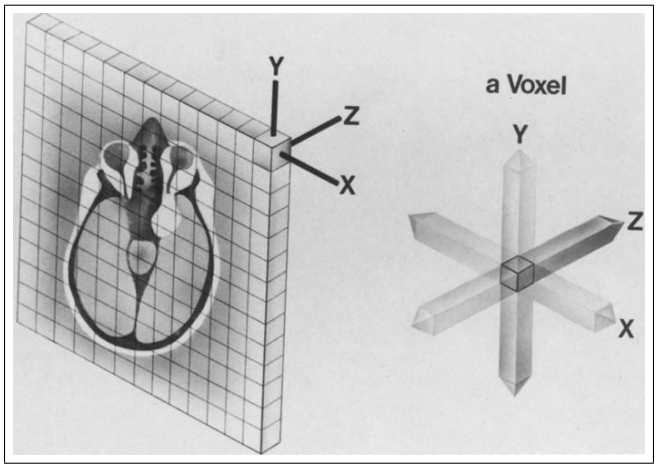
Een optimale conversie van een willekeurig 3D-model naar tristrips is NP-Compleet bewezen[11]. Vanwege de structuur van een voxel-model valt hier echter een boel aan te verbeteren. De grootte van elke voxel is immers constant, dus er is geen verschil in dichtheid van driehoeken tussen twee gebieden, in tegenstelling tot andere 3D-modellen. Een nadeel is wel dat een tristrip moeilijk bochten kan maken over een veld van vierkanten. Als een pad rechtdoor loopt en dan een bocht moet maken, zal het hiervoor de volgorde van vertices iets moeten veranderen om te zorgen dat er geen hoeken overgeslagen worden. Verder hangt de richting waarin zo'n bocht gemaakt kan worden af van de volgorde van de allereerste twee vertices in de tristrip, omdat anders de volgorde van de huidige vertices niet goed verandert kan worden. Het moet dus een set van rechte paden worden, die nog wel om iets heen gewikkeld kunnen worden.

Om in dit algoritme het voxel-model te converteren wordt er iteratief door de *slices* van het model gelopen. Een slice is een enkel vel van het model, in de hoogte-richting. Het gehele model bestaat dus uit een hoeveelheid slices die op elkaar gestapeld worden. Elke slice heeft

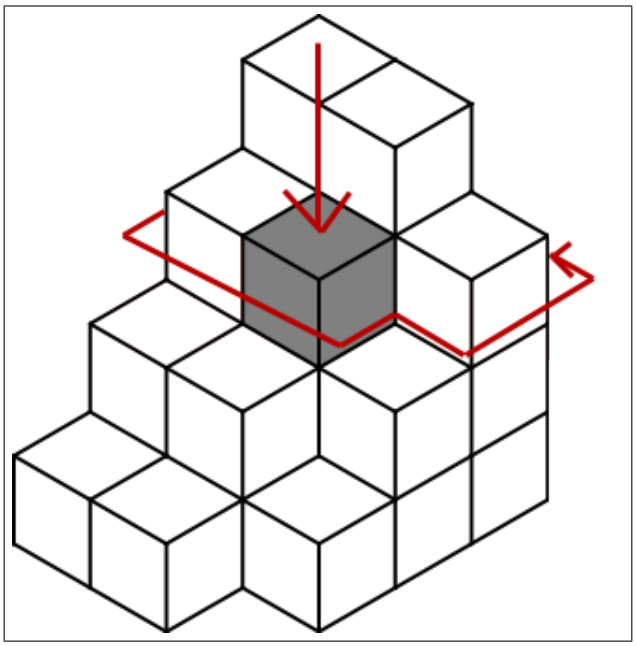
vaak meerdere groepen voxels, waardoor er clusters gevormd worden. Om elk van die clusters kan precies een tristrip gewikkeld worden om de zijkant te tekenen. Dan blijven er enkel nog mogelijke randen over: als een toren in het model in de hoogte smaller of breder wordt komen er meer zijdes van voxels bloot te liggen. Dit kan gedetecteerd worden door naar de vorige slice te kijken, waarna die bepaalde plaats aangemerkt kan worden om ook de boven- en onderkanten van de desbetreffende voxels te controleren.

Het controleren van de boven- en onderkanten van de slices gebeurt door per rij voxels in de slice te kijken welke delen nog bloot liggen en deze vervolgens in een pad om te zetten.

Dit plaatje laat zien hoe een slice eruit ziet:



En de volgende afbeelding laat zien hoe dit gebeurt met meerdere slices in het algoritme:



Het algoritme in pseudocode:

Algorithm 2 Toren Algoritme

Require: $model \wedge (paths = \emptyset) \wedge threshold$
Ensure: $paths = \text{visible}(model)$
  Slice $prev \leftarrow \perp$
  List $check \leftarrow new$ Array$[ysize(model)]$ of Boolean
  for all Slice $z \in model$ **do**
    for all Boolean b in $check$ **do**
      $e \leftarrow false$
    end for
    $paths \leftarrow paths \cup \text{circumferences}(z, threshold)$
    if $prev = \perp$ **then**
      for all Row $r \in z$ **do**
        $check[r] \leftarrow true$
      end for
    else
      for all Row $r \in \text{discrepancies}(prev, z)$ **do**
        $check[r] \leftarrow true$
      end for
    end if
    for all Row $r \in z$ **do**
      if $check[r] = true$ **then**
        $paths \leftarrow paths \cup \text{sides}(r, z, threshold)$
      end if
    end for
  end for

Functies die hier worden gebruikt:

Functie	**Werking**
$visible(model)$	Alle zichtbare faces van $model$. Alleen gebruikt als definite in de post-conditie.
$discrepancies(a, b)$	Alle $rows$ waarvan de paden uit a en b niet direct naast elkaar liggen. Dit betekent immers dat er hier zijkanten van voxels zijn die nog bekeken moeten worden.
$sides(row, slice, threshold)$	Een of meerdere paden die de te tekenen zijkanten uit $slice$ op row omvatten.
$ysize(model)$	Geeft de grootte in de y-richting van het model.

Verder wordt de functie *circumferences* in het algoritme gebruikt om alle zichtbare zijkanten van een slice in paden om te zetten. Deze functie heeft de volgende pseudocode:

Algorithm 3 Circumferences

Require: $slice \wedge (result = [])$
Ensure: $result = \text{sides}(slice)$
  List $visited \leftarrow new$ Array of Boolean $false$
  for all Voxel v in $slice$ **do**
    if $v \geq threshold \wedge \neg visited[v]$ **then**
      $path \leftarrow []$
      $done \leftarrow false$
      while $\neg done$ **do**
        $visited[v] \leftarrow true$
        $path \leftarrow path + \text{currentpath}(\text{faces}(v))$
        if hasnext(v) **then**
          $v \leftarrow \text{next}(v)$
        else
          $done \leftarrow true$
        end if
      end while
      $result \leftarrow result + path$
    end if
  end for
  return $result$

Functies die in deze functie worden gebruikt:

Functie	**Werking**
$sides(slice)$	Alle zichtbare zijdes van *slice*. Alleen gebruikt als definite in de post-conditie.
$currentpath(faces)$	Bepaalt de faces die bij het huidige pad horen door over het oppervlak te lopen.
$faces(voxel)$	Alle faces van *voxel*.
$hasnext(voxel)$	Functie die bepaald of *voxel* een volgende heeft voor het huidige pad.
$next(voxel)$	Functie die deze volgende ophaalt, aangenomen dat hij bestaat voor *voxel*.

6.2 Complexiteit

In dit algoritme wordt elke slice precies een enkele keer behandeld, waarbij de pilaren gevonden en beschouwd worden. In het slechste geval moeten alle zijkanten nog een keer bekeken worden, hiervoor moeten dus alle voxels behandeld worden. Voor de randen is dus $2N$ tijd nodig.

Voor het bepalen van de pilaren is het nodig om de hele slice een keer te doorlopen. Het bepalen van de paden voor elke pilaar kost hierna ook nog, in het slechtste geval, $2N$ stappen. Voor het hele algoritme is dus $5N$ tijd nodig, dus is de complexiteit lineair, $O(N)$.

6.3 Hybride Versie

Als illustratie van de verschillen tussen het basis-algoritme en het toren-algoritme is er een mogelijkheid voor een simpele hybride versie. Als paden erg kort zijn en dus de winst op het gebied van rendertijd laag ligt, kan dit pad nog worden omgezet in vertices en indices voor het basisgeval. Op deze manier kan voorkomen worden dat de hardware het te druk krijgt met het tekenen van vele kleine paden.

Om dit voor elkaar te krijgen is er, na het uitvoeren van het toren-algoritme, nog een extra stap die alle paden met een lengte onder een instelbare grens samenvoegt tot normale vertices en indices zoals het basisgeval die produceert. De focus van het onderzoek ligt op een meer direct verschil tussen het toren-algoritme en het basisgeval dus is deze hybride versie niet het directe onderwerp. Enkele tests zijn nog steeds uitgevoerd, met de resultaten in de relevante sectie, om het verschil tussen het basis-algoritme en het toren-algoritme te illustreren.

7 Equivalentie

Voor het Toren-algoritme is het belangrijk dat het tot een equivalent tekenresultaat als het basisgeval leidt. Dit betekent dat het niet meer en ook niet minder moet tekenen; dus precies hetzelfde aantal driehoeken. Als dit het geval is kan het algoritme onconditioneel vergeleken worden met andere algoritmes die ditzelfde bereiken, zoals het basisgeval.

Om te laten zien dat het Toren-algoritme tot hetzelfde resultaat leidt als het basisgeval wordt er eerst geredeneerd over afzonderlijke faces, waarna het genoeg is om te laten zien dat beide algoritmes precies de zichtbare face van een model beschouwen, en niks meer of minder.

7.1 Faces

In het basisgeval wordt elk face op dezelfde manier behandeld en zijn er geen uitzonderingen. Elk face wordt simpelweg omgezet naar vier vertices, waarna er zes indices worden gemaakt en opgeslagen. Drie indices leveren een enkele driehoek op, waardoor het face getekend wordt.

In het Toren-algoritme worden er geen indices gebruikt. Verder zijn er ook twee gevallen: het face is onderdeel van de zijkant van een pilaar, of het face ligt plat tussen twee lagen. In het eerste geval is het face onderdeel van het pad dat zich op die laag om de pilaar heen wikkelt. Als het face ergens in het midden van dit pad of aan het eind van dit pad ligt zitten de eerste twee vertices al in het pad en worden de andere twee dus toegevoegd om dit face te kunnen tekenen. Als het face echter aan het begin van het pad zit moeten de eerste twee vertices gebruikt worden als beginpunt en dus toegevoegt worden. Hierdoor zitten alle vertices van het face in het pad en wordt het dus uiteindelijk volledig getekent.

In het tweede geval ligt het face dus naar boven of naar beneden gericht, waarbij hij geen onderdeel van de zijkant van een pilaar is. Hierdoor is hij onderdeel van een pad dat over de X-richting loopt om een stuk van de bovenkant/onderkant van een laag te tekenen. Hier is er weer onderscheid tussen de gevallen waar het face aan het begin van de streep ligt en ergens anders. Aan het begin moeten namelijk weer alle vier de vertices toegevoegd worden, maar ergens anders enkel de laatste twee. Verder worden de vertices andersom toegevoegd als het face naar beneden gericht is, zodat het voor de Direct3D culling-techniek de goede kant op wijst.

7.2 Model

Elke voxel wordt afzonderlijk behandeld in het basisgeval. Voor elk face van deze voxel wordt gecontroleerd of hij zichtbaar is, en als dit het geval is wordt hij ook omgezet naar vertices en indices. Het Toren-algoritme beschouwt de voxels echter op basis van clusters in een gegeven laag van het model.

In de eerste stap van het algoritme wordt er door de hele slice heengelopen op zoek naar een oppervlak van het model. Dit kan zowel aan de buitenkant van een groep voxels liggen als aan de binnenkant. Als een oppervlakte gevonden is wordt er een pad gemaakt op dat punt. Dit pad loopt over de zijkant van de cluster totdat hij weer bij hetzelfde punt uitkomt. Hierdoor wordt dus de hele zijkant van de cluster beschouwt. Als er helemaal door de laag heen is gelopen zijn dus alle zijkant-faces van alle voxels toegevoegd aan de lijst met paden om te tekenen.

Na deze stap blijven dus enkel de boven- en onderkanten van de voxels over. Tijdens de eerste stap van het algoritme wordt bijgehouden waar het pad loopt, waarna dit wordt vergeleken met de vorige laag. Als op een bepaalde lijn in de Y-richting de paden niet precies langs elkaar lopen wordt dit Y-coordinaat opgeslagen. De tweede stap van het algoritme beschouwt dan vervolgens alle opgeslagen Y-coordinaten, omdat er zich daar zichtbare boven- of onderkanten van voxels bevinden. Deze beschouwing loopt simpelweg door die lijn van het model heen. Als een zichtbaar vlak tegengekomen wordt dan wordt op dat punt een pad gestart. Zodra wordt gedetecteerd dat dit vlak is afgelopen wordt het pad voltooid en toegevoegd. Op deze manier worden zeker alle zichtbare boven- en onderkanten van de lagen bekeken.

8 Datasets

8.1 Initiele Dataset

De dataset die werd gebruikt voor het ontwikkelen en implementeren van de hierboven gegeven algoritmes is een gesimuleerd MRI testresultaat van menselijke hersenen. Het is gegenereerd met behulp van BrainWeb, via een techniek die beschreven wordt in verscheidende artikelen[7][8][9][10]. Gezien het feit dat dit onderzoek zich concentreert op weergave en niet op radio- of neurologische technieken biedt het gesimuleerde dataset een typisch resultaat, namelijk: normale hersenen.

Elke voxel in de dataset bevat een intensiteitswaarde van 0 tot 255. Omdat voxels geen locatie-informatie bevatten kunnen ze ieder worden opgeslagen in een enkele byte. De dimensies van het voxel-model zijn 181 x 217 x 181, waardoor er in totaal ruim zeven miljoen voxels zijn. Iedere voxel stelt precies 1 kubieke millimeter in de MRI-scan voor.

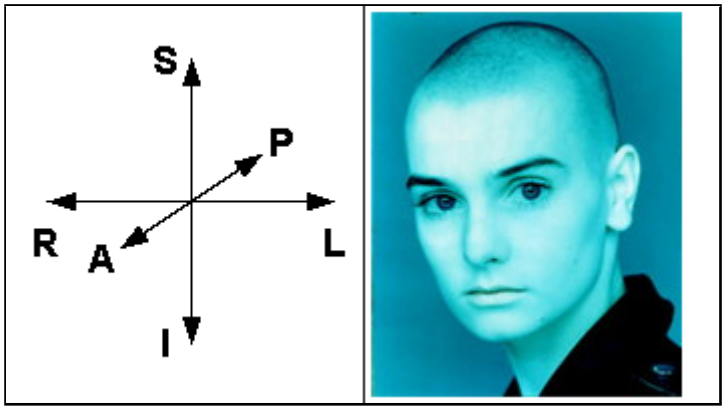
De opslagvolgorde voor de dataset is *(L-R, P-A, I-S)*. Dit betekent dat de X-dimensie in het set omhoog gaat voor de patient van links naar rechts, de Y dimensie van de achterkant van het hoofd van de patient naar de voorkant, en de Z-dimensie van onder de kin tot boven het hoofd[(c)]. Dit is echter niet het geval voor de meeste render-technieken, dus moeten om het gewenste resultaat te behalen deze coordinaten vertaald worden. Dit gebeurt via de volgende converteerregels:

Direct3D	**Dataset**
x	$-x$
y	z
z	$-y$

Deze regels zorgen ervoor dat de uiteindelijke orientatie zoveel mogelijk op de standaard-orientatie lijkt. Dit betekent dat omhoog en omlaag dezelfde kant op gericht zijn en dat de "patient"de camera direct aankijkt. Andere coordinaat-systemen zijn ook mogelijk en hangen van de opslagvolgorde van de dataset af. De verschillende letters, hierboven al kort vermeld, betekenen:

Letter	**Naam**	**Richting**
L	Left	Links van de patient
R	Right	Rechts van de patient
P	Posterior	Achter de patient
A	Anterior	Voor de patient
I	Inferior	Onder de patient
S	Superior	Boven de patient

Hier is een afbeelding van de verschillende assen ten opzichte van een persoon:

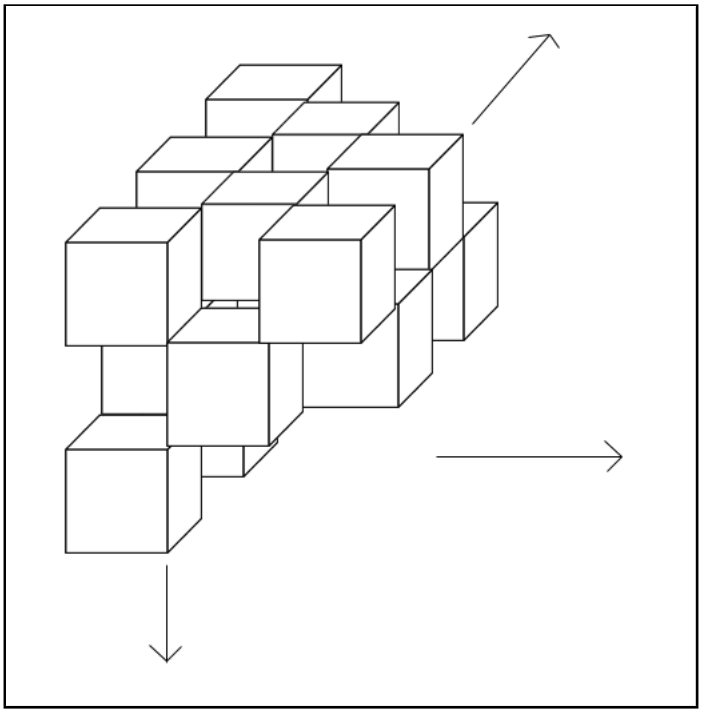


8.2 Test Datasets

8.2.1 Worst-Case

Een dataset voor het slechtste geval is handig voor de performance tests, omdat het aangeeft hoe de algoritmes omgaan met dit de slechtste situaties. Voor het basisgeval is een worst-case voxel model een model met zo veel mogelijk voxel faces om te tekenen. Het maximum wordt bereikt door elke slice van het voxelmodel een dambord-patroon te geven en dit patroon te inverteren op elke volgende slice. Hierdoor moet elk face van elke voxel getekend worden.

Dit dataset is ook het slechtst mogelijke geval voor het Toren-algoritme omdat zowel het aantal pilaren als het aantal paden gemaximalizeerd worden. Hierdoor zal de reduceertijd het langst duren en zal het aantal paden ook hoog liggen.

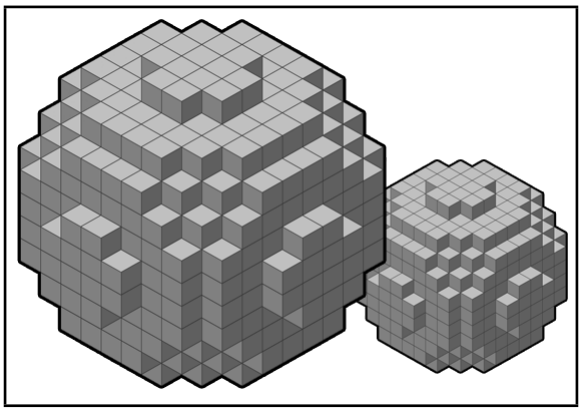


Figuur 1: Het patroon van een worst-case dataset

8.2.2 Bollen

Deze datasets bestaan uit een groep bollen in 3D-ruimte die willekeurig worden gegenereerd met een aantal parameters. Deze parameters zijn de grenzen van het aantal bollen en de grenzen van de grootte van deze bollen. Alle bollen worden willekeurig in het model geplaatst.

Voor de tests zijn deze datasets interessant omdat ze geen torens bevatten en er dus veel op kantjes gecontroleerd moet worden, terwijl het basisgeval hier geen last van heeft. Aan de andere kant is het aantal pilaren in een slice wel beperkt en kan het toren-algoritme dus op dit gebied tijd inwinnen, alsmede een lager aantal paden opleveren.

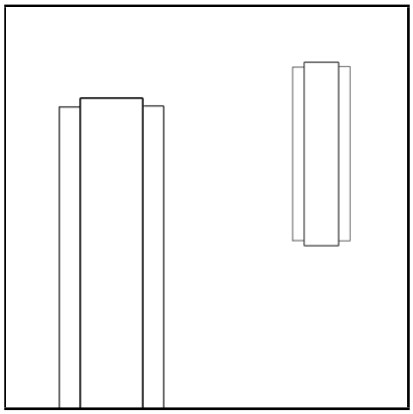


Figuur 2: Een dataset bestaande uit bollen

8.2.3 Pilaren

Het toren-algoritme werkt met pilaren en verschillen hiertussen, terwijl het basisgeval geen onderscheid maakt tussen pilaren en willekeurige voxels. Daarom is een dataset bestaande uit pilaren goed om het algoritme mee te testen, omdat er dus een substantieel verschil merkbaar zou moeten zijn.

Deze datasets worden willekeurig gegenereerd met een instelbaar aantal pilaren en gemiddelde dikte. Verder kunnen er willekeurig kleine variaties in de pilaren worden aangebracht om het toren-algoritme beter te kunnen toetsen. Hierdoor zijn er verschillende modellen mogelijk voor het testen, met een sterk verschillend ruis-percentage.

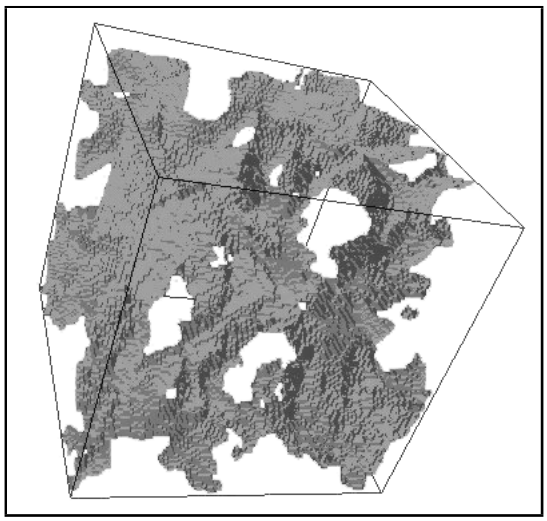


Figuur 3: Een dataset bestaande uit pilaren

8.2.4 Verspreiding

De laatste datasets die gebruikt worden voor het testen zijn modellen met een volledig willekeurige spreiding van voxels, zonder enige structuur hierin. Deze kunnen worden gegenereerd met een bepaalde dichtheid, zodat bijvoorbeeld 20% van het model solide is en de rest lege ruimte.

Dit soort datasets hebben een groot oppervlak vanwege het ontbreken van substantiele groepen in het model. Het is dus een goede manier om de brute kracht van de algoritmes te meten. Verder levert het veel faces op die getekend moeten worden, dus is het ook nuttig voor de render-tests.



Figuur 4: Een compleet willekeurig dataset

9 Testresultaten

9.1 Geheugengebruik

Om de kwaliteit van het toren-algoritme te toetsen is geheugengebruik van groot belang. Het voornaamste voordeel van tristrips ten opzichte van het basisgeval is immers dat elke vertex minder vaak gebruikt hoeft te worden, en dus ook minder vaak in het geheugen opgeslagen moet worden.

Voor de uitvoering van deze tests zijn verscheidende gegenereerde datasets gebruikt, zoals beschreven in de bijbehorende sectie hierboven. De algoritmes zijn losgelaten op de datasets, waarna is gekeken naar hoeveel ruimte de resulterende vertices en indices opnamen, afhankelijk van het type dataset.

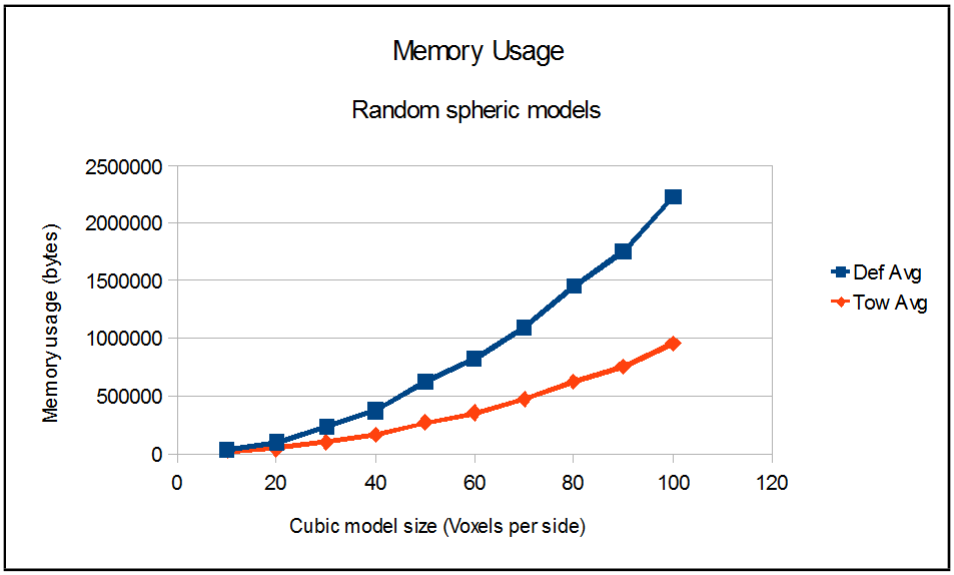
De uit de geheugentests resulterende grafieken hebben op de x-as het aantal voxels per zijde van het model, en op de y-as het totaal aantal bytes dat nodig is voor opslag van het resultaat. Er zijn twee lijnen voor elke grafiek: de lijn voor het basisgeval (Def Avg) en de lijn voor het toren-algoritme (Tow Avg).

De x-as geeft dus het aantal voxels per zijde van het model weer. Alle modellen zijn kubiek gegenereerd. Dit betekent dat het maximum aantal voxels in het model gelijk is aan het aantal voxels per zijde tot de derde macht. Deze as gaat dus niet lineair omhoog.

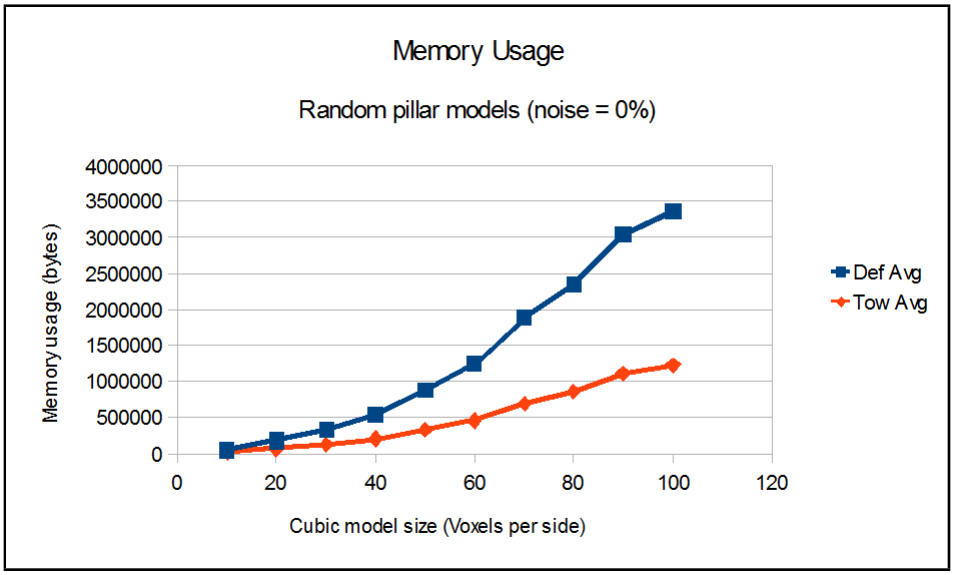
De y-as geeft, zoals vermeld, het geheugengebruik weer. Dit is het totale aantal bytes dat nodig is voor opslag van de vertices en indices, dus dit correspondeert ook met het benodigde videogeheugen van de grafische kaart.

Elk punt op de grafiek is bepaald door tien tests uit te voeren met het desbetreffende algoritme, behalve in de worst-case tests. Dit komt doordat het worst-case altijd hetzelfde is, waardoor er geen extra informatie bemachtigt kan worden door de test opnieuw te draaien. Dit is dus niet het geval voor de willekeurig gegenereerde modellen.

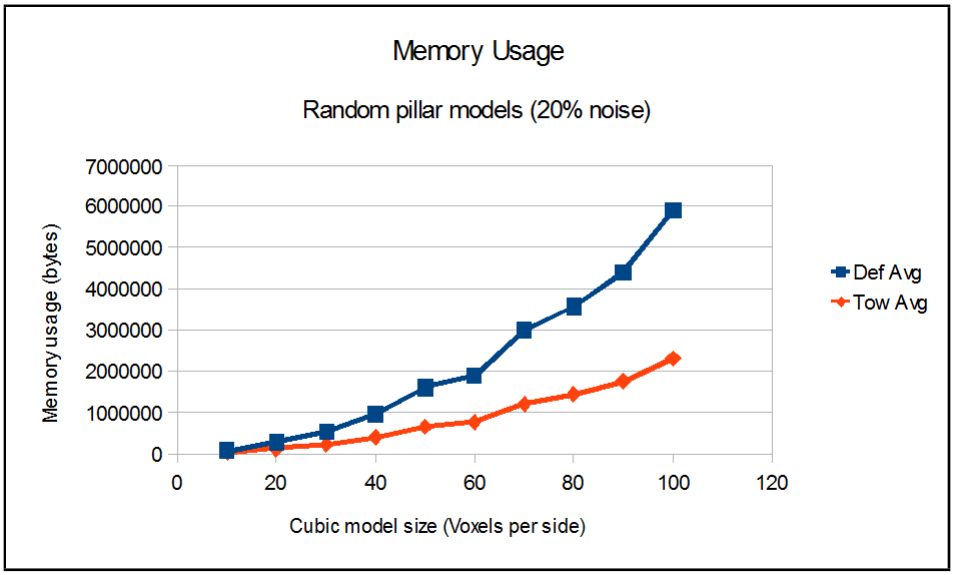
De volgende grafiek is het resultaat van de tests op de datasets bestaande uit bollen:



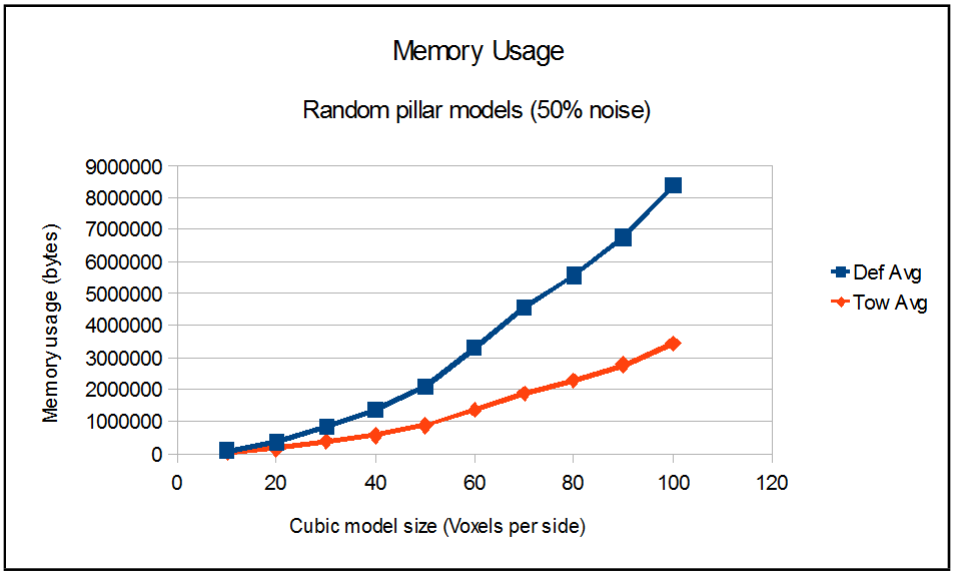
Nu volgen drie grafieken voor de resultaten van de willekeurig gegenereerde pilaar-datasets. Deze eerste grafiek is voor de datasets gegenereerd met 0% ruis:



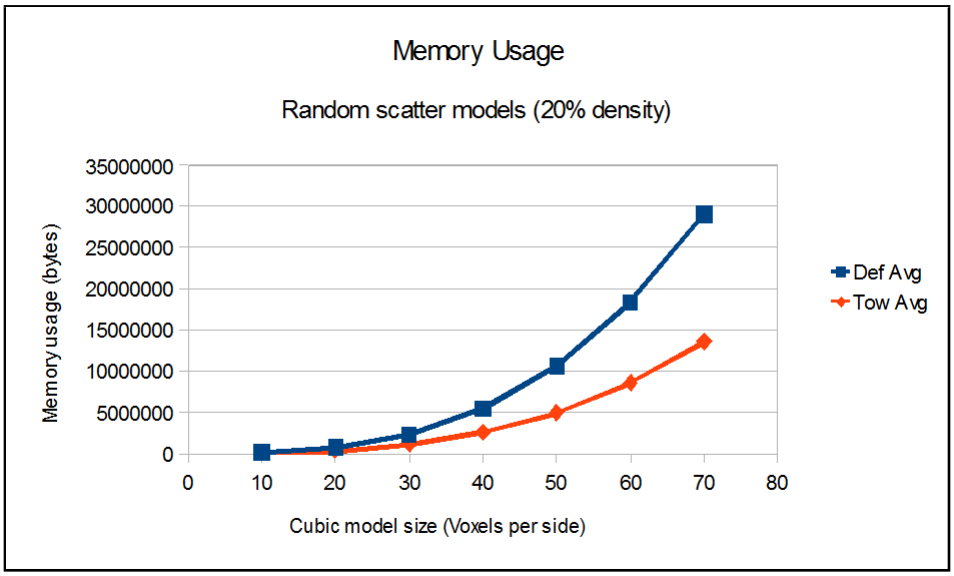
Deze is gegenereerd met 20% ruis:



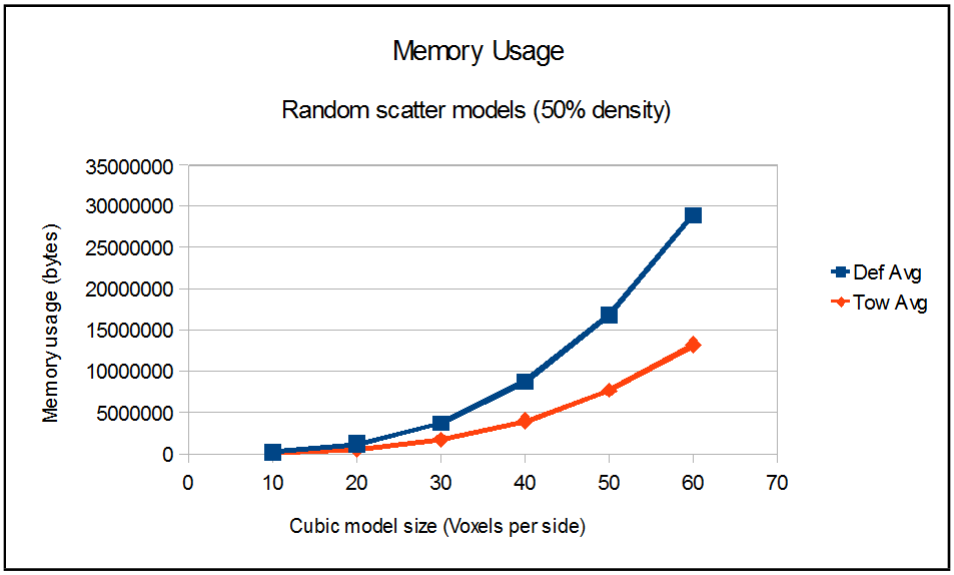
En als laatste de resultaten voor 50% ruis:



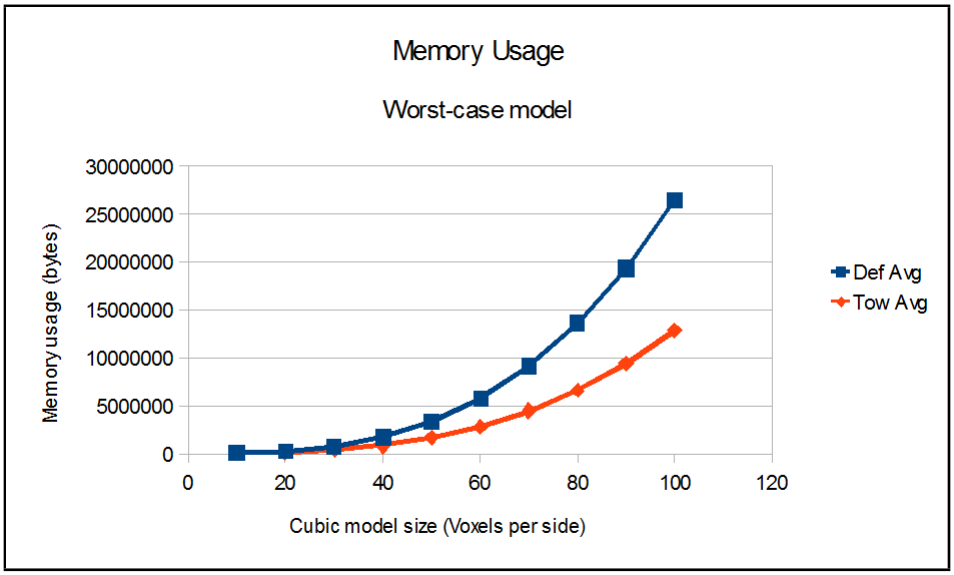
Hier zijn de resultaten voor de verspreidings-datasets met een bepaalde dichtheid. De eerstvolgende grafiek is voor de datasets gegenereerd met een dichtheid van 20%:



En dan de datasets gegenereerd met een dichtheid van 50%:



Als laatste zijn er nog deze resultaten voor het worst-case dataset van variabele grootte:



Uit deze resultaten kan de conclusie worden getrokken dat het toren-algoritme tot een sterk verminderd geheugengebruik leidt dan het basis-algoritme voor vrijwel alle datasets, dankzij de verschillende opslagmethodes voor de resulterende data.

9.2 Rendertijd

Net zo belangrijk als het geheugengebruik voor het toren-algoritme is de rendertijd; dit is de tijd die nodig is om een enkel frame van het model op het scherm te tekenen. Hoe lager dit is, hoe sneller het beeld op het scherm ververst kan worden. Het voordeel van tristrips hierbij is dat de grafische hardware minder moeite hoeft te doen om een enkele tristrip te tekenen. Een nadeel is echter wel dat elke tristrip een nieuwe aanroep moet doen aan de hardwarematige tekenfunctie, waardoor de tijd toch kan oplopen als er veel paden zijn.

Voor deze tests zijn, net zoals bij de vorige, gegenereerde datasets gebruikt, zoals beschreven in de relevante sectie. Elk dataset werd geconverteerd, waarna het naar de grafische hardware geladen werd. Vervolgens zijn er 100 frames per model getekent, uitgaande van de ingeladen data, en de tijd die dit kostte opgeslagen.

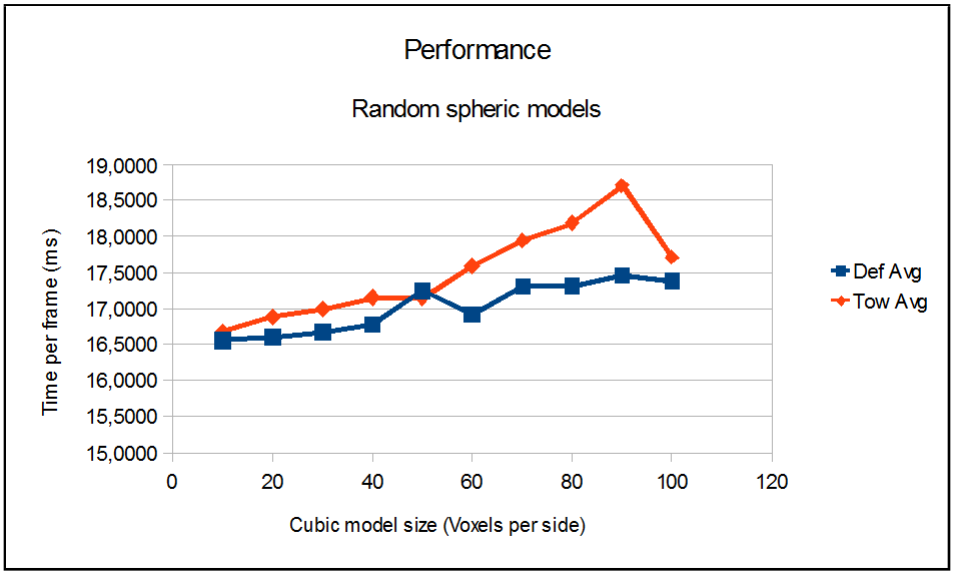
De resulterende grafieken hebben op de x-as het aantal voxels per zijde van het model, en op de y-as de gemiddelde tijd die het kostte om een enkel frame te tekenen. Er zijn weer twee lijnen voor elke grafiek: de lijn voor het basisgeval (Def Avg) en de lijn voor het toren-algoritme (Tow Avg).

De x-as geeft dus het aantal voxels per zijde van het model weer. Alle modellen zijn kubiek gegenereerd. Dit betekent dat het maximum aantal voxels in het model gelijk is aan het aantal voxels per zijde tot de derde macht. Deze as is dus derde-orde kwadratisch.

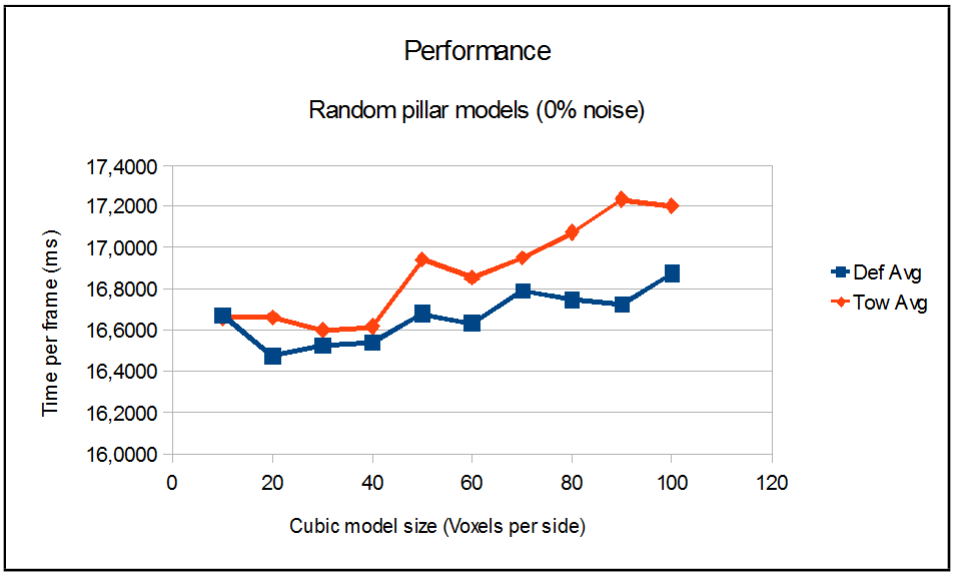
De y-as geeft de gemiddelde tekentijd per frame weer. Dit is gemeten met de systeemafhankelijke hoge-resolutie timer van Windows, die op het systeem gebruikt voor deze tests accuraat is tot op de microseconde.

Elk punt op de grafiek is gemaakt door middel van vijf gegenereerde modellen. Voor elk model en algoritme zijn er 100 frames gemeten. De enige uitzondering hierop is weer het worst-case model, maar deze heeft wel hetzelfde aantal frames.

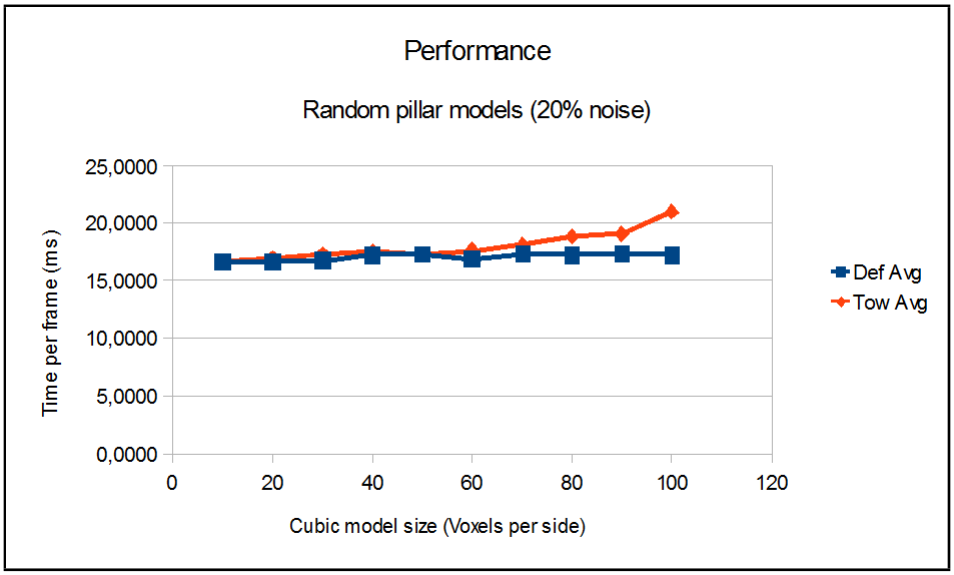
De volgende grafiek is het resultaat van de tests op de datasets bestaande uit bollen:



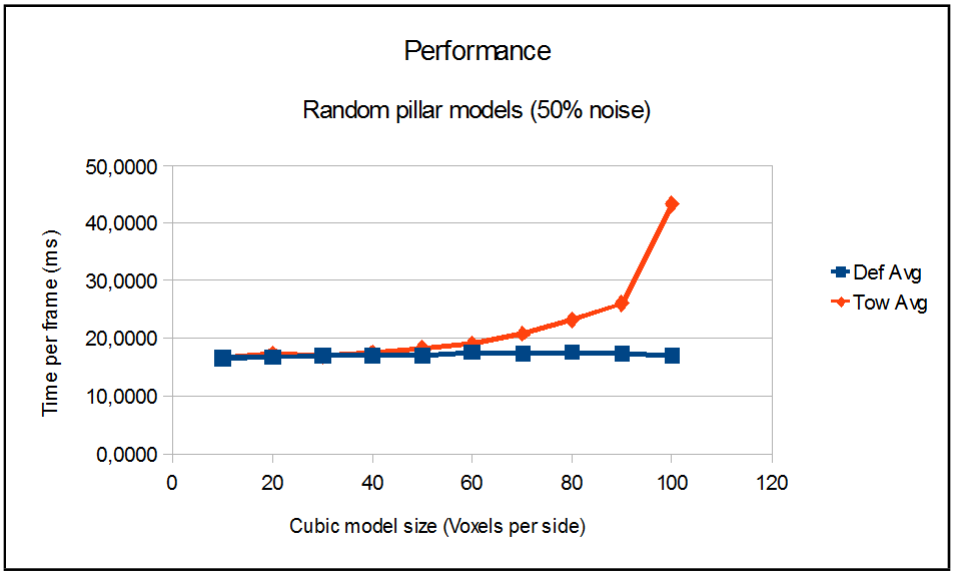
Nu volgen drie grafieken voor de resultaten van de willekeurig gegenereerde pilaar-datasets. Deze eerste grafiek is voor de datasets gegenereerd met 0% ruis:



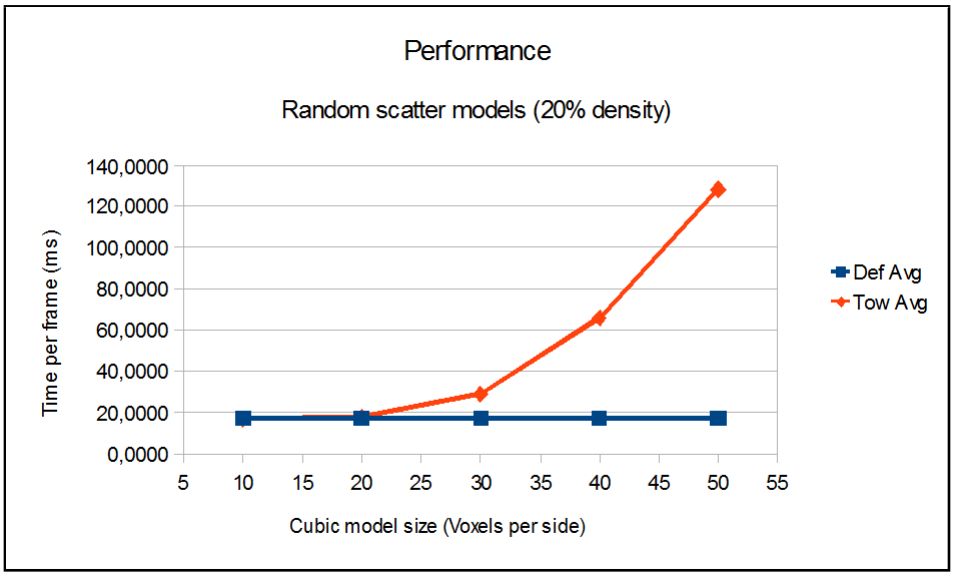
Deze is gegenereerd met 20% ruis:



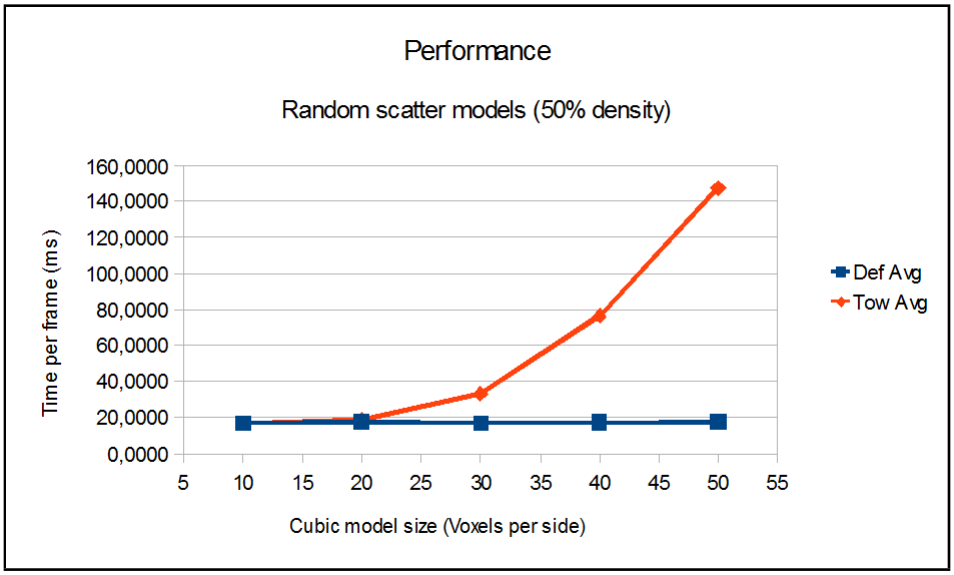
En als laatste de resultaten voor 50% ruis:



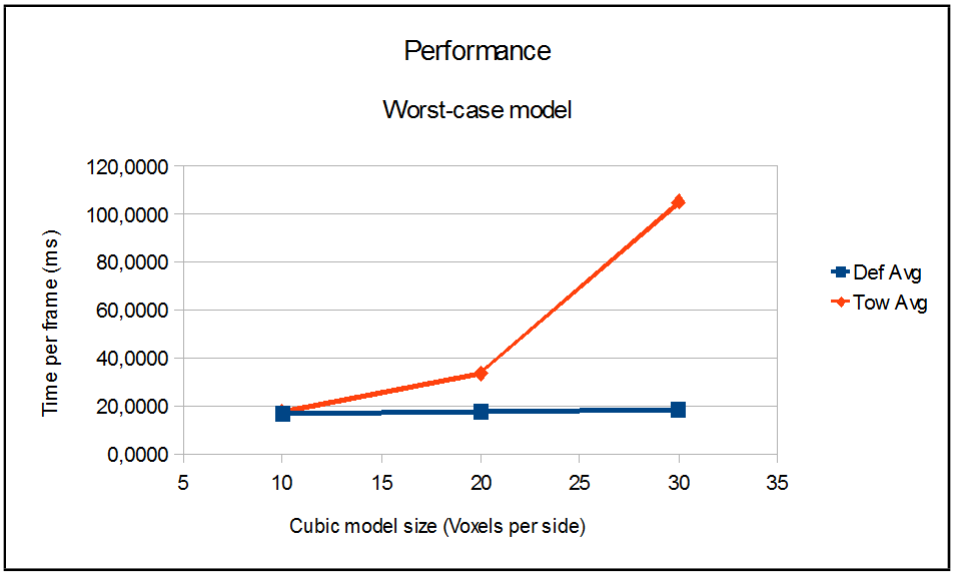
Hier zijn de resultaten voor de verspreidings-datasets met een bepaalde dichtheid. De eerstvolgende grafiek is voor de datasets gegenereerd met een dichtheid van 20%:



En dan de datasets gegenereerd met een dichtheid van 50%:



Als laatste zijn er nog deze resultaten voor het worst-case dataset van variabele grootte:



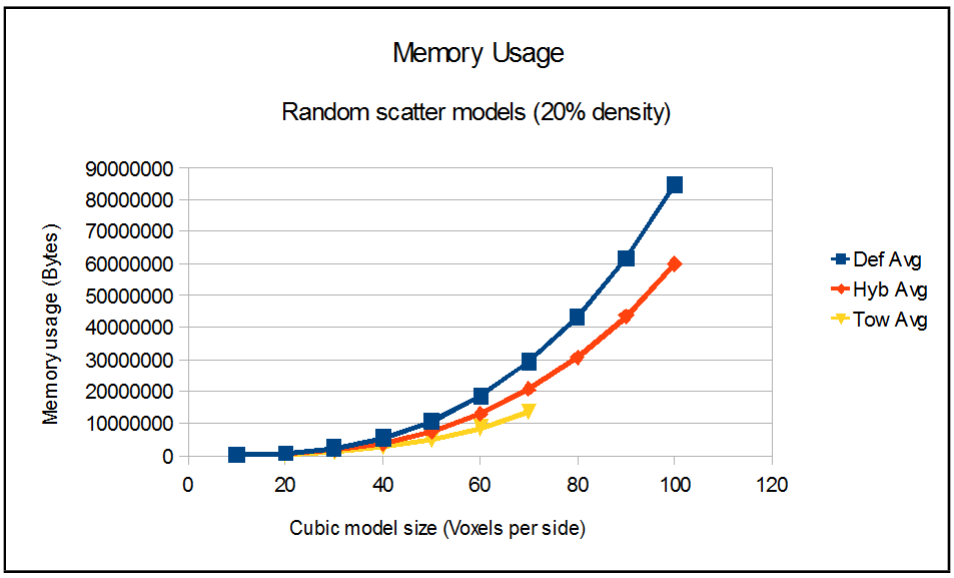
Deze resultaten laten zien dat het toren-algoritme voor een substantieel langere rendertijd per frame zorgt dan het basis-algoritme. De meestwaarschijnlijke oorzaak hiervoor is dat het aantal paden te hoog werd, waardoor de tijd die elke aanroep naar de API kostte snel opliep.

N. B. De latere resultaten gaan niet verder dan weergegeven omdat de rendertijd per frame dan substantieel groter wordt dan 100 milliseconden en het programma daardoor vastloopt.

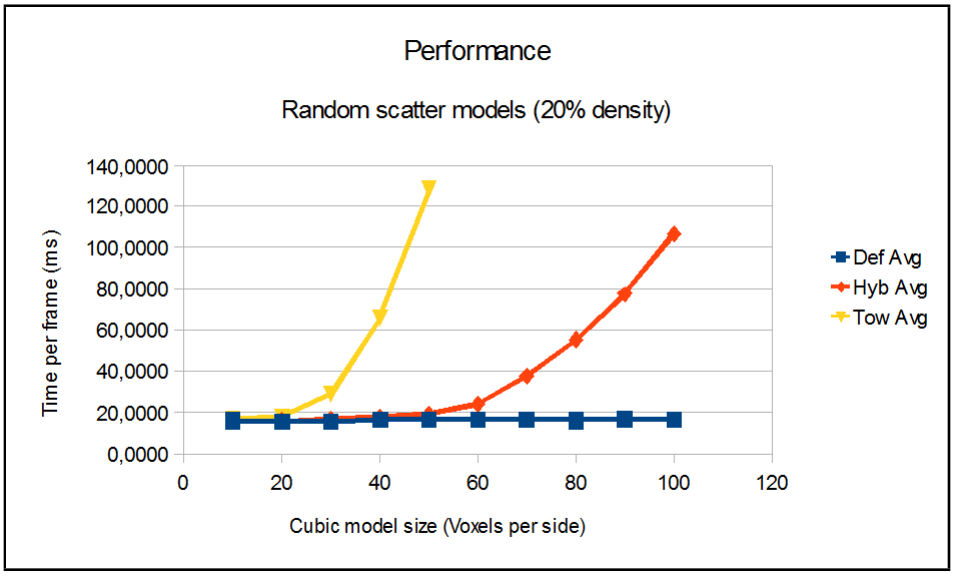
9.3 Hybride Resultaten

Voor de hybride versie van het toren-algoritme, zoals beschreven in de sectie van het Toren-algoritme, zijn er ook enkele tests uitgevoerd op verspreidings-datasets. Om tot deze resultaten te komen zijn alle paden met een lengte kleiner dan of gelijk aan tien vertices samengevoegd. De specificaties van de grafieken zijn hetzelfde als hierboven, de nieuwe "Hyb Avg"grafieken staan nu echter voor het resultaat van de hybride versie.

De resultaten voor het geheugengebruik:



De resultaten voor de rendertijd:



Zoals verwacht liggen de resultaten van de hybride versie tussen de resultaten van de twee algoritmes in. Dit illustreert dus het verschil tussen de twee "extreme" algoritmes en de manieren waarop zij een model reduceren.

10 Toekomstig Onderzoek

Mogelijk toekomstig werk om een Voxel-model tot tristrips te reduceren kan zich focussen op het vinden van een meer optimale oplossing, dus een oplossing die minder paden genereert. Dit kan bijvoorbeeld worden bereikt door het model wat abstracter te maken, zodanig dat er minder paden nodig zijn om een onderdeel te tekenen. Door groepen voxels als een enkel onderdeel te beschouwen, en dus te abstraheren, kan er op deze manier winst worden behaald.

Verder kan er worden gekeken naar methodes om voxels die bij elkaar liggen tegelijk te beschouwen. Als dit enkel in een rechte lijn gebeurt is er echter alleen winst op het gebied van geheugengebruik, terwijl als het ook in de breedte gebeurt er andere paden zijn die juist onderbroken worden, waardoor het aantal paden omhoog gaat. Als hier op de een of andere manier slim mee wordt omgegaan kan er misschien nog winst gemaakt worden.

Een andere mogelijkheid is om naar bestaande algoritmes voor het reduceren van een willekeurig 3D-model te kijken en deze aan te passen voor een Voxel-model. Als er een algoritme is dat goed gebruik kan maken van de structuur van een Voxel-model op deze manier, is er veel winst mogelijk.

11 Conclusies

Het doel van dit onderzoek was om het real-time 3D-renderen van een voxel-model te verbeteren door een algoritme te beschrijven en testen dat een willekeurig voxel-model kon omzetten naar een set van triangle strips om te tekenen. Het idee hierachter was dat, vanwege het feit dat tristrips minder geheugen in beslag nemen en individueel sneller getekent kunnen worden, hierdoor significante verbetering kon worden bereikt op het gebied van video-geheugengebruik en de tijd die het kost om een frame te renderen.

De testresultaten van het basis-algoritme in vergelijking tot het beschreven toren-algoritme wijzen op het gebied van geheugengebruik op een significante verbetering. Omdat er zowel minder vertices als minder tot geen indices worden gebruikt ligt het geheugengebruik bij het toren-algoritme vaak veel lager.

Echter, de resultaten voor de rendertijd per frame laten zien dat het toren-algoritme significant slechter wordt qua performance als de grootte van het model, en dus het aantal paden, toeneemt. De meest waarschijnlijke verklaring voor dit resultaat is dat de afzonderlijke API-functie aanroepen die nodig zijn per tristrip tot een grote toename in rekentijd leiden. Dit is voor het basis-algoritme niet van toepassing, waardoor de rendertijd in vergelijking hiervoor een stuk lager ligt. Het basis-algoritme heeft namelijk maar een enkele aanroep van de API-functie nodig.

De combinatie van deze twee resultaten zorgt ervoor dat het algoritme geintroduceert in dit onderzoek niet geschikt is voor de meeste toepassingen. Echter, als geheugengebruik erg belangrijk is of als de toepassing modellen gebruikt die zo gestructureerd zijn dat het algoritme hier voordeel uit haalt zou het nog gebruikt kunnen worden.

12 Literatuur

12.1 Wetenschappelijke artikelen

1: *Applying Space Subdivision Techniques to Volume Rendering*
K. R. Subramanian, Donald S. Fussell
Published in: Proceedings of the First IEEE Conference on Visualization, 23-26 Oct 1990, p.150-159
doi:10.1109/VISUAL.1990.146377

2: *Octree-R: an adaptive octree for efficient ray tracing*
Kyu-Young Whang; Ju-Won Song; Ji-Woong Chang; Ji-Yun Kim; Wan-Sup Cho; Chong-Mok Park; Il-Yeol Song
Published in: IEEE Transactions on Visualization and Computer Graphics, Dec 1995, Volume 1 issue 4, p.343-349
doi:10.1109/2945.485621

3: *Computational surface flattening: a voxel-based approach*
Grossmann, R.; Kiryati, N.; Kimmel, R.
Published in: IEEE Transactions on Pattern Analysis and Machine Intelligence, Apr 2002, Volume 24 issue 4, p.433-441
doi:10.1109/34.993552

4: *GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering*
Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, Elmar Eisemann
Published in: Proceedings of the 2009 symposium on Interactive 3D graphics and games, 2009, p.15-22
ISBN:978-1-60558-429-4
doi:10.1145/1507149.1507152

5: *Interactive Manipulation of Voxel Volumes with Free-formed Voxel Tools*
Jrg Ayasse, Heinrich Mller
Published in: VMV '01 Proceedings of the Vision Modeling and Visualization Conference 2001, p.359-366
ISBN:3-89838-028-9

6: *Volume rendering for interactive 3D segmentation*
Klaus D. Toennies, Claus Derz
Published in: Proceedings of SPIE Medical Imaging 1997, p.602-609.
doi:10.1117/12.273941

7: *BrainWeb: Online Interface to a 3D MRI Simulated Brain Database*
C.A. Cocosco, V. Kollokian, R.K.-S. Kwan, A.C. Evans
Published in: NeuroImage, vol.5, no.4, part 2/4, S425, 1997 – Proceedings of 3-rd International Conference on Functional Mapping of the Human Brain, Copenhagen, May 1997.

8: *MRI simulation-based evaluation of image-processing and classification methods*
R.K.-S. Kwan, A.C. Evans, G.B. Pike
Published in: IEEE Transactions on Medical Imaging. 18(11):1085-97, Nov 1999.

9: *An Extensible MRI Simulator for Post-Processing Evaluation*
R.K.-S. Kwan, A.C. Evans, G.B. Pike
Published in: Visualization in Biomedical Computing (VBC'96). Lecture Notes in Computer Science, vol. 1131. Springer-Verlag, 1996. p.135-140.

10: *Design and Construction of a Realistic Digital Brain Phantom*
D.L. Collins, A.P. Zijdenbos, V. Kollokian, J.G. Sled, N.J. Kabani, C.J. Holmes, A.C. Evans
Published in: IEEE Transactions on Medical Imaging, vol.17, No.3, p.463–468, June 1998.

11: *Optimal decomposition of polygonal models into triangle strips*
Regina Estkowski, Joseph S. B. Mitchell, Xinyu Xiang
Published in: SCG '02 Proceedings of the eighteenth annual symposium on Computational geometry, p.254-263
ISBN:1-58113-504-1
doi:10.1145/513400.513431

12.2 Informatie

De volgende bronnen bevatten basis-informatie over de onderwerpen die gebruikt worden in dit artikel. Dit zijn zeker niet de enige bronnen, maar ze bevatten wel de kennis gebruikt voor dit onderzoek.

(a) Direct3D reference: MicroSoft Development Network

(b) Voxel, in de context van scanning and imaging: CSI Facility

(c) Analyze file format, voxel ordering informatie: Graham Wideman

(d) Beschrijving Triangle Strips: MicroSoft Development Network